Resilience-Aware Resource Management for Exascale Computing Systems

Daniel Dauwe, *Student Member, IEEE*, Sudeep Pasricha, *Senior Member, IEEE*, Anthony A. Maciejewski, *Fellow, IEEE*, and Howard Jay Siegel, *Fellow, IEEE*

Abstract—With the increases in complexity and number of nodes in large-scale high performance computing (HPC) systems over time, the probability of applications experiencing runtime failures has increased significantly. Projections indicate that exascale-sized systems are likely to operate with mean time between failures (MTBF) of as little as a few minutes. Several strategies have been proposed in recent years for enabling systems of these extreme sizes to be resilient against failures. This work provides a comparison of four state-of-the-art HPC resilience protocols that are being considered for use in exascale systems. We explore the behavior of each resilience protocol operating under the simulated execution of a diverse set of applications and study the performance degradation that a large-scale system experiences from the overhead associated with each resilience protocol as well as the re-computation needed to recover when a failure occurs. Using the results from these analyses, we examine how resource management on exascale systems can be improved by allowing the system to select the optimal resilience protocol depending upon each application's execution characteristics, as well as providing the system resource manager the ability to make scheduling decisions that are "resilience aware" through the use of more accurate execution time predictions.

Index Terms—Exascale resilience, checkpoint restart, multilevel checkpointing, message logging, fault tolerance, HPC resource management

1 Introduction

As the computing power of large-scale computing systems increases exponentially over time, the failure rates of these systems have increased exponentially as well. While most current large-scale computing systems experience failures of some type every few days, projection models indicate that the next generation of these machines will experience failures many times an hour [1]. On average, today's 13.3 petaflop Blue Waters system experiences an application failure every 15 minutes. If these failed applications were not able to be recovered through the use of the traditional checkpoint and restart resilience protocol their failed execution would likely cost the system over $400 K in energy wasted due to unrecoverable failed applications [2]. However, the resilience protocols implemented in today's high performance computing (HPC) and cloud computing systems are impractical at the exascale level, due to their high overheads [3]. Several new promising resilience protocols have recently been proposed for next generation HPC systems [3], [4], [5]. Unfortunately, little work has been done to assess the performance of these emerging protocols on a common computing environment [6]. This not only makes the relative strengths and weaknesses of these protocols unclear, but also makes assessing the underlying assumptions about each protocol's behavior less consistent. Our work provides a methodology for simulating the execution of applications operating at exascale-like system sizes in the presence of uncertainty due to failures across the system. We use our methodology to model an exascale computing environment and utilize this environment to simulate four resilience protocols: one contemporary protocol (checkpointing and restarting) and three protocols proposed for use in future systems (multilevel checkpointing, parallel recovery, and partial redundancy). Using this common environment, we simulate each resilience protocol's performance as greater demands are placed on each one through increasing system utilization for a set of applications with a variety of execution characteristics.

We developed a set of synthetic benchmark applications inspired by an analysis of today's scientific benchmark suites operating at scale [7]. The resulting benchmarks provide the simulated exascale system with a set of applications that have a diverse range of execution characteristics capable of scaling to extreme sizes. We demonstrate how application performance compares when using each resilience protocol and identify the trade-offs present for different combinations of applications and resilience protocols.

We also analyze an exascale-sized system under a typical HPC use-case scenario as it is utilized over a period of days to weeks to service a large number of submissions of applications with a wide variety of execution characteristics. We show the impact that failures and overhead from employing resilience have on this environment and the level of benefit

- *D. Dauwe and A. A. Maciejewski are with the Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO 80523. E-mail: ddauwe@rams.colostate.edu, aam@colostate.edu.*
- *S. Pasricha and H. J. Siegel are with the Department of Electrical and Computer Engineering and the Department of Computer Science, Colorado State University, Fort Collins, CO 80523. E-mail: {sudeep, hj}@colostate.edu.*

Manuscript received 13 Oct. 2017; accepted 7 Jan. 2018. Date of publication 24 Jan. 2018; date of current version 6 Dec. 2018.
(Corresponding author: Daniel Dauwe.)
Recommended for acceptance by A. Louri.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSUSC.2018.2797890

2377-3782 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

that each resilience protocol would provide to such a system. We conclude by utilizing our analyses of resilience protocol performance and trade-offs to demonstrate how resource management can be used to improve system performance by making "resilience-aware" scheduling decisions. The resulting resource management techniques are able to both select the resilience protocol likely to provide the best efficiency for each application (based on the execution characteristics of the application), as well as providing the resource manager with more accurate predictions of application execution time in the presence of failures and overhead from resilience.

In summary, we make the following novel contributions:

- we create a simulation-based methodology capable of modeling and analyzing the execution of applications in an exascale environment;
- we develop a set of exascale synthetic benchmark applications inspired by modern scientific benchmarks;
- we provide a performance comparison of four state-of-the-art HPC resilience protocols (checkpoint restart, multilevel checkpointing, parallel recovery, and partial redundancy) operating over the simulated execution of applications with a diverse range of execution characteristics and sizes;
- we analyze the behavior of a simulated exascale system over an extended period of time when executing many applications under the influence of several strategies for HPC resilience and application scheduling, and demonstrate that overhead from failures and resilience protocols negatively affect system resource management;
- we analyze some of the assumptions made by the parallel recovery protocol [3] and show that, given less optimistic assumptions, our implementation allows for consideration of and alternative response to situations with more severe failures;
- we demonstrate the ability to improve system performance in a large-scale failure-prone system by making the system's resource management techniques "resilience aware."

The remainder of this paper is organized as follows. Section 2 discusses contemporary and proposed resilience protocols, highlights and describes the background of the four protocols compared in this paper, as well as discusses some prior work related to scheduling in failure-prone systems. In Section 3, we describe the modeling methodology we use for our system simulator. Our implementation of HPC resilience protocols is detailed in Section 4. Section 5 analyzes the behavior of the simulated resilience protocols as applications scale to exascale system sizes. Section 6 describes the resource management techniques for our study of a typical system use-case scenario, and the results of these studies are discussed in Sections 7 and 8. We conclude with a summary of this work in Section 9.

2 Related Work

2.1 Overview

Our work broadly covers topics associated with modeling and evaluating the performance a system can achieve when it is oversubscribed (i.e., there are more applications than the system is capable of executing) as well as the performance of HPC resilience protocols in extreme-scale HPC systems. Many important papers have explored these two topics separately, but very few have examined the effects that failures and overhead from resilience have on the system's performance when managing system resources with multiple applications executing in the system.

2.2 HPC Resilience

2.2.1 Background

The work we consider here discusses system-level HPC resilience that allows application programmers and users of the system to be oblivious of the strategies for HPC resilience that are being employed on their behalf. We focus on providing a comparison of several checkpoint-based HPC resilience protocols. Our prior work ([8], [9]) has been among the first efforts to provide an analytical comparison of these protocols in large-scale systems. However, these efforts were less comprehensive than our work here. We have greatly extended our prior work to analyze the impacts of varying workloads and varying application execution characteristics, and to examine trade-offs among resilience strategies in the presence of resource management strategies at exascale system sizes. We acknowledge that other strategies for providing resilience to HPC systems exist and we refer the reader to the surveys of such work, [6], [10]. Our work differs significantly from these high-level surveys by providing simulated comparisons of the performance of the protocols.

All checkpointing-based protocols rely on periodically saving the system's execution state and restarting from an earlier error-free state after the occurrence of a failure [11], [12]. Because recovery from a failure requires these protocols to load a copy of the system state that is not up to date, all checkpointing protocols lose some productivity because of the need to recompute work lost between the time of the failure and the time of the last checkpoint, as well as the time involved with storing and loading those saved system states. We provide a comparison of four HPC resilience protocols that utilize system checkpointing: checkpoint restart, multilevel checkpointing, message logging, and checkpointing combined with redundancy.

2.2.2 Checkpointing and Restarting

Checkpointing is by far the most commonly used resilience protocol employed by today's large-scale computing systems. The most general implementation of the checkpointing protocol operates by stopping the system's execution at regular intervals to save the state of all executing applications to a permanent storage device, typically a parallel file system. Such a protocol is referred to as a blocking, coordinated checkpoint [6].

Several variations and improvements on this protocol have been made since its initial inception. Attempts have been made to create non-blocking or semi-blocking checkpointing that allows the system to continue to execute while checkpoints are saved to permanent storage [13], [14]. Attempts also have been made to allow for uncoordinated checkpoints of the system, preventing the need for all processes in the system to restart when a failure occurs [15].

However, the length of time associated with checkpointing, restarting, and recomputing work lost due to a failure, and the frequency that the system needs to take checkpoints for very large-scale applications when implementing any of these checkpointing protocols, has been shown to be ineffective at managing increasing system sizes [3]. Thus, traditional checkpointing alone is not expected to be capable of providing satisfactory resilience at exascale sizes.

2.2.3 Multilevel Checkpointing

Because different types of failures can affect a computing system in different ways, not all failures require restarting the system from a checkpoint to the parallel file system [16]. Multilevel checkpointing exploits this by providing several levels of checkpointing. A multilevel checkpointing scheme may allow for checkpoints (a) to RAM that are faster but able to recover from fewer types of failures and (b) checkpoints saved to a partner node's RAM that are less frequent but able to recover from more types of failures, in addition to (c) checkpoints saved to the system's parallel file system. Each level offers a trade-off between the time required to checkpoint and/or restart, and the level of failure severity from which the checkpoint can recover [4]. Checkpoint levels may also employ various encoding techniques (such as RAID or Reed-Solomon coding) to improve the resilience offered by a particular checkpoint level [4], [17]. Other attempts have been made to reduce checkpointing's dependence on the parallel file system [18], [19]. One challenge associated with using a multilevel checkpointing protocol is in determining the optimal number of checkpointing levels, and the optimal schedule of checkpoints at each level. Various solutions to this problem have been proposed [4], [20], [21], [22].

2.2.4 Message Logging

Message logging attempts to provide resilience to a system by recording messages sent among processes to create snapshots of the system's execution distributed across system memory [23]. When a failure occurs, the failed node is able to use messages stored in the memory of other system nodes to reduce the amount of rework that is performed by the system when recovering [24]. Using message logging for resilience may save computation time, because the recovering node does not need to wait for the re-computation of other nodes, but rather only for the stored results from the node's computation to be sent. Message logging also saves on the energy used by the system during recovery, because only the failed system node needs to perform re-computation, and the rest of the system can remain idle until the failed node has recovered [25]. The parallel recovery resilience protocol that we consider is an extension to message logging that allows the work that was executing on the failed node to be parallelized across neighboring nodes during recovery [3].

2.2.5 Redundancy

Redundancy improves a system's reliability by executing multiple copies of the same piece of code [26]. It is possible to implement redundancy in either hardware or software [10], but in either case the improved reliability of the system comes at the cost of using additional resources.

Recent attempts allow the system to utilize redundancy in less resource-intensive ways. Dynamic redundancy allows for the executing application to choose a subset of processes for redundant execution [27]. Partial redundancy combines redundancy with checkpointing, and allows for applications to redundantly execute a portion of processes in the system, providing improved resilience for part of the system, using only a portion of the resources [5]. Adaptive process replication attempts to combine partial redundancy with proactive fault tolerance to make better decisions about which processes should be replicated [28].

2.3 HPC Resource Management

2.3.1 Scheduling for HPC Systems

A very large body of work exists on scheduling applications in HPC systems to improve performance and resource utilization. We examine how resource management in an exascale-sized system can be improved when considering the use of value-based (or utility) functions. A large amount of work exists discussing how the use of value functions can improve system resource management [29], [30], [31], [32], [33], [34], [35], [36].

These papers do not examine how the behavior of their resource management techniques change when executing applications at extreme scales, where the probability of system failure negatively affects the quality of allocation decisions. Our work does take this extreme-scale behavior into account in our analyses, and further provides methods for allowing some of these techniques to be "resilience aware" and to make better decisions in the unpredictable exascale environment.

2.3.2 Resilience-Aware Resource Management

Most approaches to provide parallel applications with resource management that is aware of uncertainty in system reliability deal either with scheduling that in some form relies on replicated copies of the executed applications (redundancy) or only with application sizes that are substantially smaller than the size of the system [37], [38], [39], [40], [41], [42]. None address the system-level scheduling problem associated with extreme-sized applications arriving to an exascale-sized system or the use of system-level resilience protocols. Furthermore, our results indicate that replication-based resilience protocols are not capable of performing well when scheduling large applications (discussed in Section 7).

There have been other of resilience-aware resource management in addition our prior work in [9]. For example, the work in [43] analyzes the occurrence of failures in a system and uses that information to construct a method for analyzing the reliability of any given set of nodes. An arriving application can then be scheduled to the set of nodes that are likely to provide it with the best reliability. The work in [44] proposes a resource management technique that redistributes failed applications in the system to assist in minimizing the application's execution time. Through the use of system-level strategies for resilience, our work has an advantage in that it is not resource management technique specific, and can potentially benefit any resource management technique the system designer chooses to implement.

3 Exascale Modeling Methodology

3.1 Overview

Given the impossibility of performing experiments on an exascale system, we have designed an event-based simulator used for modeling systems of arbitrary size [8], [9], [45]. The system experiences randomly generated failures that affect the simulated execution of applications in the system. Throughout the system's simulation an application's execution is affected by events associated with each application's:

- *arrival*: time at which an application arrives,
- *mapping*: process by which the resource management heuristic assigns the application to system nodes,
- *computation*: execution toward application completion,
- *failures*: the failure of a system node,
- *checkpoints*: saving a backup of the application's current computation progress,
- *restarts*: restoring the application progress saved in the last system checkpoint after a failure occurs,
- *recovery*: recomputing progress lost to a failure after the system has restarted,
- *failed checkpoints or restarts*: when failures occur during checkpoint or restart events.

Checkpoints, restarts, and recovery are all resilience-protocol specific events that determine how an application behaves with failures. Each of these events affects applications differently based on the type of resilience protocol employed by the application, the application's execution characteristics, and the characteristics of the failures. This is discussed in detail in Section 4. The remaining events associated with the simulator's management of application arrival, mapping, computation, and failures are all attributes of the system and behave the same regardless of the resilience protocol being used. In particular, while failure events have a large impact on the behavior of the resilience-protocol related events, failure events themselves are a function of the size of the system and the reliability of the system's nodes, and are not affected by the resilience protocol employed.

3.2 Modeling Extreme Scale Applications

To ensure our simulated environment has access to a diverse range of applications that will behave similarly to future exascale applications, we create a set of synthetic benchmarks that vary in their attributes of communication behavior, memory use, and size. We base most of our modeling assumptions for these extreme scale applications on the analysis of the NAS Parallel Benchmark applications [46] performed in [7]. The analysis focused primarily on the Block Tridiagonal (BT) benchmark application, but concluded with a general analysis of the entire NAS Parallel Benchmark suite. The authors determined that, with the exception of the Embarrassingly Parallel (EP) application (which experiences almost no communication), the applications in the benchmark suite all become heavily communication bound at large system sizes. The analysis performed for the BT application indicates that at extreme scales communication began to dominate as much as 80 percent of the application's execution time depending on which parameter set was used for the application's execution.

TABLE 1
Characteristics of Application Types

	memory per node	
communication intensity	32 GB	64 GB
low communication: 0% ($T_C = 0.0$)	L_{32}	L_{64}
high communication: 75% ($T_C = 0.75$)	H_{32}	H_{64}

Similar to the BT application, our synthetic benchmarks are defined with a discrete set of time steps. The number of time steps needed to execute an application is represented by the variable T_S, with identical execution characteristics in each time step. Each benchmark spends some percentage of each time step communicating, represented by the variable T_C, with the remaining portion of each time step spent working on computation, represented by the variable T_W. We assume time steps are one minute in length with, $T_W, T_C \geq 0$, and $T_W + T_C = 1$ minute, thus allowing application execution times to be of arbitrary length (equal to a chosen number of time steps) and unaffected by the application's size. For all simulated studies performed here, applications have between 180 to 1440 time steps giving every executing application an execution time between three hours to a full day when executed without delays from failures or events related to resilience (such as time spent checkpointing). This delay-free execution time is the application's *baseline execution time* and is represented by the variable T_B.

In keeping with the results in [7], our synthetic benchmarks have two levels of communication, $T_C = 0$ (representing a type of application similar to the NAS EP application with little to no communication) and $T_C = 0.75$) representing a similar level of communication seen by the communication dominant applications from the analysis in [7]). Each of the two levels of communication to have two sizes of memory requirements represented by the variable N_m. Applications can have values of $N_m = 32$ GB of memory per node or $N_m = 64$ GB of memory per node. Defining the synthetic benchmarks in this way allows the system access to four application types with a diverse range of communication and memory characteristics. Each of the application types are defined in Table 1.

We assume that all of our synthetic application types exhibit weak scaling so that as the number of nodes used by the application increases with application size. The application's attributes of computation time, communication time, and memory used per node remain constant. Details about the sizes of applications in each simulated study are discussed further in Sections 5, 7, and 8.

For the simulated studies in Sections 7 and 8, each application arriving has an individual *deadline*. Applications that are removed from the oversubscribed system because they could not meet their deadlines are called *dropped applications* and the percentage of total applications that are dropped is one performance metric that we use. Deadline values for each application are selected to be the application's arrival time, T_A, plus the application's baseline execution time multiplied by a random value, $\mathcal{U}$, uniformly selected between 1.5 and 2.5, giving a deadline of

$$T_D = T_A + \mathcal{U}(1.5, 2.5) * T_B. \tag{1}$$

3.3 Simulated System Setup

The simulated exascale system is a homogeneous system inspired by the architecture used to develop China's Sunway TaihuLight supercomputer [47], the world's highest performing system since June of 2016 and still the world's top system as of June 2017 [48]. Each Sunway TaihuLight system node has a multicore architecture composed of four clusters of 64 computational processing elements (CPEs), with each cluster managed by a single management processing element (MPE) that also performs computational work allowing for a total of 65 cores of computation in each core cluster. The four core clusters in a node provide a total of about 3.1 TFLOPs over 260 cores. Our exascale system assumes that the number of CPEs on a node will roughly increase by a factor of four by the time an exascale machine is developed, allowing for a total of 1,028 cores per node providing approximately 12 TFLOPs for each system node. A system composed of 120,000 of these nodes would perform at an exascale level.

The Sunway TaihuLight system has 8 GB of DDR3 RAM at each of its four core clusters, giving each node a total of 32 GB of RAM. We again assume that future systems are likely to have memory increases of about a factor of four in comparison to today's systems, giving our simulated system a total memory capacity of 128 GB per node. In addition to an increase in capacity, we also assume that future memory is likely to utilize newer architectures, allowing for increased aggregate memory bandwidth, B_M. Today's best memories perform at a rate of up to 25 GB/s [49], and we conservatively estimate that future memories will perform at about $B_M = 40$ GB/s.

3.4 System Failure Model

We assume that failures can be characterized by three attributes: the time of the failure's occurrence, the node on which the failure occurs (the location of the failure), and the *severity class* of the failure. We model the uncertainty associated with each attribute using random variables and assume independence between both the individual failure occurrences as well as the attributes of each failure.

The time between system failures is modeled by a Poisson process, a common assumption in failure modeling [50]. Every failure occurs according to the previous failure time ($T_{F_{i-1}}$, with $T_{F_0} = 0$) plus a random variate generated from an exponential distribution $\mathcal{T}_i \sim Exp(\lambda_s)$ with an expected rate of $E[\mathcal{T}_i] = \frac{1}{\lambda_s}$. The parameter λ_s indicates the average failure rate of the entire system, and is defined as the number of nodes in the simulated system that are active, N_a, divided by the mean time between failures (MTBF) of the system nodes, M_n, i.e.,

$$\lambda_s = \frac{N_s}{M_n}. \tag{2}$$

The location of the failure's occurrence represents which system node failed and consequently which application is impacted by the failure. When determining which node has failed, the simulator assumes a uniform random distribution over all active nodes (nodes that are not idle) in the system, and selects one node at random as the failed node.

The severity class of failure corresponds to the type of failure that has occurred in the system. This attribute is used by multilevel checkpointing protocols to determine which level of saved checkpoint is necessary to enable recovery from a specific type of failure and is also used when determining the optimal duration of intervals between checkpoints of different levels. For the implementation of the message logging rollback recovery protocol that we consider, the severity of the failure that occurs is used to determine if it will be possible to use the system's logged messages during recovery, or if (beyond a certain failure class) the system will need to rework from a saved checkpoint without the assistance of message logging. These assumptions and the effect of a failure's class on each resilience protocol's behavior is discussed further in Section 4.

We define the specific mapping of types of failures to classes of failure severities based on the analyses of types of failures present in HPC systems presented in [2] and [51]. We define the probability of experiencing a class j severity failure according to the ratio of the number of failures that occur at each failure severity class, λ_{C_j}, to the total number of failures, λ_{C_t}, measured over an extended interval of time. The resulting discrete set of ratios for each class is used to create a probability mass function from which random variates are sampled to define the severity attribute of each failure. We assume that failure types in a future exascale-sized system will occur in similar relative frequencies to those experienced by today's system, but with failures occurring more frequently.

We define three severity classes. The first class corresponds to failures that could reasonably be recovered from using a checkpoint stored in a node's local RAM (events such as software-related network and file system interrupts). The second failure severity class relates to failures that require a system node to be restarted or replaced and consequently require restarting the application from a checkpoint stored outside of the node, in this case a checkpoint stored in a partner node's RAM. Class two severity failures include hardware failures affecting only a single node or software interrupts that require the node to restart. The third class of failure severity encompasses all other types of failures that cannot be handled by failure classes one and two, including hardware failures affecting multiple nodes and hardware or software events that cause a system-wide outage. The data presented in [2] and [51] details the frequencies of 33 types of failures in the Blue Waters system that caused several hundreds of system interrupts over a timespan of hundreds of days of system execution. Using this information we calculated the probabilities of failure class severities for our three-class failure model to be approximately $\frac{\lambda_{C_1}}{\lambda_{C_t}} = 0.138$, $\frac{\lambda_{C_2}}{\lambda_{C_t}} = 0.784$, and $\frac{\lambda_{C_3}}{\lambda_{C_t}} = 0.078$. Because the Blue Waters system is heterogeneous (with CPUs and GPUs running together) and our simulated system is homogeneous, our calculations of failure severity class distribution excludes failures related to GPUs. Additionally, our failure severity class distribution excludes failures that do not cause system interrupts.

3.5 Communication Model

System communication plays a large role in the behavior and performance of some of the resilience protocols we examine. We assume that future exascale systems are likely

TABLE 2
Resilience Protocol Parameters

parameter	use in modeling
T_S	number of time steps
T_B	application baseline execution time
T_C	portion of each time step spent on communication
T_W	portion of each time step spent on computation work
N_m	memory used by the application
N_a	number of system nodes used by the applications
L	network latency
B_N	communication bandwidth
B_M	memory bandwidth
N_S	number of network switch connections in router
λ_a	application failure rate
M_n	system component MTBF
τ	optimal checkpoint period
$T_{C_{PFS}}$	time required to checkpoint to a parallel file system
$T_{C_{L1}}$	time required for a level one checkpoint
$T_{C_{L2}}$	time required for a level two checkpoint
μ	message logging slowdown
r	degree of redundancy

to have improved communication over today's systems, and base our communication model on the "NDR InfiniBand" network in [52]. Our communication network assumes a latency value of $L = 0.5\ \mu$s, a bandwidth value of $B_N = 600$ GB/s, and a maximum number of simultaneous connections at each switch $N_S = 12$. Further details about the role of communication is discussed for each resilience protocol in Section 4.

4 Resilience Protocol Modeling

4.1 Overview

Four styles of HPC resilience protocols have been implemented in our system simulator. A traditional checkpoint restart based protocol, *checkpoint restart*, as well as three protocols proposed for next-generation computing systems: a multilevel checkpointing approach described in [4], *multilevel checkpoint*; an implementation of message logging based on the work presented in [3], *parallel recovery*; and a protocol combining traditional checkpointing with partial or full redundancy of the executing application from [5], *redundancy*. The following sections present details of how each resilience protocol was modeled, with all relevant parameters summarized in Table 2. Optimal checkpoint interval determination for each resilience protocol is discussed in [53].

4.2 Checkpoint Restart

Our implementation of the checkpoint restart resilience protocol performs periodic blocking uncoordinated checkpointing, with its checkpoints saved to a parallel file system. This checkpointing strategy allows simultaneously executing applications to be checkpointed or restarted independently from one another. This protocol also allows for optimal checkpoint intervals to be defined for individual applications rather than for the system as a whole, which benefits smaller applications that would otherwise experience suboptimal performance if checkpointed at exascale failure frequencies.

The time that the checkpoint restart protocol requires to read and write its checkpoint data to a parallel file system, $T_{C_{PFS}}$, is dependent on application size, memory use, and system parameters for communication. We assume that at any given time the maximum number of nodes from which the parallel file system is capable of receiving data is equal to the maximum number of switch connections that enter the memory system (N_S). Application checkpoints to the parallel file system are typically slowed down by network congestion due to the large number of nodes needing access the parallel file system during a checkpoint. Because the latency associated with checkpoints to the parallel file system is negligible in comparison to the total checkpoint time, we do not consider latency in our equation for checkpoints to the parallel file system. The time for checkpoints to a parallel file system is defined as

$$T_{C_{PFS}} = \frac{N_m}{B_N} * \frac{N_a}{N_S}. \tag{3}$$

Parameters for the applications and environment in this study impose a checkpoint and restart time of between 8-17 minutes for an exascale-sized application depending on the application's type.

The optimal checkpoint period is dependent on the application's checkpoint time and the observed failure rate. Each application's failure rate is given by $\lambda_a = \frac{N_a}{M_n}$.

4.3 Multilevel Checkpointing

The multilevel checkpointing approach from [4] that we implement in our simulator is a three-level checkpointing model. Each checkpointing level offers a trade-off between the time required to save or restore a checkpoint and the severity class of the failure from which it can recover.

The first checkpoint level writes to the node's local RAM, with the time required for taking a level one checkpoint being simply the amount of memory per node required by the application divided by the node's memory transfer rate

$$T_{C_{L1}} = \frac{N_m}{B_M}. \tag{4}$$

The second checkpoint level stores its checkpoints to RAM in a partner node. Application nodes are assumed to be contiguous allowing for minimum latency between checkpoints sent between nodes. The nodes used by the application are partitioned into two groups, with each node in one group having a corresponding partner node in the second group. During a level two checkpoint, the nodes in one group first perform a level one checkpoint (saving the checkpoint data to their own local node) and then send that checkpoint data to their respective partner nodes in the second group. The two groups then switch roles to allow for checkpoint data from the second group to be saved. The time to perform a level two checkpoint is equal to the time required to perform a level one checkpoint, plus the time required to send the data to the partner node, plus the time required to write the data to memory in the partner node. This time is then multiplied by two to account for the time required for both groups of partner nodes to store checkpoint data making the total time for a level two checkpoint equal to

$$T_{C_{L2}} = 2\left(T_{C_{L1}} + L + \frac{N_M}{B_M}\right). \tag{5}$$

The third level checkpoint is written to a parallel file system, and the time required is the same as presented in

Eqn. (3). During a level three checkpoint, a level two checkpoint (and consequently also a level one checkpoint) is also saved by the system. These lower-level checkpoints can be performed in parallel with the level three checkpoint. Additionally, we also assume checkpoint and restart times are symmetric. Failure severity classes are defined according to the outline in Section 3.4.

4.4 Parallel Recovery

The parallel recovery protocol in [3] is an improvement to the message logging resilience protocol, and we base most assumptions about the behavior of a message logging protocol on the behavior of parallel recovery. Parallel recovery allows for faster recovery from a system failure by allowing the failed node's work to be temporarily parallelized across several nodes after being restarted, thereby reducing the time needed by the system to recompute the work lost to a failure. As with all message logging protocols, parallel recovery benefits the system by allowing most of the system to remain idle while only the failed node is recovered. This decreases both the system power needed during recovery as well as the chance that a failure will interrupt the recovering system. However, unlike other message logging protocols, parallel recovery improves checkpointing and restart time by utilizing the in-memory checkpointing protocol outlined in [54]. The in-memory checkpointing protocol behaves similarly to the level two checkpoint to a partner node described in Section 4.3. We therefore used Eqn. (5) to represent the time required for an in-memory checkpoint or restart.

Utilization of the parallel recovery protocol imposes additional overhead involved with message logging, because the system must spend time storing every message that is sent. The amount of overhead an application experiences from message logging, μ, is therefore directly proportional to the amount of communication required by the application. Here we assume this value for our synthetic applications is equal to $\mu = 1 + \frac{T_C}{10}$ which gives a range of values for message logging slowdown that are very close to those listed in [3]. The increase in execution time from message logging increases the application baseline execution time when using parallel recovery to

$$T_B^* = \mu T_S(T_W + T_C) = \mu T_S. \tag{6}$$

The parallelization allowed for a recovering system node in [3] is assumed to be limited to parallelizing across a maximum of eight other system nodes for an estimated $7.26\times$ reduction in recovery time. Through experimentation, we found that for our simulated exascale system the improved performance achievable though node parallelization during recovery starts to produce diminishing returns if parallelized across more than 32 nodes. Based on [3], we allow a recovering application parallelized across 32 nodes to produce an approximate $29.04\times$ estimated reduction in recovery time. The remainder of the parallel recovery specific parameter values are from [3].

The parallel recovery protocol from [3] assumes that all system failures can use in-memory checkpointing, restarting, and stored messages required for parallel recovery. However, because some types of failures can cause multiple simultaneous node failures (i.e., failures of the type belonging to failure severity class three defined in Section 3.4), this assumption is likely to be optimistic if parallel recovery were employed in a real exascale system. In contrast, the implementation of parallel recovery employed by our simulated system allows for applications to utilize the recovery advantages offered by parallel recovery and in-memory checkpointing for failures of severity class one and class two, but requires a restart from the parallel file system for class three severity failures, similar to the behavior of multilevel checkpointing. Aside from the impact that this has on the resilience protocol's performance, this also means that the parallel recovery protocol will operate with two checkpoint levels (one level checkpointing to a partner node's RAM and a second level checkpointing to the parallel file system).

4.5 Partial Redundancy

The partial redundancy protocol in [5] combines the traditional checkpointing protocol with varying degrees of hardware redundancy. "Partial" redundancy is achieved by allowing only a fraction of the total system nodes required by the executing application to have redundant hardware during its execution. For example, a degree of redundancy of $r = 1.5$ dictates that each *virtual process* of an executing application requiring a single node will have at least one physical node performing the application's required computation but half of the virtual processes will have a second physical node performing the same computation. Checkpoints are still taken by the system at regular intervals. When failures occur on nodes in the system, the system only requires a restart from the last checkpoint if failures occur on all (possibly redundant) physical nodes associated with the application's virtual processes before the next system checkpoint.

Apart from the application baseline execution time, all parameters associated with the partial redundancy resilience protocol remain the same as for the checkpoint restart protocol. To account for the increase in application execution time for duplicated communication used by redundancy, based on [5], the communication term in the equation for baseline execution time is scaled by the degree of system redundancy, r,

$$T_B^* = T_S(T_W + rT_C). \tag{7}$$

5 Resilience Protocol Performance with Application Scaling

We utilized our simulation environment to conduct several sets of experiments examining the performance of each resilience protocol. We evaluated the performance of each of the four application types defined in Table 1, with each application type being scaled in size from one percent of the exascale system (about 1.2 million CPU cores, similar in size to some of today's largest applications) through to an exascale-sized application requiring 123 million CPU cores. For these experiments, the baseline execution time for each application is defined as $T_B = 1440$ minutes (one full day).

As systems trend towards manycore architectures, with hundreds or thousands of CPU cores on a single socket, component failure rates are likely to increase [55], [56]. Our simulated exascale system assumes an approximate $4\times$

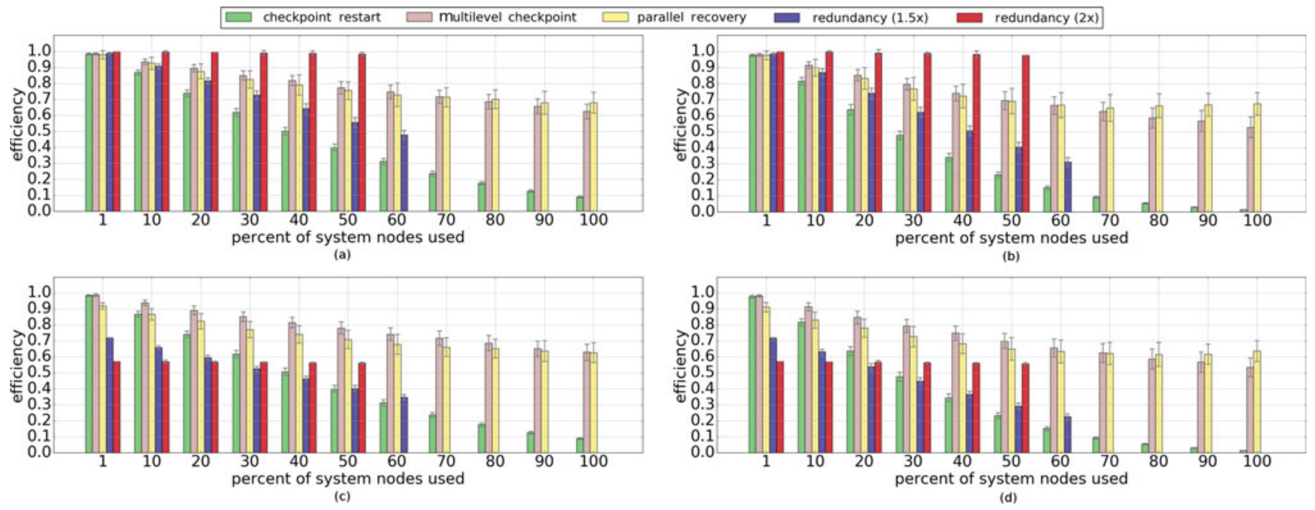


Fig. 1. Resilience protocol efficiency at increasing percentages of total system use, (a) by the low memory use and low communication application defined in Table 1 as L_{32}, (b) by the high memory low communication application (L_{64}), (c) by the low memory high communication application (H_{32}), and (d) by the high memory high communication application (H_{64}). Efficiency is defined to be the ratio of an application's execution time without slowdowns (from failures or checkpointing) over the application's execution time with slowdowns (from failures and checkpointing). Processors in the system experience a 2.5 year MTBF. Each bar in the figure represents the average of 200 trials. Standard deviations are shown for each bar.

increase in the number of CPU cores per processor over the Sunway TaihuLight system, which is likely to decrease processor reliability and increase the likelihood of failures in the system. Most of the works we consider assume a node mean time between failures of ten years for current HPC systems. We assume component failure rates will increase linearly with the increased size of system nodes. For our experiments, we assume an MTBF of $M_n = 2.5$ years.

Fig. 1 highlights the results from this set of experiments analyzing execution efficiency for varying application sizes. Efficiency is defined to be the ratio of an application's baseline execution time over the application's execution time with slowdowns from failures or resilience protocol overhead delay, e.g., due to checkpointing. Each bar in the figure represents the average of 200 trials simulated with the uncertainty of randomly occurring failures. The standard deviation of these trials are shown around each bar.

The data in Figs. 1a-d depicts the efficiency of each of the applications in Table 1 as the size of each application increases. This is indicated on the x-axis of each figure as an increase in the percentage of system nodes occupied.

Analyzing trends when comparing the two high-communication applications (Figs. 1c and 1d) to the two low-communication applications (Figs. 1a and 1b), it can be seen that both the parallel recovery protocol and the two redundancy protocols suffer a larger decrease in efficiency for all application sizes when the application has a higher amount of communication than the checkpoint restart or multilevel checkpoint protocols suffer. This decrease is due to the parallel recovery and redundancy protocol's higher reliance on communication.

An increase in application memory use increases the time required to perform checkpoints. Because the checkpoint restart, multilevel checkpointing, and $1.5\times$ redundancy protocols rely more heavily on checkpoints (particularly to the parallel file system) than the parallel recovery and $2\times$ redundancy protocols, results observed when comparing the high-memory applications (Figs. 1b and 1d) to the low-memory applications (Figs. 1a and 1c) indicate that the checkpoint restart, multilevel checkpointing, and $1.5\times$ redundancy protocols are more negatively impacted by an application's memory use.

One trend seen throughout Fig. 1 is a trade-off among which resilience protocol provides the best performance for any size of the given application type. For low communication applications (Fig. 1a and 1b) occupying at most half of the system, the $2\times$ redundancy protocol allows for the best application performance. However, because of their costly need for additional system resources, the redundancy protocols provide zero efficiency when the application is scaled above certain applications sizes because there are not enough nodes available in the system to employ either redundancy protocol. The multilevel checkpointing and parallel recovery protocols also change between which protocol is optimal as application sizes increase in all but the low-memory high-communication application (Fig. 1c). Results in the figure indicate that the multilevel checkpointing protocol tends to dominate over the parallel recovery protocol for application occupying between 1 to 50 percent of the system (depending on the application type), but the parallel recovery protocol tends to dominate multilevel checkpointing for larger application sizes. There are no situations in which checkpoint restart or $1.5\times$ redundancy resilience protocols are optimal. These results motivate the need for an adaptive strategy to achieve low-overhead resilience.

6 System Resource Management

6.1 Overview

We explore the behavior of several techniques for resource management operating in a system with failures and considering overhead from employing resilience techniques. Each resource management technique takes as input the set of unmapped applications and idle system nodes (nodes that are not currently executing an application) at a mapping event and outputs a mapping of applications to system nodes. System mapping events occur immediately after an application arrives to the system as well as immediately after an application finishes its execution. If not enough idle system nodes are available during the mapping event to accommodate all unmapped applications, then the remaining applications stay in the set of unmapped applications until they are either scheduled during a future mapping event or reach their deadlines and are removed from the system.

6.2 FCFS Technique

First come first served (FCFS) is the most commonly employed resource management technique in HPC systems and it is therefore both important to assess its behavior in an exascale system and also an important point of comparison for other resource management techniques. This technique operates by scheduling applications from the set of unmapped applications in the order that they arrive to the system until there are not enough nodes left for the most recently arrived application. Applications not assigned to nodes are scheduled in a future mapping event.

6.3 Random Technique

The *random* resource scheduling technique randomly selects an application from the set of unmapped applications and assigns it to execute on any available set of nodes able to accommodate the application's size. If not enough nodes are available, then the application is returned to the set of unmapped applications for consideration in the next mapping event. This process is repeated until the set of mappable applications is empty.

6.4 Slack-Based Technique

An application's slack is calculated as the application's deadline minus the sum of its baseline execution time and its time of arrival to the system. The slack-based resource management technique allows the system a means to prioritize applications based on the application's execution time and deadline. The set of unmapped applications is sorted based on each application's slack value. A negative slack value indicates that an application will not be able to complete execution before its deadline, and it is "dropped" from the system. The technique schedules applications to nodes in the system ordered based on increasing slack. Applications that cannot begin execution immediately are returned to the set of unmapped applications. The slack-based scheduler continues evaluating applications until all applications are either executing in the system or have been returned to the set of unmapped applications to be considered in future mapping events.

6.5 Value-Based Techniques

Typically, not all applications that arrive to a system have the same importance. Furthermore, because the results of executing applications are typically more valuable if they are processed quickly, the benefit that an HPC system can provide to a set of users can typically be improved if these aspects are kept in consideration when making resource management decisions. To more accurately analyze system performance when considering the time-varying nature and importance of applications, we use *value functions*. A value function is a monotonically decreasing function defined for each application that arrives to the system and is commonly used to quantify an application's importance based on information associated with the application's expected execution time and resource requirements [29].

For the work performed here, each application's time-varying value is a function of the application's execution time, size, and the time since the application's arrival. Each application has a starting value, V_S, that is proportional to both an increase in the application's size and execution time

$$V_S = \left(100\frac{N_a}{N_S}\right) * \left(\frac{T_S}{360}\right). \tag{8}$$

An application that arrives to the system is awarded its starting value if it completes execution by a specified "soft" deadline, T_{D_S}, defined by its arrival time and execution time

$$T_{D_S} = T_A + 1.5T_B. \tag{9}$$

Applications unable to complete execution by their soft deadlines experience a linear depreciation in value defined by the current time of the simulation, C_T, the application's hard deadline defined in Eqn. (1), its soft deadline defined in Eqn. (9), and its starting value from Eqn. (8). The value earned by the application if it completes execution after its soft deadline, V_F, is equal to

$$V_F = V_S + -\frac{\frac{3}{5}V_S}{T_D - T_{D_S}}(C_T - T_{D_S}). \tag{10}$$

Defining the value function in this way implies that an application's final value at the time it reaches its hard deadline will have depreciated down to a fraction of its starting value, which is consistent with other work utilizing value-based heuristics. Applications that reach their hard deadlines without completing execution are removed from the system and earn the system zero value.

Our resource management techniques consider four scheduling heuristics based on [35] to analyze the value that will be earned by the application for successfully completing execution. We consider heuristics that prioritize scheduling applications by:

- *Max Value*: the total value that will be earned by the application upon its completion
- *Max VPT*: the value able to be earned by each application per unit time
- *Max VPN*: the value able to be earned by each application per node
- *Max VPR*: the value able to be earned by each application per resource required by the application (a product of the application's execution time and number of nodes)

7 Resilience Protocol Effects on Resource Management

In practice, it is unlikely that exascale systems will always be used for executing a single exascale-sized application. Instead, in many cases such systems will spend the majority of their time executing a larger number of smaller applications. We explore the behavior of an exascale-sized system under this more typical use-case scenario as the system is utilized over a period of several days to service a large

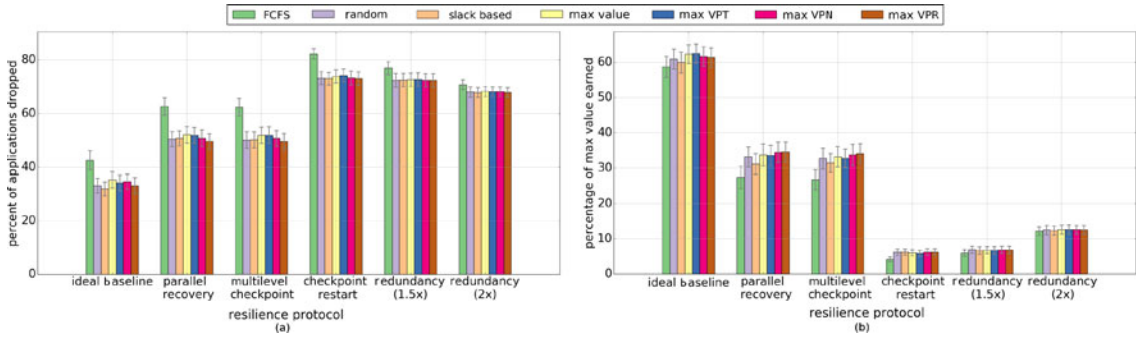


Fig. 2. Performance of the system by: (a) percentage of applications dropped from the system, and (b) percentage of the maximum value earned by the system, for each resilience protocol and resource management technique combination. Bars in the figure represent the average of 50 arrival patterns. Standard deviations are shown for each bar.

number of petascale sized applications with a wide variety of execution characteristics. We show the impact that failures have on this environment and the level of benefit that is provided to the system by each resilience protocol.

We assume the exascale environment is oversubscribed, meaning there are always more applications submitted to the system needing to be executed than the system has the capacity to execute. Oversubscription is typical in most HPC environments. Our experience working with Oak Ridge National Labs and the DoD has been that their large-scale systems are almost always oversubscribed [34], [36]. Furthermore, because an undersubcribed system is never at risk of having applications that are unable to execute to completion, the impact of failures and resilience on the performance of the system at any given time will simply be a function of the system's utilization and can be inferred from our analyses in Section 5. We therefore provide an analysis on performance in an oversubscribed system that is constrained by requiring individual applications to meet deadlines as defined in Section 3.3.

Each simulation begins by filling the entire exascale system with applications, forcing the system to begin operation at full utilization. Applications then arrive to the system randomly according to a Poisson process with a mean arrival time of six hours until a total of 500 applications have arrived to the system. Each application that arrives to the system is uniformly randomly selected from the set of four application types discussed in Table 1. Baseline execution times for each arriving application are uniformly randomly selected to be either three, six, twelve, or twenty-four hours in length. The number of system nodes required by each arriving application is uniformly randomly selected to use between one to one-hundred percent of the exascale system, based on the eleven application sizes experimented with in Section 5. The processor MTBF for these studies is 2.5 years.

Each set of 500 applications that arrives to the simulated system is referred to as an *arrival pattern*. Fifty different such arrival patterns were created. The behavior of each resilience protocol and resource management technique was examined using the same set of arrival patterns for fair comparisons.

We compare the five resilience protocols by averaging the results of 50 arrival patterns for each experiment and comparing those values to the average results of an *Ideal Baseline* that executes without delays from failures or delays associated with overhead from resilience protocols. Each resource management technique and resilience protocol combination is assessed in terms of both its ability to minimize the percentage of applications that are dropped from the system, shown in Fig. 2a, as well as their ability to maximize value earned by the system, shown in Fig. 2b.

In comparison to the performance of the Ideal Baseline, the results from these simulated studies in Fig. 2 show how the presence of failures and overhead from resilience protocols negatively impacts system performance by both increasing the percentage of dropped applications in the system and earning the system less value regardless of which resilience protocol is utilized. The results also demonstrate the superiority of the multilevel checkpointing and parallel recovery resilience protocols over checkpoint restart and both redundancy-based protocols in an oversubscribed system. However, the performance of multilevel checkpointing and parallel recovery are very similar in their ability to both maximize value and minimize the number of dropped applications. Although the results from Section 5 indicate that the $2\times$ redundancy protocol is capable of providing the best efficiency to some applications, the results in Figs. 2a and 2b demonstrate that while a particular resilience protocol may allow certain application types the best performance in some situations, it may not be suitable for more typical system use cases of oversubscribed systems where resources are in demand by many applications. Because a large number of the proposed resilience protocols discussed in Section 2 rely on some form of redundancy, these results imply that there is a substantial constraint on the utility that can be expected from these proposed works.

Another observation that can be made from these results is that, with the exception of FCFS which always performs worse than the other resource management techniques, when using the checkpoint restart or redundancy-based resilience protocols the set of applications that are able to be successfully executed by the system is so low that it reduces the decision making ability of the resource management techniques, making their performance very similar. The multilevel checkpoint and parallel recovery protocols see a greater variability between the performance of each resource management technique. However, the performance of the multilevel checkpoint and parallel recovery protocols themselves are consistent with the results in Section 5.

8 Resilience-Aware Resource Management

The implication from the results of our studies in Sections 5 and 7 is that there is a potential for improving system

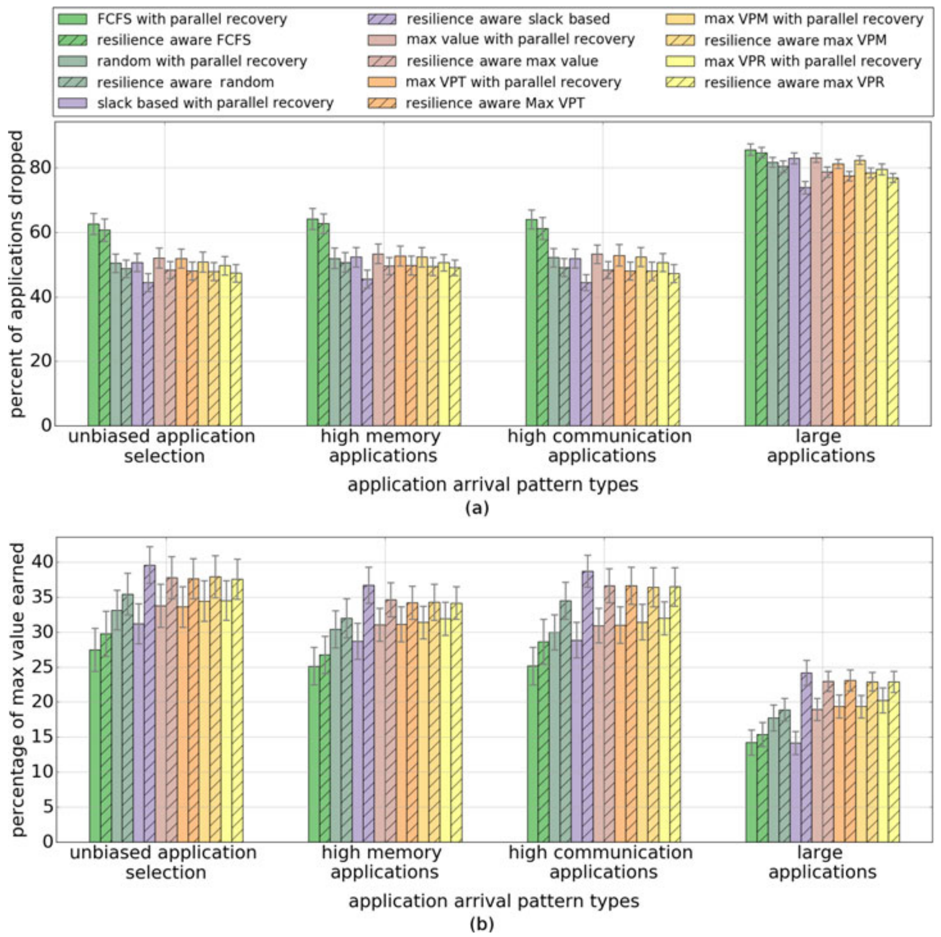


Fig. 3. Performance of the system by (a) percentage of applications dropped from the system, and (b) percentage of the maximum value earned by the system, for each resource management technique when resilience-naïve and using the parallel recovery resilience protocol, and each resource management technique when resilience-aware and employing resilience-selection. Groupings of bars show four different types of application arrival patterns. Bars in the figure represent the average of 50 arrival patterns. Standard deviations are shown for each bar.

performance by providing the system a means of making scheduling decisions that are aware of the negative impact that system failures and overhead associated with resilience protocols has on application execution. In addition to deciding when and on what nodes an application will execute, the system resource manager intelligently selects the resilience protocol that is most likely to provide the best performance for each application. The optimal application-specific resilience protocol choice can be made by comparing the efficiencies each application is expected to achieve based on predictions made for each resilience protocol by the execution time prediction equations described in [53]. We call this resilience-awareness feature *resilience selection*, and it allows the system to take advantage of the trade-offs between resilience protocol optimality described in Section 5 regardless of the resource management technique being employed.

If applications are being scheduled in the system using either the slack-based resource management technique or any of the value-based resource management techniques, then the system has the additional advantage of being able to use these same equation-based execution time predictions to make scheduling decisions that are also aware of the decreased performance that the application will experience in a system with failures. While the results shown in Fig. 1 indicate that some application types perform most efficiently when utilizing $2\times$ redundancy, the results from Section 7 demonstrate that in oversubscribed systems, such as the one considered here, use of redundancy has a negative impact on the overall system performance. Use of redundancy is therefore excluded from consideration when resource managers are utilizing resilience selection.

Simulated studies exploring resilience-aware resource management have a similar setup to Section 7. However, in addition to the unbiased application arrival pattern seen in Section 7 that allows for a uniformly random selection of applications of different sizes and types, these studies also experiment with arrival patterns that are biased toward:

- high memory applications requiring $N_m = 64\,\text{GB}$;
- high communication applications having communication values of $T_C = 0.75$;
- large applications that occupy at least fifty percent of the exascale system.

These application arrival pattern types were chosen because they are likely exist in exascale environments [7]. Our results depict the average percentage of applications dropped in each executed arrival pattern in Fig. 3a and the average percentage of maximum value earned by the system for each arrival pattern in Fig. 3b. Results are shown for each resource management technique from Section 6 when utilizing the parallel recovery resilience protocol (indicated by each of the bars without hash marks in the figures) because it is most consistently the best performing resilience protocol. Each resource management technique utilizing parallel recovery is also compared to execution of the same set of arrival patterns when each technique is resilience-aware both in its prediction of execution times and in the resilience protocol that is selected to provide the application the best performance possible. Resilience-aware results are indicated by the hashed bars in the figures.

In general, the results shown in the figures demonstrate that in all cases the resilience-aware resource management techniques improve upon their resilience-naïve counterparts by several percent, and in some cases the improvement is substantially more. Because scheduling decisions made by the FCFS and Random resource management techniques do not rely on execution time predictions, the relatively small improvements they gain when resilience-aware in comparison to the slack-based and value-based resource management techniques demonstrate that the improvement achievable using only resilience protocol selection is not as great as that achievable when also allowing for execution time predictions.

Even though the high-memory and high-communication sets of application arrival patterns were expected to prove more challenging for the resource management techniques, they both perform similarly to the unbiased set of application arrival patterns. Unsurprisingly, arrival patterns biased toward large applications perform notably worse than the other arrival pattern types because they require more system resources. But the large application arrival patterns still benefit by a similar amount from being resilience-aware as the other three arrival pattern types.

Results also indicate that, for all metrics and for all application arrival patterns, the resilience-naïve slack-based and four resilience-naïve value-based resource management techniques using parallel recovery generally only perform within 1 to 2 percent of the resilience-naïve Random resource management technique using parallel recovery due to inaccurate execution time estimates. However, allowing these techniques to perform "resilience-aware" on the same application arrival patterns enables every technique to earn the system much more value (in some cases as much as 10 percent more). In the case of the value-based techniques, this improvement comes by allowing each technique to more accurately assess the relative values achievable when scheduling different applications. However, even though there are differences in performance between the resilience-naïve value-based techniques the resilience-aware value-based techniques, all resilience-aware techniques tend to perform similarly within each arrival pattern type, indicating that if resilience-aware predictions are used then the type of value-based heuristic used by the system does not matter much. Improvement in value for the slack-based technique occurs as a bi-product of the resilience-aware technique being able to more accurately assess which applications are closest to their deadlines, and respond by successfully preventing more of them from being dropped.

The performance of the slack-based resource management techniques in particular demonstrates the benefit that can be obtained when providing certain resource management techniques with resilience information during scheduling. For all application arrival pattern types, the slack-based resource management techniques improve from being one of the worst-performing techniques when resilience-naïve (not even able to out-perform the Random technique), to clearly being among the best performing technique in all metrics by a significant margin when resilience-aware. The findings from our study motivate the design of a new class of resource management techniques that consider resilience protocol selection as an integral part of their decision making.

9 Conclusions

HPC resilience has become an increasingly important topic as we approach exascale system sizes. It has also become increasingly important that the resilience protocols that are proposed for use in these systems are analyzed in a common computing environment. This work is one of a few studies that tests a variety of new HPC resilience protocols in such a manner. We describe a methodology that can be used to simulate exascale HPC system sizes with a diverse set of applications able to scale to arbitrary sizes.

We utilize our simulation models to evaluate four protocols for HPC resilience, i.e., the traditionally employed checkpoint restart protocol, as well as three techniques proposed for next generation large-scale systems: multilevel checkpointing, parallel recovery, and partial redundancy. Our analysis indicates that each resilience protocol has performance trade-offs that vary based on application execution characteristics.

Because a production exascale system is unlikely to execute only exascale sized applications, we study the effects that HPC resilience and system failures have on resource management for a workload consisting of several petascale applications. Our results indicate that while multilevel checkpointing and parallel recovery are likely to be the best performing resilience protocols, for all of the resource management techniques we consider there is still a significant decrease in system performance due to failures and overhead from resilience protocols. However, we also show that the system performance can be improved by using resilience-aware resource management techniques that schedule applications to nodes, select the resilience protocol used for each application, and use execution time predictions to address both the uncertainties introduced by system failures and the overhead introduced by resilience.

Acknowledgments

The authors thank Dylan Machovec for his valuable comments on this research. This work was supported by the NSF under grants CCF-1252500 and CCF-1302693. This work utilized CSU's ISTeC Cray system, which is supported by the NSF under grant number CNS-0923386. The authors thank Hewlett Packard (HP) of Fort Collins for providing them with some of the machines used for testing. A preliminary version of portions of this work appeared in [9]. The additions to this work include the creation of four new resource management heuristics (used for both better assessing the effects that failures and resilience have on scheduling as well as the benefit that can be achieve by allowing a system to be resilience-aware), utilization of a resilience-aware application execution time prediction model (to both improve resilience-aware resource management and optimal checkpoint interval

selection), a closer examination of some assumptions made in other works resulting in adaptations of those resilience protocols that more accurately reflect their likely behavior of a real system, and a much broader and extensive examination of related work in the field.

REFERENCES

[1] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *Int. J. HPC Appl.*, vol. 23, pp. 212–226, Aug. 2009.

[2] C. D. Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs," in *Proc. IEEE Int. Conf. Dependable Syst. Netw.*, Jun. 2015, pp. 25–36.

[3] E. Meneses, X. Ni, G. Zheng, C. Mendes, and L. Kalé, "Using migratable objects to enhance fault tolerance schemes in supercomputers," *IEEE Trans. Parallel Distrib. Syst. Syst.*, vol. 26, no. 7, pp. 2061–2074, Jul. 2015.

[4] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proc. Int. Conf. HPC Netw. Storage Anal.*, Nov. 2010, pp. 1–11.

[5] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for HPC," in *Proc. Int. Conf. Dist. Comput. Syst.*, Jun. 2012, pp. 615–626.

[6] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing Frontiers Innovations*, vol. 1, pp. 5–28, Jun. 2014.

[7] R. F. Van der Wijngaart, S. Sridharan, and V. W. Lee, "Extending the BT NAS parallel benchmark to exascale computing," in *Proc. Int. Conf. HPC, Netw. Storage Anal.*, Nov. 2012, pp. 94:1–94:9.

[8] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, "A performance and energy comparison of fault tolerance techniques for exascale computing systems," in *Proc. 6th IEEE Int. Symp. Cloud Serv. Comput.*, Dec. 2016, pp. 436–443.

[9] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, "An analysis of resilience techniques for exascale computing platforms," in *Proc. 19th Workshop Adv. Parallel Distrib. Comput. Models*, May 2017, pp. 914–923.

[10] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *J. Supercomputing*, vol. 65, pp. 1302–1326, Sep. 2013.

[11] D. P. Jasper, "A discussion of checkpoint restart," *Softw. Age*, vol. 3, pp. 9–14, Oct. 1969.

[12] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, pp. 530–531, Sep. 1974.

[13] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking versus non-blocking coordinated checkpointing for large-scale fault tolerant MPI," in *Proc. Conf. Supercomputing*, Nov. 2006, Art. no. 127, pp. 1–13.

[14] X. Ni, E. Meneses, and L. V. Kalé, "Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm," in *Proc. Int. Conf. Cluster Comput.*, Sep. 2012, pp. 364–372.

[15] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic MPI applications," in *Proc. Int. Par. Dist. Proc. Symp.*, May 2011, pp. 989–1000.

[16] N. H. Vaidya, "A case for two-level distributed recovery schemes," in *Proc. ACM SIGMETRICS Conf. Measurement Modeling Comput. Syst.*, May 1995, vol. 23, pp. 64–73.

[17] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *Proc. Int. Conf. HPC Netw. Storage Anal.*, Nov. 2011, pp. 32:1–32:32.

[18] K. Mohror, A. Moody, G. Bronevetsky, and B. de Supinski, "Detailed modeling and evaluation of a scalable multilevel checkpointing system," *IEEE Trans. Parallel. Distrib. Syst.*, vol. 25, no. 9, pp. 2255–2263, Nov. 2014.

[19] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed diskless checkpoint for large scale systems," in *Proc. Int. Conf. Cluster Cloud Grid Comput.*, May 2010, pp. 63–72.

[20] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello, "Optimization of multi-level checkpoint model for large scale HPC applications," in *Proc. Int. Parallel Distrib. Proc. Symp.*, May 2014, pp. 1181–1190.

[21] S. Di, Y. Robert, F. Vivien, and F. Cappello, "Toward an optimal online checkpoint solution under a two-level HPC checkpoint model," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 244–259, Jan. 2017.

[22] A. Benoit, A. Cavelan, V. L. Fvre, Y. Robert, and H. Sun, "Towards optimal multi-level checkpointing," *IEEE Trans. Comput.*, vol. 66, no. 7, pp. 1212–1226, Jul. 2017.

[23] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 63–75, Feb. 1985.

[24] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using asynchronous message logging and checkpointing," in *Proc. Symp. Principles Distrib. Comput.*, Jan. 1988, pp. 171–181.

[25] E. Meneses, O. Sarood, and L. V. Kal, "Energy profile of rollback-recovery strategies in high performance computing," *Parallel Comput.*, vol. 40, pp. 536–547, Oct. 2014.

[26] J. F. Wakerly, "Microcomputer reliability improvement using triple-modular redundancy," *Proc. IEEE*, vol. 64, no. 6, pp. 889–895, Jun. 1976.

[27] S. Hukerikar, P. C. Diniz, and R. F. Lucas, "A case for adaptive redundancy for HPC resilience," in *Proc. Workships Euro-Par*, Aug. 2014, pp. 690–697.

[28] C. George and S. Vadhiyar, "Fault tolerance on large scale systems using adaptive process replication," *IEEE Trans. Comput.*, vol. 64, no. 8, pp. 2213–2225, Aug. 2015.

[29] E. D. Jensen, C. D. Locke, and H. Tokuda, "A time-driven scheduling model for real-time operating systems.," in *Proc. Real-Time Syst. Symp.*, Dec. 1985, vol. 85, pp. 112–122.

[30] B. Ravindran, E. D. Jensen, and P. Li, "On recent advances in time/utility function real-time scheduling and resource management," in *Proc. Int. Symp. Object-Oriented Real-Time Distrib. Comput.*, May 2005, pp. 55–60.

[31] K. Chen and P. Muhlethaler, "A scheduling algorithm for tasks described by time value function," *Real-Time Syst.*, vol. 10, pp. 293–312, May 1996.

[32] M. Kargahi and A. Movaghar, "Performance optimization based on analytical modeling in a real-time system with constrained time/utility functions," *IEEE Trans. Comput.*, vol. 60, no. 8, pp. 1169–1181, Aug. 2011.

[33] C. B. Lee and A. E. Snavely, "Precise and realistic utility functions for user-centric performance analysis of schedulers," in *Proc. 16th Int. Symp. High Perform. Distrib. Comput.*, Jun. 2007, pp. 107–116.

[34] B. Khemka, R. Friese, S. Pasricha, A. A. Maciejewski, H. J. Siegel, G. A. Koenig, S. Powers, M. Hilton, R. Rambharos, and S. Poole, "Utility maximizing dynamic resource management in an oversubscribed energy-constrained heterogeneous computing system," *Sustainable Comput.: Inform. Syst.*, vol. 5, pp. 14–30, Mar. 2015.

[35] D. Machovec, B. Khemka, N. Kumbhare, S. Pasricha, A. A. Maciejewski, H. J. Siegel, A. Akoglu, G. A. Koenig, S. Hariri, C. Tunc, M. Wright, M. Hilton, R. Rambharos, C. Blandin, F. Fargo, A. Louri, and N. Imam, "Utility-based resource management in an oversubscribede energy-constrained heterogeneous environment executing parallel applications," *Parallel Comput.*, pp. 1–26, accepted 2017, to appear.

[36] B. Khemka, R. Friese, L. D. Briceno, H. J. Siegel, A. A. Maciejewski, G. A. Koenig, C. Groer, G. Okonski, M. M. Hilton, R. Rambharos, and S. Poole, "Utility functions and resource management in an oversubscribed heterogeneous computing environment," *IEEE Trans. Comput.*, vol. 64, no. 8, pp. 2394–2407, Aug. 2015.

[37] X. Tang, K. Li, R. Li, and B. Veeravalli, "Reliability-aware scheduling strategy for heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 70, pp. 941–952, Sep. 2010.

[38] A. Benoit, M. Hakem, and Y. Robert, "Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems," *Parallel Comput.*, vol. 35, pp. 83–108, Feb. 2009.

[39] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel, "Generation of fault-tolerant static scheduling for real-time distributed embedded systems with multi-point links," in *Proc. Int. Parallel Distrib. Process. Symp.*, Apr. 2001, Art. no. 125.

[40] G. Manimaran and C. S. R. Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 11, pp. 1137–1152, Nov. 1998.

[41] X. Qin and H. Jiang, "A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems," *Parallel Comput.*, vol. 32, pp. 331–356, Jun. 2006.

[42] Q. Zheng and B. Veeravalli, "On the design of communication-aware fault-tolerant scheduling algorithms for precedence constrained tasks in grid computing systems with dedicated communication devices," *J. Parall. Distrib. Comput.*, vol. 69, pp. 282–294, Mar. 2009.

[43] N. R. Gottumukkala, C. B. Leangsuksun, N. Taerat, R. Nassar, and S. L. Scott, "Reliability-aware resource allocation in HPC systems," in *Proc. Int. Conf. Cluster Comput.*, Sep. 2007, pp. 312–321.

[44] A. Benoit, L. Pottier, and Y. Robert, "Resilient application co-scheduling with processor redistribution," in *Proc. 45th Int. Conf. Parallel Process.*, pp. 123–132, Aug. 2016.

[45] D. Dauwe, E. Jonardi, R. D. Friese, S. Pasricha, A. A. Maciejewski, D. A. Bader, and H. J. Siegel, "HPC node performance and energy modeling with the co-location of applications," *J. Supercomputing*, vol. 72, pp. 4771–4809, Dec. 2016.

[46] NAS parallel benchmarks. [Online]. Available: http://www.nas.nasa.gov/publications/npb.html, Accessed on: Aug.-2016

[47] J. Dongarra, "Report on the Sunway Taihulight System," Tech. Rep. UT-EECS-16–742, Jun. 2016. [Online]. Available: http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf

[48] Top500 Jun. 2017. [Online]. Available: https://www.top500.org/lists/2017/06/, Accessed on: Aug.-2017

[49] J. S. S. T. Association, "DDR4 SDRAM standard," Tech. Rep. JESD79–4B, Jun. 2017. [Online]. Available: https://www.jedec.org/system/files/docs/JESD79-4B.pdf

[50] G. Yang, *Life Cycle Reliability Engineering*. Hoboken, NJ, USA: John Wiley & Sons, 2007.

[51] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *Proc. Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 610–621.

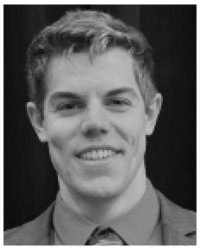
[52] N. R. Tallent, K. J. Barker, D Chavarria-Miranda, A. Tumeo, M. Halappanavar, A. Marquez, D. J. Kerbyson, and A. Hoisie, "Modeling the impact of silicon photonics on graph analytics," in *Proc. Int. Conf. Netw., Archit. Storage*, Aug. 2016, pp. 1–11.

[53] D. Dauwe, "Resource management for extreme scale high performance computing systems in the presence of failures," Ph.D thesis, ECE Dept., Colorado State University, Fort Collins, CO, USA, 2018.

[54] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *Proc. Int. Conf. Cluster Comput.*, Sep. 2004, pp. 93–103.

[55] A. Marowka, "Back to thin-core massively parallel processors," *Comput.*, vol. 44, pp. 49–54, Dec. 2011.

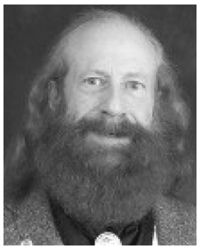
[56] T. Simunic, K. Mihic, and G. De Micheli, "Optimization of reliability and power consumption in systems on a chip," in *Proc. Int. Workshop Integr. Circuit System Des. Power Timing Model. Optim. Simul.*, Sep. 2005, pp. 237–246.



Daniel Dauwe received the BS degree in computer engineering and the BA degree in applied mathematics from the University of Colorado at Colorado Springs. He is currently working toward the PhD degree in electrical engineering with Colorado State University. His research interests include high performance computing resilience and energy efficient scheduling.

Sudeep Pasricha received the BE degree in electronics and communications from the Delhi Institute of Technology, in 2000, and the MS and PhD degrees in computer science from the University of California, Irvine, in 2005 and 2008, respectively. He is currently an associate professor in the Department of Electrical and Computer Engineering and also the Department of Computer Science, Colorado State University. He is a senior member of the IEEE and ACM. A complete vita is available at: http://www.engr.colostate.edu/∼sudeep.

Anthony A. Maciejewski received the BSEE, MS, and the PhD degrees from the Ohio State University, in 1982, 1984, and 1987, respectively. From 1988 to 2001, he was a professor of electrical and computer engineering with Purdue University, West Lafayette. He is currently a professor and department head of electrical and computer engineering with Colorado State University. He is a fellow of the IEEE. A complete vita is available at: http://www.engr.colostate.edu/∼aam.



Howard Jay ("H.J.") Siegel received the BS degree from MIT, and the MSE and PhD degrees from Princeton University. He is a Professor Emeritus with Colorado State University. From 2001 to 2017, he was the Abell Endowed chair distinguished professor of electrical and computer engineering there, where he was also a professor of computer science. From 1976 to 2001, he was a professor at Purdue University. He is a fellow of the IEEE and ACM. A complete vita is available at: http://www.engr.colostate.edu/∼hj.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.