2017 IEEE International Parallel and Distributed Processing Symposium Workshops

An Analysis of Resilience Techniques for Exascale Computing Platforms

Daniel Dauwe*, Sudeep Pasricha*†, Anthony A. Maciejewski* and Howard Jay Siegel*†
*Department of Electrical and Computer Engineering
†Department of Computer Science
Colorado State University, Fort Collins, CO, 80523, USA
Email: ddauwe@rams.colostate.edu, sudeep@colostate.edu, aam@colostate.edu, hj@colostate.edu

***Abstract*—With the increase in the complexity and number of nodes in large-scale high performance computing (HPC) systems, the probability of applications experiencing failures has increased significantly. As the computational demands of applications that execute on HPC systems increase, projections indicate that applications executing on exascale-sized systems are likely to operate with a mean time between failures (MTBF) of as little as a few minutes. A number of strategies for enabling fault resilience in systems of extreme sizes have been proposed in recent years. However, few studies provide performance comparisons for these resilience techniques. This work provides a comparison of four state-of-the-art HPC resilience techniques that are being considered for use in exascale systems. We explore the behavior of each resilience technique under simulated execution of a diverse set of applications varying in communication behavior and memory use. We examine how each resilience technique behaves as application size scales from what is considered large today through to exascale-sized applications. We further study the performance degradation that a large-scale system experiences from the overhead associated with each resilience technique as well as the application computation needed to continue execution when a failure occurs. Using the results from these analyses, we examine how application performance on exascale systems can be improved by allowing the system to select the optimal resilience technique for use in an application-specific manner, depending upon each application's execution characteristics.**

Keywords: exascale resilience; checkpoint restart; multilevel checkpointing; message logging; fault tolerance;

I. Introduction

As the computing power of large-scale computing systems increases exponentially, the failure rates of these systems increase exponentially as well. While 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 up to several times an hour [1]. The resilience techniques implemented in today's high performance computing (HPC) and cloud computing systems are either incapable or impractical when executing at an exascale level [2]. However, several new promising resilience techniques have recently been proposed for next generation computing systems [2] [3] [4]. Unfortunately, little work has been done to assess the performance of these techniques on a common computing environment [5].

This 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 techniques: one contemporary technique and three techniques proposed for use in future systems. Using this common environment, we simulate each resilience technique's performance as greater demands are placed on them with increasing system utilization and test their ability to handle varying levels of system reliability.

We developed a set of synthetic benchmark applications inspired by an analysis of today's scientific benchmark suites operating at scale [6]. The resulting equation-based 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 technique and identify the trade-offs present for different combinations of applications and resilience techniques.

We also analyze an exascale-sized system under a typical 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 have on this environment and the level of benefit that each resilience technique would provide to the system. We conclude by utilizing our analyses of resilience technique trade-offs to demonstrate how system performance in such a failure-prone computing environment can be improved by allowing the system to select the resilience technique likely to provide optimal fault tolerance based on the execution characteristics of each application.

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 large-scale synthetic benchmark applications inspired by current scientific benchmark suites;
- we provide a performance comparison of four state-of-the-art HPC resilience techniques 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

978-0-7695-6149-3/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.41

for HPC resilience and resource management techniques used for scheduling applications;
- we demonstrate the ability to improve system performance in a large-scale failure-prone system by intelligently selecting resilience techniques based on application execution characteristics.

The remainder of this paper is organized as follows. Section II discusses contemporary and proposed resilience techniques, as well as highlighting and describing the background of the four techniques compared in this paper. In Section III we describe the modeling methodology we use for our system simulator. Our implementation of HPC resilience is detailed in Section IV. Sections V, VI, and VII outline our simulated studies and discuss their results. We conclude with a summary of this work in Section VIII.

II. Related Work

A. Overview

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 our efforts on providing a comparison of several checkpoint-based resilience techniques. Our prior work in [7] has been one of the first efforts to provide an analytical comparison of these techniques in large-scale systems. However, that effort was 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 works in [5] and [8]. For the resilience strategies that we consider, our work differs significantly from these high-level surveys by providing simulated comparisons of the performance of each technique.

All checkpointing based techniques rely on the notion of periodically saving the system's executing state and restarting from an earlier error-free state after the occurrence of a system failure [9] [10]. Because recovery from a failure requires these techniques to load a copy of the system state that is not up to date, all checkpointing techniques necessarily lose some productivity because of the necessity of having to recompute work lost between the time of the failure and the time of the last checkpoint. We provide a comparison of four HPC resilience techniques that utilize system checkpointing: checkpoint restart, multilevel checkpointing, message logging, and checkpointing combined with redundancy.

B. Checkpointing and Restarting

Checkpointing is by far the most commonly used resilience technique employed by today's large-scale computing systems. The most general implementation of the checkpointing technique 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 checkpointing technique is referred to as a blocking, coordinated checkpoint scheme [5].

Several variations and improvements on this technique have been made since its initial inception. Attempts have been made to create non-blocking or semi-blocking checkpointing which allows the system to continue to execute while checkpoints are saved to permanent storage [11] [12]. 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 [13].

However, the length of time associated with checkpointing, restarting, and recomputing work lost to a system failure, and the frequency that the system needs to take checkpoints for very large-scale applications when implementing any of these checkpointing techniques, has been shown to provide diminishing returns with increasing system sizes. Traditional checkpointing alone is not expected to be capable of providing resilience to systems at exascale sizes [2].

C. Multilevel Checkpointing

Because different types of failures can affect a computing system by different amounts, not all failures require restarting the system from a checkpoint to the parallel file system [14]. Multilevel checkpointing exploits this by providing the system with several levels of checkpointing. A system employing a multilevel checkpointing scheme may allow for levels that trade-off faster (but able to recover from fewer types of failures) checkpoints to RAM or to a node's local disc and less frequent (but able to recover from more types of failures) checkpoints to the parallel files system, each level offering a trade-off between the time required by the system to checkpoint or restart, and the level of failure severity that the checkpoint can recover from [3]. 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 [3] [15]. Attempts have also been made to reduce checkpointing's dependence on the parallel file system [16] [17]. One challenge associated with using a multilevel checkpointing technique is in determining the optimal number of checkpointing levels to provide to the system, and the optimal computation intervals between checkpoints at each level. Various solutions to this problem have been proposed [3] [18] [19].

D. 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 [20]. 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 [21]. Using message logging as a technique for resilience has the benefit of potentially saving computation time, because the recovering node does not need to wait for the re-computation on 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 progress of the failed node has recovered [22].

E. Redundancy

Redundancy improves a system's reliability by executing redundant copies of the same piece of code [23]. It is possible to implement redundancy in either hardware or software [8], but in either case the improved reliability of the system comes at the cost of using additional resources.

Recent attempts have been made to 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 [24]. 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 [4].

III. Exascale Modeling Methodology

A. 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 [7] [25] [26]. 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*: the simulated time at which an application arrives to the system,
- *mapping*: the process by which the resource management heuristic assigns an application to system nodes,
- *computation*: execution toward application completion,
- *failures*: the simulated 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.

Checkpoints, restarts, and recovery are all resilience-technique specific events that determine how an application behaves in a system with failures. Each of these events affect applications differently based on the type of resilience technique employed by the application, the application's execution characteristics, and the characteristics of the failures that occur in the system. This is discussed in detail in Section IV. 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 technique being used. In particular, while failure events have a large impact on the behavior of the resilience-technique related events, failure events themselves are a function of the reliability and size of the nodes of the system, and are not affected by the resilience technique employed by the system.

B. Modeling Extreme Scale Applications

To ensure our simulated environment has access to a diverse range of applications that will behave similarly to future 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 [27] performed in [6]. 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 would all become heavily communication bound at large system sizes. The analysis performed by the authors for the BT application indicates that at extreme scales communication began to dominate between 22%, 50%, and 80% of the application's execution time depending on which of the three input parameter sets was used for the application's execution.

Similar to the BT application, our synthetic benchmarks are defined as a discrete set of time steps, 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. Time steps are defined so that both T_W and T_C take values between zero and one and $T_W + T_C = 1$ minute, thus allowing application execution times to be of arbitrary length (equal to the number of time steps) and unaffected by the application's size. For all simulated studies performed here, applications have between 360 and 2880 time steps giving every executing application an execution time between six hours and two days 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 seen in [6], we have defined our synthetic benchmarks to have four levels of communication ranging from $T_C = 0$ (representing an EP type of application with little to no communication) to T_C values of 0.25, 0.5, and 0.75 representing similar levels of communication dominance of application execution time seen in the analysis of BT. We also allow for each of the four 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 eight application types with a diverse range of communication and memory characteristics. Each of the eight application types are defined in Table I.

TABLE I
CHARACTERISTICS OF APPLICATION TYPES

	memory per node	
communication intensity	32 GB	64 GB
0% ($T_C = 0.0$)	A_{32}	A_{64}
25% ($T_C = 0.25$)	B_{32}	B_{64}
50% ($T_C = 0.5$)	C_{32}	C_{64}
75% ($T_C = 0.75$)	D_{32}	D_{64}

We assume 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 V, VI, and VII.

C. Simulated System Setup

The simulated exascale system is a homogeneous system inspired by the architecture used to develop China's Sunway TaihuLight supercomputer, recently determined to be the world's highest performing system as of November 2016 [28]. 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 for a total of 65 cores in each core cluster. The four core clusters in a system 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 increase by a factor of four by the time an exascale machine is developed allowing for a total of 1028 cores per node providing approximately 12 TFLOPs of compute power for each system node. A system composed of 120,000 of these high performing 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 128GB per system node. In addition to an increase in volume, we also assume that future memory is likely to utilize newer architectures, such as the hybrid memory cube specified in 2014 [29], allowing for increased aggregate Memory Bandwidth, B_M, of up to 320 GB/s.

For the simulated studies in Sections VI and VII, the system also assumes each application arriving to the system has individual *deadlines*. Applications removed from the system because they could not meet their execution deadlines are referred to as *dropped applications* and the percentage of total applications that are dropped is the performance metric that we use for the simulated studies of Sections VI and VII. 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 uniformly randomly selected value $\mathcal{U}$ between 1.2 and 2 giving the application a deadline of

$$T_D = T_A + \mathcal{U}(1.2, 2.0) * T_B \ . \tag{1}$$

D. System Resource Management Techniques

We explore the behavior of three techniques for resource management operating in a system with failures and resilience. Each 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 executing in the system 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 scheduled during a future mapping event.

1) FCFS Technique: First come first served (FCFS) is the most commonly employed resource management technique in HPC systems and is therefore 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 in the set of unmapped applications to begin execution. Applications that are not assigned to nodes are scheduled in a future mapping event.

2) Random Technique: The *random* resource scheduling technique randomly selects an application from the set of mappable applications and assigns it to execute on the first 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. This process is repeated until the set of mappable applications is empty.

3) 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 baseline execution time and deadline. The set of unmapped applications is ordered into a priority queue 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. All such applications are "dropped" from the system. After clearing out applications with negative slack, the slack-based resource management technique schedules applications to nodes in the system in the order of applications with the lowest slack. Applications that cannot begin execution immediately are returned to the set of unmapped applications. The slack-based schedule continues evaluating applications until the queue is empty and 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.

E. Modeling System Failures

We assume that failures can be characterized by three attributes: the time of the failure's occurrence, the location of the failure, and the severity 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 [30]. Every failure occurs according to the previous failure's arrival 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 arrival 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 not idle, N_s, 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 level of failure severity corresponds to the type of failure that has occurred in the system. This attribute is used by multilevel checkpointing resilience techniques to determine optimal intervals between checkpoints and the level of checkpoint needed to recover from a particular type of failure. The specific mapping of types of failures to levels of failure severities is defined by the implementation of the multilevel checkpointing technique. This work assumes the implementation described in [3]. The probability of experiencing a failure at a failure severity of level j is determined according to the ratio of the number of failures that occur at each failure severity level, λ_{L_j}, to the total number of failures, λ_{L_t}, measured for an extended interval of time. The resulting discrete set of ratios for each level is used to create a probability mass function from which random variates are sampled to define the severity attribute of each failure. We use the values in [3] determined by the study of failure logs of the BlueGene/L system to define λ_{L_j} and λ_{L_t}.

F. Communication Model

System communication plays a large role in the behavior and performance of every resilience technique we examine. We account for communication in the system and model its effects on application simulation. We assume that future exascale systems are likely to have improved communication over today's systems, and base the communication model for the studies performed here on the "NDR InfiniBand" network described in [31]. 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 specifically for each resilience technique in Section IV.

TABLE II
RESILIENCE TECHNIQUE PARAMETERS

parameter	use in modeling
T_S	application length (time steps)
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 application
L	network latency
B_N	communication bandwidth
N_S	number of network switch connections
λ_a	application failure rate
M_n	system component MTBF
τ	optimal checkpoint period
$T_{C_{PFS}}$	time required to checkpoint to a PFS
$T_{C_{L1}}$	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

IV. RESILIENCE TECHNIQUE SIMULATION

A. Overview

Four resilience techniques have been implemented in our simulator. A traditional checkpoint restart based technique, *Checkpoint Restart*, as well as three techniques proposed for next-generation computing systems: a multilevel checkpointing approach described in [3], *Multilevel Checkpoint*, an implementation of message logging outlined in [2], *Parallel Recovery*, and a technique combining traditional checkpointing with partial or full redundancy of the executing application from [4], *Redundancy*. The following subsections present details of how each resilience technique was modeled, with all relevant parameters summarized in Table II.

B. Checkpoint Restart

Our implementation of the Checkpoint Restart resilience technique 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 technique 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 technique requires to read and write its checkpoint data to a parallel file system, $T_{C_{PFS}}$, is dependent on application size, N_a, memory use, and system parameters for communication to give

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

Parameters for the applications and environment in this study impose a checkpoint and restart time of between 17-35 minutes depending on the application type. It is generally assumed that, because the rate of growth of system memory and I/O bandwidth remains similar, even as large-scale system performance continues to improve, the time necessary for the system to checkpoint and restart an application utilizing the entire system using a parallel file system remains constant, between 20-30 minutes [1].

The optimal checkpoint period is dependent on the applications's checkpoint time and failure rate. The value for each application's failure rate is dependent on application size and given by $\lambda_a = \frac{N_a}{M_n}$. The resulting equation for the optimal checkpoint period, τ, is derived according to [32] as

$$\tau = \sqrt{\frac{2T_{C_{PFS}}}{\lambda_a}} - T_{C_{PFS}} \ . \tag{4}$$

C. Multilevel Checkpointing

The Multilevel Checkpointing approach from [3] 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 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 application's required memory in the system divided by the node's memory transfer rate

$$T_{C_{L1}} = \frac{N_M}{B_M} \ . \tag{5}$$

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 time for a level two checkpoint is equal to the time required to send the data to the partner node plus the time required to write the data to memory

$$T_{C_{L2}} = 2(T_{C_{L1}} + L + \frac{N_M}{B_M}) \ . \tag{6}$$

The equation is multiplied by two to account for both the time required to checkpoint data from a given node to its partner as well as the time required to checkpoint the partner's data back to the given node.

The third level checkpoint is written to a parallel file system, and the time required is the same as presented in Equation 3. Here we assume checkpoint and restart times are symmetric. Failure severity and optimal checkpoint intervals at each level are determined based on the Markov model in [3].

D. Parallel Recovery

The Parallel Recovery technique in [2] is an improvement to the message logging resilience technique. 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 techniques, 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 techniques, parallel recovery improves checkpointing and restart time by utilizing the in-memory checkpointing technique outlined in [33]. The in-memory checkpointing technique behaves similarly to the level two checkpoint to a partner node described in Section IV-C. We therefore used Equation 6 to represent the time required for an in-memory checkpoint or restart.

Utilization of the parallel recovery technique 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 [2]. The increase in execution time from message logging increases the application baseline execution time when using parallel recovery to

$$T_B = \mu T_S(T_W + T_C) \ . \tag{7}$$

The optimal checkpoint period when using parallel recovery is given by Equation 4. The remainder of the parallel recovery specific parameter values are taken directly from [2].

E. Partial Redundancy

The Partial Redundacy technique in [4] combines the traditional checkpointing technique 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 nodes performing the same computation. At the same time, checkpoints are taken by the system at regular intervals. When failures occur on nodes in the system, the system only requires a restart if failures occur on all (possibly redundant) physical nodes associated with one of the application's virtual nodes before the next system checkpoint.

Apart from the application baseline execution time, all parameters associated with the partial redundancy resilience technique remain the same as the Checkpoint Restart technique. To account for the increase in application execution time from the higher communication associated with redundancy's necessity for duplicated communication, the communication term in the equation for baseline execution time is modified to be scaled by the degree of system redundancy, r, which results in a baseline execution time when utilizing redundancy of

$$T_B = T_S(T_W + rT_C) \ . \tag{8}$$

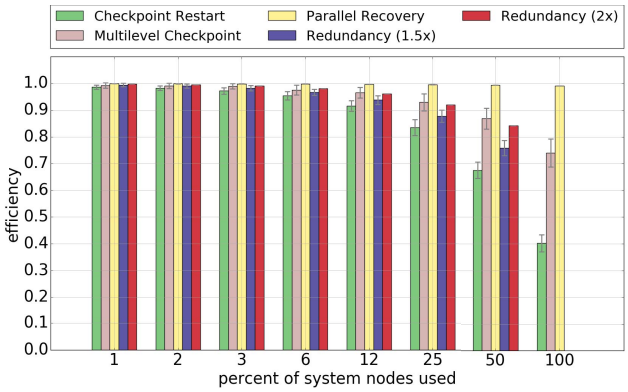

Fig. 1. Resilience technique efficiency at increasing percentages of total system use by the low memory use and low communication application defined in Table I as A_{32}. Efficiency is defined to be the ratio of an application's time without slowdowns (from failures or checkpointing) over the application's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

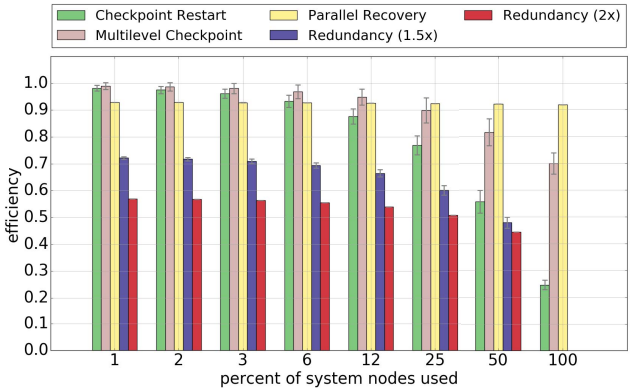

Fig. 2. Resilience technique efficiency at increasing percentages of total system use by the high memory use and high communication application defined in Table I as D_{64}. Efficiency is defined to be the ratio of an application's time without slowdowns (from failures or checkpointing) over the application's execution time with slowdowns (from failures or checkpointing). Processors in the system experience a ten year MTBF. Each bar in the figure represents the average of 200 trials. Standard deviations are shown for each bar.

V. Resilience Technique Performance with Application Scaling

We utilized our simulation environment to conduct several sets of experiments examining the performance of each resilience technique. We evaluated the performance of each of the eight application types defined in Table I as each application type is 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, or one day of execution.

Figures 1 and 2 highlight the results from the first set of experiments that analyze 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 technique overhead delays such as checkpointing. Although we show only two of the eight application types described in Table I, trends similar to these can be seen for all application types.

The experiments assume a mean time between failures of ten years based on the values used in each of the works on which the resilience techniques we consider are modeled. Figure 1 depicts the efficiency of an application that exhibits low memory requirements and a low amount of communication between nodes (indicated by application A_{32} in Table I) as the size of the application increases. This is indicated on the x-axis of the figure as an increase in the percentage of system nodes occupied by the application. For applications exhibiting these characteristics, the parallel recovery technique is the most efficient for all application sizes. The dominance of the parallel recovery technique for all application sizes is true for both of the low communication applications. The figure also demonstrates the rate at which each resilience technique varies in performance. As the application occupies larger portions of the system, the decrease in the traditional checkpointing technique efficiency drops the fastest, followed by both forms of redundancy, and then multilevel checkpointing. The parallel recovery technique is the best at maintaining its efficiency as the application size increases, but it still decreases in efficiency at larger application sizes. While results for the redundancy techniques provide better efficiency than the traditional Checkpoint Restart technique, they 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 technique.

Figure 2 provides efficiency results for a high communication high memory use application (indicated by application D_{64} in Table I). The same general trends of resilience technique efficiency decreasing with increasing application size seen in Figure 1 are also observed in Figure 2. However, both the parallel recovery technique and the two redundancy techniques suffer a larger decrease in efficiency for all application sizes than the checkpoint restart or multilevel checkpointing techniques suffer. This decrease is due to the parallel recovery and redundancy technique's higher reliance on communication, and it results in a distinct trade-off between which resilience technique is the most efficient for a given application size. In Figure 2, a shift in the optimal resilience technique from multilevel checkpointing to parallel recovery occurs when applications require 25% or more of the system.

However, when analyzing the performance of each resilience technique type when executing application D_{64} as compared to the same resilience technique type executing application A_{32}, both the parallel recovery technique and the two redundancy techniques suffer a larger decrease in efficiency for all application sizes than the checkpoint restart or multilevel checkpointing techniques suffer for the same change in application execution characteristics.

As systems trend towards manycore architectures, with hundreds or thousands of CPU cores on a single socket,

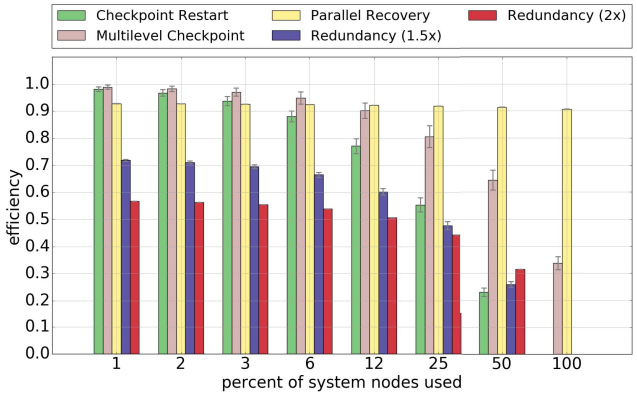

Fig. 3. Resilience technique efficiency at increasing percentages of total system use by the high memory use and high communication application defined in Table I as D_{64}. Efficiency is defined to be the ratio of an application's time without slowdowns (from failures or checkpointing) over the application's execution time with slowdowns (from failures or 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.

component failure rates are likely to increase [34] [35]. Our simulated exascale system assumes an approximate factor of four 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. We examine the sensitivity of our results to system component reliability by decreasing the mean time between failures of processor components to $M_n = 2.5$ years. Most of the same trends from Figures 1 and 2, including the trade-off in resilience technique optimality for applications with communication, also exist when system components are assumed to be less reliable. The results in Figure 3 show the performance for a high memory, high communication application (indicated by application D_{64} in Table I) executing at increasing percentages of system sizes when processor nodes have an MTBF of 2.5 years. The parallel recovery resilience technique is still the best at maintaining execution efficiency as application size increases. Predictably, the data also indicates that with an increased failure rate each resilience technique decreases in efficiency at a faster rate. Traditional Checkpoint Restart is particularly affected by a rapid decrease in efficiency, with it spending so much time creating and restoring from checkpoints that applications are unable to even complete execution at exascale sizes.

VI. Resilience Technique 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 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 technique.

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 given some constraint. 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 V. 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 III-C.

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 two hours until a total of 100 applications have arrived to the system. Each application that arrives to the system is uniformly randomly selected from the set of eight application types discussed in Table I. Baseline execution times for each arriving application are uniformly randomly selected to be either six, twelve, twenty-four, or fourty-eight hours in length. The number of system nodes required by each arriving application is uniformly randomly selected to use between 10 to 500 petaflops by utilizing approximately one, two, three, six, twelve, twenty-five, or fifty percent of the exascale system. Exascale sized applications are not considered in this study. The processor MTBF for these studies is ten years.

Each set of 100 applications that arrives to the simulated system is referred to as an *arrival pattern*. Fifty such arrival patterns were created. The behavior of each resilience technique and resource management technique was examined using the same set of arrival patterns allowing each of their performances to be compared using the same sets of arriving applications.

Given that the results from Section V indicate that redundancy-based resilience techniques will be unlikely to be implemented in an exascale system, we limit our discussion of a practical use-case scenario to only the Checkpoint Restart, Multilevel Checkpointing, and Parallel Recovery techniques. We compare the results from each of these techniques by averaging the 50 arrival patterns for each experiment and comparing those values to an *Ideal Baseline* execution that is an average of each arrival pattern that executes without delays from failures or delays associated with overhead from resilience techniques.

Results comparing the performance of each resilience technique and resource management technique combination are shown in Figure 4 by indicating the number of dropped applications averaged from each of the 50 arrival patterns. In comparison to the performance of the Ideal Baseline, the results from these simulated studies shows how the presence of failures and overhead from resilience techniques negatively impacts system performance by increasing the percentage of

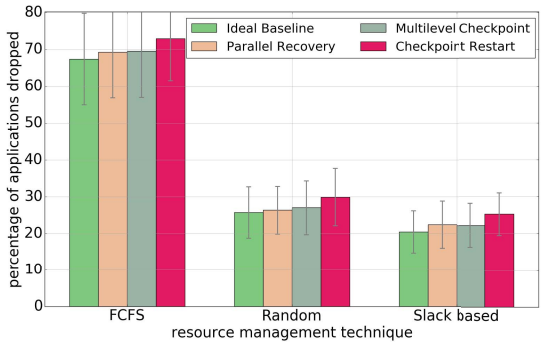

Fig. 4. Percentage of applications dropped from the system for each resilience technique and resource management technique combination. Bars in the figure represent the average of 50 arrival patterns. Standard deviations are shown for each bar.

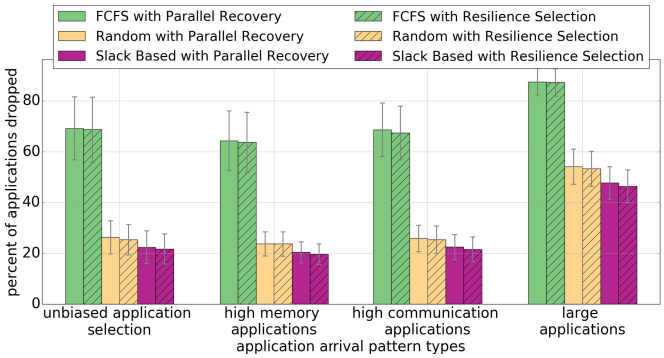

Fig. 5. Percentage of applications dropped from the system for each resource management technique for the Parallel Recovery resilience technique, and each resource management technique with Resilience Selection. Groupings of bars show four different types of application arrival patterns. Bars in the figure represent the average of 50 arrival patterns. Standard deviations are shown for each bar.

dropped applications in the system. While system performance is negatively impacted regardless of which resilience technique is utilized, the results also imply that the optimal resilience technique varies among resource management techniques.

VII. Resilience-Aware Resource Management

The implications from the results of our simulated studies in Section V and Section VI indicate that there is a potential for improving system performance. In addition to deciding when and on what nodes an application will execute, the system resource manager will also be given the opportunity to intelligently select the resilience technique that is most likely to provide the best performance for each application based on the results from Section V. Applications that are provided with this *Resilience Selection* will be able to use the best resilience technique possible for their execution, thereby improving performance of the system as a whole.

Simulated studies exploring the use of Resilience Selection have a similar setup to the simulated studies of Section VI. However, in addition to the application arrival patterns seen in Section VI that allow 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 $N_m = 64$GB;
- high communication applications having communication values of $T_C > 0.25$;
- large applications that occupy twelve, twenty-five, or fifty percent of the exascale system.

These application arrival pattern types were chosen because they are likely to be more challenging for a system to schedule and execute. The results of this study are shown in Figure 5. Results are shown for each resource management technique from Section III-D when utilizing the Parallel Recovery resilience technique (indicated by each of the bars without hash marks in the figure) because it is most consistently the best performing resilience technique. Each resource management technique utilizing Parallel Recovery is also compared to the same execution of arrival patterns when Resilience Selection is used for each application type (indicated by each of the hashed bars in the figure). The results of these simulated studies indicate that Resilience Selection provides a benefit (albeit small) to the system in all but one circumstance. Resilience Selection is able to offer the most improvement for the high communication application arrival patterns because these types of applications offer the greatest variability between which resilience technique is optimal. Unsurprisingly, arrival patterns biased toward large applications perform worse than the other arrival pattern types because they require more system resources. But the large application arrival patterns still benefit from Resilience Selection about the same amount as the unbiased application selection.

Because the Parallel Recovery technique never requires checkpoints to a parallel file system, it has an advantage over the other resilience techniques when providing fault tolerance to applications requiring more memory. This is evident from the high memory application arrival patterns in Figure 5. High memory application arrival patterns not only generally perform better than the other arrival pattern types, but they also gain the least benefit from Resilience Selection as high memory applications are less likely to have techniques other than Parallel Recovery giving them the best performance. Consequently, for applications with high memory use there are fewer opportunities for Resilience Selection to outperform Parallel Recovery.

VIII. Conclusions

HPC resilience has become an increasingly important topic as we approach exascale system sizes. It has also become increasingly important that resilience strategies that are proposed for use in these systems are analyzed in a common computing environment. This work provides one of the few attempts to test a variety of new HPC resilience techniques 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 techniques for HPC resilience, the traditionally employed Checkpoint Restart technique, as well as, Multilevel Checkpointing,

Parallel Recovery, and Partial Redundancy, three techniques proposed for next generation large-scale systems. While the Parallel Recovery resilience technique is generally the most efficient, our analyses indicate that each resilience technique 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 also study the effects that HPC resilience and system failures have on resource management. Our results indicate that while Parallel Recovery is still likely to be the best performing resilience technique, for most of the resource management techniques there is still a significant decrease in system performance due to failures and overhead from resilience techniques. However, we also show that the system performance can be slightly improved if resilience technique selection is considered for each application executing in the system.

ACKNOWLEDGMENTS

The authors thank Dylan Machovec and Ninad Hogade for their 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 National Science Foundation (NSF) under grant number CNS-0923386. The authors thank Hewlett Packard (HP) of Fort Collins for providing us some of the machines used for testing.

REFERENCES

[1] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *Int'l Journal of HPC Applications*, vol. 23, no. 3, pp. 212–226, 2009.

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

[3] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Int'l Conf. for HPC, Networking, Storage and Analysis*, 11 pp., 2010.

[4] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for HPC," in *Int'l Conf. on Dist. Computing Systems*, pp. 615–626, 2012.

[5] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 5–28, 2014.

[6] R. F. Van der Wijngaart, S. Sridharan, and V. W. Lee, "Extending the BT NAS parallel benchmark to exascale computing," in *Int'l Conf. on HPC, Networking, Storage and Analysis*, pp. 94:1–94:9, 2012.

[7] 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 *The 6th IEEE Int'l Symp. on Cloud and Service Computing*, pp. 436–443, 2016.

[8] 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," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[9] D. P. Jasper, "A discussion of checkpoint restart," *Software Age*, vol. 3, no. 10, pp. 9–14, 1969.

[10] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[11] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI," in *Conf. on Supercomputing*, pp. 22, 2006.

[12] X. Ni, E. Meneses, and L. V. Kalé, "Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm," in *Int'l Conf. on Cluster Computing*, pp. 364–372, 2012.

[13] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic MPI applications," in *Int'l Par. Dist. Proc. Symp.*, pp. 989–1000, 2011.

[14] N. H. Vaidya, "A case for two-level distributed recovery schemes," *SIGMETRICS*, vol. 23, no. 1, pp. 64–73, 1995.

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

[16] K. Mohror, A. Moody, G. Bronevetsky, and B. de Supinski, "Detailed modeling and evaluation of a scalable multilevel checkpointing cystem," *IEEE Trans. Par. and Dist. Systems*, vol. 25, no. 9, pp. 2255–2263, 2014.

[17] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka, "Distributed diskless checkpoint for large scale systems," in *Int'l Conf. on Cluster, Cloud and Grid Computing*, pp. 63–72, 2010.

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

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

[20] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.

[21] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using asynchronous message logging and checkpointing," in *Symp. on Principles of Dist. Computing*, pp. 171–181, 1988.

[22] "Energy profile of rollback-recovery strategies in high performance computing," *Parallel Computing*, vol. 40, no. 9, pp. 536–547, 2014.

[23] J. F. Wakerly, "Microcomputer reliability improvement using triple-modular redundancy," *Proceedings of the IEEE*, vol. 64, no. 6, pp. 889–895, 1976.

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

[25] 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 Computing: Informatics and Systems*, vol. 5, pp. 14–30, 2015.

[26] 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," *The Journal of Supercomputing*, vol. 72, no. 12, pp. 4771–4809, 2016.

[27] "NAS parallel benchmarks," accessed Aug. 2016.

[28] "Top500 Nov 2016," accessed Jan. 2017.

[29] H. M. C. Consortium, "Hybrid Memory Cube Specification 2.1," Tech. Rep. HMC-30G-VSR PHY, 132 pp., 2014.

[30] G. Yang, *Life Cycle Reliability Engineering*. Hoboken, NJ: John Wiley & Sons, second ed., 2007.

[31] 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 *2016 IEEE Int'l Conf. on Networking, Architecture and Storage (NAS)*, 11 pp., 2016.

[32] 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.

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

[34] A. Marowka, "Back to thin-core massively parallel processors," *Computer*, vol. 44, no. 12, pp. 49–54, 2011.

[35] T. Simunic, K. Mihic, and G. De Micheli, "Optimization of reliability and power consumption in systems on a chip," in *15th Int'l Workshop on Integrated Circuit and System Design Power and Timing Modeling, Optimization and Simulation, in PATMOS 2005 Proceedings*, pp. 237–246, 2005.