Toward Exascale Resilience: 2014 Update

Franck Cappello,[1,2] ***Al Geist,***[3] ***William Gropp,***[2] ***Sanjay Kale,***[2] ***Bill Kramer,***[2] ***Marc Snir***[1,2]

Resilience is a major roadblock for HPC executions on future exascale systems. These systems will typically gather millions of CPU cores running up to a billion threads. Projections from current large systems and technology evolution predict errors will happen in exascale systems many times per day. These errors will propagate and generate various kinds of malfunctions, from simple process crashes to result corruptions.

The past five years have seen extraordinary technical progress in many domains related to exascale resilience. Several technical options, initially considered inapplicable or unrealistic in the HPC context, have demonstrated surprising successes. Despite this progress, the exascale resilience problem is not solved, and the community is still facing the difficult challenge of ensuring that exascale applications complete and generate correct results while running on unstable systems. Since 2009, many workshops, studies, and reports have improved the definition of the resilience problem and provided refined recommendations. Some projections made during the previous decades and some priorities established from these projections need to be revised. This paper surveys what the community has learned in the past five years and summarizes the research problems still considered critical by the HPC community.

Keywords: Exascale, Resilience, Fault-tolerance techniques.

1. Introduction

We first briefly introduce the terminology that is used in this paper, following the taxonomy of Aviženis and others [3, 90]. System *faults* are causes of *errors*, which manifest themselves as incorrect system states. Errors may lead to *failures*, where the system provides an incorrect service (e.g., crashes or provides wrong answers). We deal with faults by predicting, preventing, removing, or tolerating them. *Fault tolerance* is achieved by detecting errors and notifying about errors and by recovering, or compensating for errors, for example, by using redundancy. *Error recovery* includes *rollback*, where the system is returned to a previous correct state (e.g., a checkpoint) and *rollforward*, where a new correct state is created. Faults can occur at all levels of the stack: facilities, hardware, system/runtime software, and application software. Fault tolerance similarly can involve a combination of hardware, system, and application software.

This paper deals with *resilience* for exascale platforms: the techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults. The lack of appropriate resilience solutions is expected to be a major problem at exascale: We discuss in Section 2 the many reasons errors are likely to be much more frequent in exascale systems. The current solutions used on petascale platforms, discussed in Section 3, may not scale up to exascale. We risk having systems that can perform quintillions of operations each second but never stay up long enough to progress in their computations, or produce results that cannot be trusted.

The problem of designing reliable computers out of unreliable components is as old as computing—as old as Babbage's analytical engine [15]. Frequent failures were a major problem in the earliest computers: ENIAC had an mean time to failure of two days [93]. Major advances in this area occurred in the 1950s and 1960s, for example, in the context of digital telephone

[1]Argonne National Laboratory
[2]University of Illinois at Urbana Champaign
[3]Oak Ridge National Laboratory

switches [35] and mainframes [91]. More recently, NASA examined the use of COTS (non-rad-hardened) processors for space missions, which requires tolerance of hardware errors [70]. Resilience for HPC is a harder problem, however, since it involves large systems performing tightly coupled computations: an error at one node can propagate to all the other nodes in microseconds.

Five years ago we published a survey of the state of the art on resilience for exascale platforms [19]. Since then, extraordinary technical progress has been made in many domains related to exascale resilience. Several technical options, initially considered inapplicable or unrealistic in the HPC context, have demonstrated surprising success. Despite this progress, the exascale resilience problem is not solved, and the community still faces the difficult challenge of ensuring that exascale applications complete and generate correct results while running on unstable systems. Since 2009, many workshops, studies, and reports have improved the definition of the resilience problem and provided refined recommendations [18, 19, 32, 38, 39, 58, 69, 90]. Some projections made during the previous decades and some priorities established from these projections now need to be revised. This paper surveys what the community has learned in the past five years (Section 4) and summarizes the research problems still considered critical by the HPC community (Section 5).

2. The Exascale Resilience Problem

Future exascale systems are expected to exhibit much higher fault rates than current systems do, for various reasons relating to both hardware and software.

2.1. Hardware Failures

Hardware faults are expected to be more frequent: since clock rates are not increasing, performance increases require a commensurate increase in the number of components. With everything else being equal, a system 1,000 times more powerful will have at least 1,000 times more components and will fail 1,000 times more frequently.

However, everything else is not equal: smaller transistors are more error prone. One major cause for transient hardware errors is cosmic radiation. High-energy neutrons occasionally interact with the silicon die, creating a secondary cascade of charged particles. These can create current pulses that change values stored in DRAM or values produced by combinatorial logic. Smaller circuits are more easily upset because they carry smaller charges. Furthermore, multiple upsets become more likely. Smaller feature sizes also result in larger manufacturing variances, hence larger variances in the properties of transistors, which can result in occasional incorrect or inconsistent behavior. Smaller transistors and wires will also age more rapidly and more unevenly so that permanent failures will become more frequent. Energy consumption is another major bottleneck for exascale. Subthreshold logic significantly reduces current leakage but also increases the probability of faults.

Vendors can compensate for the increase in fault rates with various techniques. For example, for regular memory arrays, one can use more powerful error correction codes and interleave coding blocks in order to reduce the likelihood of multiple bit errors in the same block. Buses are usually protected by using parity codes for error detection and by retries; it is relatively easy to use more powerful codes. Logic units that transform values can be protected by adding

redundancy in the circuits. Researchers have estimated that an increase in the frequency of errors can be avoided at the expense of 20% more circuits and more energy consumption [90].

Whether such solutions will be pursued is unclear, however: the IC market is driven by mobile devices that are cost and energy sensitive and do not require high reliability levels. Most cloud applications are also cost sensitive but can tolerate higher error rates for individual components. The small market of high-end servers that require high reliability can be served by more costly solutions such as duplication or triplication of the transactions. This market is not growing in size or in the size of the systems used. Thus, if exascale systems will be built out of commodity components aimed at large markets, they are likely to have more frequent hardware errors that are not masked or not detected by hardware or software.

2.2. Software Failures

As hardware becomes more complex (heterogeneous cores, deep memory hierarchies, complex topologies, etc.), system software will become more complex and hence more error-prone. Failure and energy management also add complexity. Similarly, the increase use of open source layers means less coordinated design in software, which will increase the potential for software errors. In addition, the larger scale will add complexities as more services need to be decentralized, and complex failure modes that are rare and ignored today will become more prevalent.

Application codes are also becoming more complex. Multiphysics and multiscale codes couple an increasingly large number of distinct modules. Data assimilation, simulation, and analysis are coupled into increasingly complex workflows. Furthermore, the need to reduce communication, allow asynchrony, and tolerate failures results in more complex algorithms. Like system software, these more complex algorithms and application codes are more error-prone.

Researchers have predicted that large parallel jobs may fail as frequently as once every 30 minutes on exascale platforms [90]. Such failure rates will require new error-handling techniques. Furthermore, silent hardware errors may occur, requiring new error-detection techniques in (system and/or application) software.

3. Lessons learned from Petascale

Current petascale systems have multiple component failures each day. For example, a study of the Blue Waters system [1] during its 2013 preproduction period showed that, across all categories of events, an event that required remedial repair action occurred on average every 4.2 hours and that systemwide events occurred approximately every 160 hours [34]. The reported rates included events that failed transparently to applications or were proactively detected before application use but required remedial actions. Events with performance inconsistency and long failover times were reported as errors even if applications and the failover operation eventually completed successfully. Hence the events that were actually distuptive to applications were about half as frequent. In the first year of Blue Waters' full production, the rates improved by approximately a factor of 2. Similar rates are reported on other systems. A significant portion of failures are due to software—in particular, file and resource management systems. Furthermore, software errors take longer than hardware errors to recover from and account for the majority of downtime.

Software errors could be avoided by more rigorous testing. Testing HPC software at scale is hard and expensive, however. Since very large systems are one of a kind, each with a unique

configuration, they are expensive and require unique machine rooms. Usually, vendors are not able to deploy and test a full system ahead of installation at the customer site, and customers cannot afford long testing periods ahead of actual use. Some scalability bugs will occur only intermittently at full scale. Subtle, complex interactions of many components may take a long time to occur. Many of the software products deployed on a large platform are produced by small companies or by teams of researchers that have limited resources for thorough testing on multiple large platforms.

The standard error-handling method on current platforms is periodic application checkpoint. If a job fails, it is restarted from its last checkpoint. Checkpoint and restart logic is part of the application code. A user checkpoint can be much smaller than a system checkpoint that would save all the application's memory; the checkpoint information can be also used as the output from a simulation; and a user checkpoint can be used to continue a computation on another system.

For single-level checkpointing, the checkpoint interval can be computed by using the formula developed by Young [97] or Daly [29]. Young's formula is particularly simple: $Interval = \sqrt{2 \times MTTI \times checkpt}$, where $MTTI$ is the mean time to interrupt and $checkpt$ is the checkpoint time. This formula approximates the optimum checkpoint interval, assuming a memoryless model for failures. For large jobs, this typically implies a checkpoint about every hour or multiple times per hour.

Root cause analysis of failures, especially software failures, is hard. Error logs may report which component signaled a failure, but failures can be due to a complex chain of events. For illustration, consider the actual case of a system crash due to a fan failure: The failure of one fan caused multiple hardware components to stop working; the cascade of errors reported from all these components overwhelmed the monitoring network and crashed the monitoring system; this crash, in turn, caused the entire system to crash [57]. The volume of system activity and event processing for large-scale systems is on the order of several tens of gigabytes and hundreds of millions of events per day during normal activity.

Current systems do not have an integrated approach to fault tolerance: the different subsystems (hardware, parallel environment software, parallel file system) have their own mechanisms for error detection, notification, recovery, and logging. Also, current systems do not have good error isolation mechanisms. For example, the failure of any component in a parallel job results in the failure of the entire job; the failure of the hardware monitoring infrastructure may impact the entire machine.

We note that all reports on failure rates focus on hardware and system software and ignore application software. From the viewpoint of the supercomputing center, a failure due to a bug in the application code is not a failure. Such failures are not reported, and no statistics of their frequency exist. Also, supercomputing centers have no information on errors that were not detected by hardware or system software. An incorrect result may be detected by the user once the run is complete, but it will be practically impossible to correctly attribute such an error to a root cause. Anecdotal evidence suggests that undetected hardware errors do occur on current systems [57].

4. Progress toward Exascale Resilience

We present in this section the most significant research progress in resilience since 2009.

4.1. System Software Approaches

We first discuss the progress in handling fail-top errors by checkpointing. We then describe other approaches, including forward recovery, replication, failure prediction, and silent data corruption mitigation.

4.1.1. Checkpointing

To tolerate fail-stop errors (node, OS or process crash, network disconnections, etc.), all current production-quality technologies rely on the classic rollback recovery approach using checkpoint restart (application-level or system-level). Solutions focus on applications using MPI and Charm++. The most popular approach is application-level checkpointing, where the programmer defines the state that needs to be stored in order to guarantee a correct recovery in case of failure. The programmer adds some specific functions in the application to save essential state and restore from this state in case of failure. One of the drawbacks of checkpointing in general, and of application-level checkpointing in particular, is the nonoptimality of checkpoint intervals. Another drawback is the burden placed on the I/O infrastructure, since the checkpoint I/O may actually interfere with communication and I/O of other applications. The impact of suboptimal checkpoint intervals is investigated in [68]. System-level checkpoint can also be used in production, with technologies such as BLCR [63]. Some recent research in this domain focuses on the integration of incremental checkpointing in BLCR. In the past few years, research teams have developed mechanisms to checkpoint accelerators [81, 83].

The norm in 2009 was to store the application state on remote storage, generally a parallel file system, through I/O nodes. Checkpoint time was significant (often 15–30 minutes), because of the limited bandwidth of the parallel file system. When checkpoint time is close to the MTBF, the system spends all its time checkpointing and restarting, with little forward progress. Since the MTJI may be an hour or less on exascale platforms, new techniques are needed in order to reduce checkpoint time.

One way of achieving such a reduction is to reduce the checkpoint size. Techniques such as memory exclusion, data compression, and compiler analysis to detect dead variables have been proposed. More recently some researchers have explored hybrid checkpointing [94], data aggregation [67], incremental checkpointing in the context of OpenMP [17], and data deduplication with the hope that processes involved in parallel HPC executions have enough similarities in memory [6] to reduce the size of the data sets necessary to be saved. However, what we mentioned five years ago is still valid: Programmers are in the best position to know the critical data of their application but they cannot use adaptive data protection other than by doing it manually. Annotations about ways to protect or check key data, computations, or communications are still a relevant direction.

Another way of reducing checkpoint time is to reduce the usage of disks for checkpoint storage. In-memory checkpointing has been demonstrated to be fast and scalable [99]. In addition, multilevel checkpointing technologies such as FTI [4] and SCR [78] are increasingly popular. Multilevel checkpointing involves combining several storage technologies (multiple levels) to store the checkpoint, each level offering specific overhead and reliability trade-offs. The checkpoint is first stored in the highest performance storage technology, which generally is the local memory of a local SSD. This level supports process failure but not node failure. The second level stores the checkpoint in remote memory of remote SSD. This second level supports a single-node failure.

The third level corresponds to the encoding the checkpoints in blocks and in distributing the blocks in multiple nodes. This approach supports multinode failures. Generally, the last level of checkpointing is still the parallel file system. This last level supports catastrophic failures such as full system outage. Charm++ provides similar multilevel checkpointing mechanisms [98]. The checkpoint period can be defined in different ways. Checkpoints also can be moved between levels in various ways, for example, by using a dedicated thread [4] or agents running on additional nodes [87]). A new semi-blocking checkpoint protocol leverages multiple levels of checkpoint to decrease checkpoint time [80]. A recent result computes the optimal combination of checkpoint levels and the optimal checkpoint intervals for all levels given the checkpoint size, the performance of each level, and the failure distributions for each level [82]. Other recent research concerns the understanding of the energy footprint of multilevel checkpointing and the exploration of trade-offs between energy optimization and performance overhead. The progress in multilevel checkpointing is outstanding, and some initiatives are looking at the definition of a standard interface.

By offering fast checkpointing at the first level, multilevel checkpointing also offers the opportunity to increase the checkpoint frequency and checkpoint at a smaller granularity. For example instead of checkpointing at the outermost loop of an iterative method, multilevel checkpointing could be used in some inner loops of the iterative method. Reducing the granularity of checkpointing is also the objective of task-based fault tolerance. The principle is to consider the application code as a graph of tasks and checkpoint the input parameter of each task on local storage with a notion of transaction: either a task is completed successfully and its results are committed in memory, or the task is restarted from its checkpointed input parameters. This approach is particularly relevant for future nonvolatile memory on computing nodes. The nonvolatile memory would store the input parameters checkpoints and the committed results. The OMPsS environment offers a first prototype of this approach [92]. One of the key questions is how to ensure a consistent restart in case of a node failure. Solutions need to consider how to store the execution graph and the local memory content in order to tolerate a diversity of failure scenarios.

The past five years have also seen excellent advances in checkpointing protocols. Classic checkpointing protocols are used to capture the state of distributed executions so that, after a failure, the distributed executions (or a part of them) restart and produce a correct execution [44]. In the HPC domain, classic checkpointing protocols are rarely used in production environments because most of the applications use application-level checkpointing that implicitly coordinates the capture of the distributed state and guarantees the correctness of the execution after restart. Without additional support, however, application-level checkpointing imposes global restart even if a single process fails. Message-logging fault-tolerant protocols offer a solution to avoid global restart and to restart only the failed processes. By logging the messages during the execution and replaying messages during recovery, they allow the local state reconstruction of the failed processes (partial restart). The main limitation is the necessity to log all messages of the execution. Several novel fault tolerance protocols overcome this limitation by reducing significantly the number of messages to log. They fall into the class of hierarchical fault tolerance protocols, forming clusters of processes and using coordinated checkpointing inside clusters and and message logging between clusters [61, 77, 84]. Such protocols need to manage causal dependencies between processes in order to ensure correct recovery. Multiple approaches have been proposed for reducing the overhead of storing causal dependency information [11, 20].

Partial restart opens opportunities for accelerated recovery. Charm++ and AMPI accelerate recovery by redistributing recovering processes on nodes that are not restarting [22]. Message logging can by itself accelerate recovery because messages needed by recovering processes are immediately available and messages to be sent to nonrestarting processes are just canceled [84]. These two aspects reduce the communication time during recovery. A recent advance in hierarchical fault tolerance protocols is the formulation and solving of the optimal checkpoint interval in this context [9]. Partial restart also opens the opportunity to schedule other jobs (than the recovering one) on resources running nonrestarting processes during the recovery of the failed processes. While this principle does not improve the execution performance for the job affected by the failure, it significantly improved the throughput of the platform that execute more jobs in a given amount of time compared with restarting all the processes of a job when a failure occurs [12]. This new result demonstrates another benefit of message-logging protocols that was not known before.

4.1.2. Forward Recovery

In some cases, the application can handle the error and execute some actions to terminate cleanly (checkpoint on failure [7]) or follow some specific recovery procedure without relying on classic rollback recovery. Such applications use some form of algorithmic fault tolerance. The application needs to be notified of the error and runs forward recovery steps that may involve access to past or remote data to correct (sometimes partially) or compensate the error and its effect, depending on the latency of the error detection.

A prerequisite for rollforward recovery is that some application processes and the runtime environment stay alive. While standard MPI does not preclude an application continuing after a fault [60], the MPI standard does not provide any specification of the behavior of an MPI application after a fault. For that purpose, researchers have developed several resilient MPI designs and implementations. The FT-MPI (fault-tolerant MPI) library [50, 66] was a first implementation of that approach. As the latest development, ULFM [8] allows the application to get notifications of errors and to use specific functions to reorganize the execution for forward recovery. Standardization of resilient MPI is complex; and despite several attempts, the MPI Forum has not reached a consensus on the principles of a resilient MPI. The GVR [100] system developed at the University of Chicago also provides mechanisms for application-specified forward error recovery. GVR design features two key concepts: (1) a global view of distributed arrays to processes and (2) versioning of these distributed arrays for resilience. APIs allow navigation of multiple versions for flexible recovery. Several applications have been tested with GVR.

4.1.3. Replication

Understanding of replication has progressed significantly in the past five years. Several teams have developed MPI and Charm++ prototypes offering process-level replication. Process-level replication of parallel executions is more reliable than replicated parallel executions under the assumption that the replication scheme itself is reliable. Several challenges need to be solved in order to make replication an attractive approach for resilience in HPC. First, the overhead of replication on the application execution time should be negligible. Second, replication needs to

guarantee equivalent state of process replicas,[4] which is not trivial because some MPI operations are not deterministic. A third, more complex challenge is the reduction of the resource and energy cost of replication. By default, replication needs twice the number of resources compared with nonreplicated execution.

One major replication overhead comes from the management of extra messages required for replication. Without specific optimization, when a replicated process sends a message to another replicated process, four communications of that message take place. rMPI addresses this problem by reducing the number of communications between replicas [51]. rMPI and MR-MPI [46] orchestrate non-MPI deterministic operations between process replicas to ensure equivalence of internal state. While rMPI and MR-MPI focus on process replication to address fail-stop errors, RedMPI [53] leverages process replication for detection of silent data corruptions (SDCs). The principle of RedMPI is to compare on the receiver side messages sent by replicated senders. If the message contents differ, a silent data corruption is suspected. RedMPI offers an optimization to avoid sending all messages needed in a pure replication scheme and to avoid comparing the full content of long messages. For each MPI message, replicated senders compute locally a hash of the message content, and only one of the replicated senders actually sends the message and the hash code to replicated receivers. Other replicas of the senders send only the hash code. Receivers then compare locally the hash code received from the replicated senders.

Reducing the resource overhead of process replication is a major challenge. One study [52], however, shows that replication can be more efficient than rollback recovery in some extreme situations where the MTBF of the system is extremely low and the time to checkpoint and restart is high. While recent progress in multilevel checkpointing make these situations unlikely, the results in [52] demonstrate that high rollback recovery overheads can lead to huge resource waste, up to the point where replication becomes a more efficient alternative. MR-MPI explores another way of reducing replication resource overhead by offering partial replication, where only a fraction of the processes are replicated. This approach is relevant when the platform presents some asymmetry in reliability (some resources being more fragile than others) and when this asymmetry can be monitored. Partial replication should be complemented by checkpoint restart to tolerate failures of non replicated processes [42]. Some libraries, for example, the ACR library [79] for MPI and Charm++ programs, cover both hard failures and SDCs from replication.

4.1.4. Failure Prediction

One domain that has made exceptional progress in the past five years is failure prediction. Before 2009, considerable research focused on how to avoid failures and their effects if failures could be predicted. Researchers explored the design and benefits of actions such as proactive migration of checkpointing [47, 74, 95]. The prediction of failures itself, however, was still an open issue. Recent results from the University of Illinois at Urbana-Champaign [54–56] and the Illinois Institute of Technology [101, 102] clearly demonstrate the feasibility of error prediction for different systems: the Blue Waters CRAY system based on AMD processors and NVIDIA GPUs and the Blue Gene system based on IBM proprietary components. Failure prediction techniques have progressed by combining data mining with signal analysis and methods to spot outliers.

[4]Internal state of process replicas do not need to be identical, but external interactions of each process replica should be consistent with a correct execution of the parallel application

Some failures can be predicted with more than 60% accuracy[5] on the Blue Waters system. Further research is needed to extend the results to other subsystems, such as the file system. We are still far from the accuracy needed to switch from pure reactive fault tolerance to truly proactive fault tolerance.

Other advances concern the combination of application-level checkpointing and failure prediction [10]. An important question is how to run the failure predictor on large infrastructures. One approach is to run a failure predictor in each node of a system in order to avoid the scalability limitation of global failure prediction. This approach faces two difficulties: (1) local failure prediction will impose an overhead on the application running on the node, and (2) local failure prediction is less accurate than global failure prediction because the failure predictors have only a local view. These two difficulties are explained in [10]. Another important question is how to compute the optimal interval of preventive checkpoints when a proportion of the failures are predicted [10]. In particular, many formulations of the optimal checkpoint interval problem consider that failures follow an exponential distribution of interarrival times. This approximation may be acceptable if we consider all failures in the system. Is it acceptable for failure that are not predicted correctly and for which preventive checkpointing is needed?

4.1.5. Energy Consumption

A new topic emerged in the community few years ago: energy consumption of fault tolerance mechanisms. The first paper on the topic we are aware of [41] presents a study of the energy footprint of fault tolerance protocols. The important conclusion of the paper is that, at least for clusters, little difference exists between checkpointing protocols. The study also shows that the difference in energy depends mainly on the difference in execution time and only slightly on the difference of power consumption of the operation performed by the different protocols. The reason is that the power consumptions of computing, checkpointing, and message logging are close in clusters.

Some teams [76] developed models for expected run time and energy consumption for global recovery, message logging, and parallel recovery protocols. These models show in an exascale scenario that parallel recovery outperforms coordinated checkpointing protocols since parallel recovery reduces the rework time. Other researchers [2] developed performance and energy models and formulated an optimal checkpointing period considering energy consumption as the objective to optimize.

Another research issue is the energy optimization of checkpoint/restart on local nonvolatile memory [85]. The intersection of resilience and energy consumption is also explored in a recent study [86].

4.1.6. Mitigating Silent Data Corruptions

One of the main challenges that HPC resilience faces at extreme scale and in particular in exascale systems and beyond is the mitigation of silent data corruptions (SDCs). As mentioned in previous sections, the risk of silent data corruptions is increasing. Several studies have explored the impact of SDCs in execution results [16, 43, 73]. These studies show that a majority of SDCs leads to noticeable impacts such as crashes and hangs but that only a small fraction of them

[5]Accuracy here is defined as the prediction recall: the number of correctly predicted failures divided by the number of actual failures

actually corrupt the results. Nevertheless, the likelihood of generating wrong results because of SDC is significant enough to warrant study of mitigation techniques.

An excellent survey of error detection techniques is presented in [71]. A classic way to detect a large proportion of SDCs (but not all) is replicating executions and comparing results. Following this approach, RedMPI [53] offers replication at the MPI process level, and replication at the thread level is studied in [96] by leveraging hardware transactional memory. The first issue with SDC detection by replication is the overhead in resources. A second issue is that, in principle, comparison of results supposes that execution generates identical results, which means obtaining bit-to-bit deterministic results from executions using same input parameters. Applications may not have this property because of nondeterministic operations performed during the execution. In general the trend toward more asynchrony and more load balancing plays against deterministic executions. Then the detection from replication becomes the problem of evaluating the similarity of results generated from replicated executions. Quantification of this similarity is an extremely hard problem because it assumes an understanding of how results diverge as a result of indeterministic operations, which itself depends on the thorough understanding of roundoff error propagation. Consequently, in SDC detection explore solutions that requires less resources and potentially relax the precision of detection.

A recent direction could be called approximate replication. The principle of approximate replication is to run the normal computation along with a an approximate computation that generates an approximate result. The comparison is then performed between the result and the approximate result. The approximate calculation gives upper and lower bounds within which the result of the normal calculation should be. Results outside the bounds are suspect; and corrective actions, such as further verification or re-execution, may be triggered. Approximate replication is a generic approach. It could be performed at the numerical method [5] level. It also could be used at the hardware level by comparing floating-point results of a normal operator with the ones of an approximate operator [37, 40, 75].

Another important issue related to SDC detection is the choice of methodology for evaluating and comparing algorithms. The standard process is to inject faults and obtain statistics on coverage and recovery overheads. Simulating hardware at the physical level is not feasible, and simulating it at a register transfer level is onerous. Most researchers inject faults in higher-level simulators [33]. But it is difficult to validate such fault injectors and know how the fault patterns they exhibit are related to faults exhibited by real hardware. A recent study compares different injection methods and their accuracy for the SPECint benchmark [27]. However, the accuracy of injection in HPC applications is still an open problem.

4.1.7. Integrative Approaches

The community has expressed in several reports the need for integrative approaches considering all layers from the hardware to the application. At least five projects explore this notion in different ways. One recent study demonstrates the benefit of cross-layer management of faults [64]. The containment domains approach [28] proposes a model of fault management based on a hierarchical design. Domains at a given level are supposed to detect and contain faults using techniques available at that level. If faults cannot be handled at that level, then the fault management becomes the responsibility of the next level in the hierarchy. Containment approaches operate between domains of the same level and between levels. Such approaches are applicable, in principle, at the hardware level and at all other software levels.

A different approach is proposed by the BEACON pub/sub mechanism in the Argo project [48], the GIB infrastructure in the Hobbes project [49], and the open resilience framework of the GVR project [100]. These mechanisms extend in different ways the concept of communication between software layers, originally proposed in the CIFTS project [62]. BEACON, GIB, and CIFTS/FTB are backplanes implementing exchanges of notifications and commands about errors and failures between software running on a system. Response managers should complement backplane infrastructures by lessening error and failure events and implementing mitigation plans.

4.2. Algorithmic Approaches

In our paper of five years ago, we discussed approaches for either recovering from or successfully ignoring faults, including early efforts in algorithm-based fault tolerance and in fault-oblivious iterative methods. Since that time, significant progress has been made in algorithmic approaches to detecting and recovering from faults. For example, the 2014 SIAM conference on parallel processing featured four sessions of seventeen talks covering many aspects of resilient algorithms. Here, we describe some of the progress in this area. The work presented is just a sampling of recent results in algorithm-based fault tolerance and is meant to give the reader a starting point for further exploration of this area.

Perhaps the most important change has been a clearer separation of the faults into two categories: fail-stop (a process fails and stops, causing a loss of all state in the process) and fail-continue (a process fails but continues, often due to a transient error). The latter has two important subcases based on whether the fault is detected or not. An example of the former is a failure of an interconnect cable, allowing the messaging layer to signal a failure but permitting the process to continue. An example of the latter is a undetected, uncorrectable memory error. Considerable progress has been made in the area of fail-continue faults, spurred by the recognition that transient faults (sometimes called soft faults), rather than fail-stop faults, are likely to be the most important type of faults in exascale systems.

Dense Linear Algebra. Fault-tolerant algorithms for dense linear algebra have a long history; at the time of our original paper, several techniques for algorithms such as matrix-matrix multiplication that made use of clever checksum techniques were already known [24, 65]. Progress in this area includes the development of ABFT for dense matrix multiplication [25] and factorizations [36] that addresses fail-stop faults. These provide the necessary building blocks to address complete applications; for example, a version of the HPLinpack benchmark that handles fail-stop faults with low overhead has been demonstrated [31]. Recent work has extended to the handling of soft faults or the fail-continue case for dense matrix operations [30].

Sparse Matrices and Iterative Methods. Algorithms involving sparse matrices and using iterative methods are likely to be important for exascale systems because many of the science applications that are expected to run on these systems solve large sparse linear and nonlinear systems. Work here has looked at both fail-stop and fail-continue faults. For example, [88, 89] evaluates a number of algorithms for both stability and accuracy in the presence of faults and describes an approach for transforming applications to be more resilient. A related approach can detect soft errors in Krylov methods by using properties of the algorithm so that the computation can be restarted or corrected [23]. Recent work has also looked at working with the system to

provide more information and control for the library or application in handling likely errors, such as different types of DRAM failures [14].

Designing for Selective Reliability. One of the limitations of ABFT is that it can address errors only in certain parts of the application, for example, in the application's data but not in the program's instructions. This situation can be addressed in part by using judicious replication of pointers and other data structures [21], though this still leaves other parts of the code unprotected. An alternative is to consider variable levels of reliability in the hardware—using more reliable hardware for hard-to-repair problems and less reliable hardware where the algorithm can more easily recover and structure the ABFT to take advantage of the different levels of reliability [13, 72].

Efficient Checkpoint Algorithms. For many applications, a checkpoint approach is the only practical one, as discussed in Section 4.1. In-memory checkpoints can provide lower overheads than I/O systems but in their simplest form consume too much memory. Thus, approaches that use different algorithms for error-correcting code have been developed. An early example that exploited MPI I/O semantics to provide efficient blocking and resilience for file I/O operations is [59]. A similar approach has been used in SCR [78]. A more sophisticated approach, building on the coding and checksum approach for dense linear algebra, is given in [26].

5. Future Research Areas

The community has identified several research areas that are particularly important to the development of resilient applications on exascale systems:

- Characterization of hardware faults
- Development of a standardized fault-handling model
- Improved fault prediction, containment, detection, notification, and recovery
- Programming abstractions for resilience
- Standardized evaluation of fault-tolerance approaches

Characterization of hardware faults is essential for making informed choice about research needs for exascale resilience. For example, if silent hardware faults are exceedingly rare, then the hard problem of detecting such errors in software or tolerating their impact can be ignored. If errors in storage are exceedingly rare, while errors in compute logic are frequent, then research on mechanisms for hardening data structures and detecting memory corruptions in software is superfluous.

Suppliers and operators of supercomputers are often cagey about statistics on the frequency of errors in their systems. A first step would be to systematically and continually gather consistent statistics about current systems. Experiments could also be run in order to detect and measure the frequency of SDCs, using spare cycles on large supercomputers.

As work progresses on the next generation of integrated circuits, experiments will be needed to better characterize their sensitivity to various causes of faults.

Development of a standardized fault-handling model is key to providing guidance to application and system software developers about how they will be notified about a fault, what types of faults they may be notified about, and what mechanisms the system provides to assist recovery from the fault. Applications running on today's petascale systems are not even notified of faults or given options as to how to handle faults. If the application happens to detect an

error, the computer may also eventually detect the error and kill the application automatically, making application recovery problematic. Therefore, today's petascale applications all rely on checkpoint-restart rather than on resilient algorithms.

Development of a standardized fault-handling model implies that computer suppliers can agree on a list of common types of fault that they are able to notify applications about. Recovery from a node failure differs significantly from recovery from an uncorrectable data value corruption. Resilient exascale applications may incorporate several recovery techniques targeted to particular types of faults. Even so, these resilient exascale applications are expected to be able to recover from only a subset of the common types of faults. Also, the ability to recover depends on the time from fault occurrence to fault notification. At the least, "fence" mechanisms are needed in order to ensure that software waits until all pending notifications that could affect previous execution are delivered.

A standardized fault-handling model needs to have a notification API that is common across different exascale system providers. The exact mechanism for notifying applications, tools, and system software components is not as critical as the standardization of the API. A standardized notification API will allow developers to develop portable, resilient codes and tools.

A fault-handling model also needs to define the recovery services that are available to applications. For example, if notified of a failed node, is there a service to add a spare node or to migrate tasks to other nodes? If the application is designed to survive data corruption, is there a way to specify reliable memory regions? Can one specify reliable computational regions in a code? In today's petascale systems, if such services exist, they are available only to the RAS system and are not exposed to the users. A useful fault model would have a standard set of recovery services that all computer suppliers would provide to the software developers to develop resilient exascale applications.

The fault-handling model should support hierarchical fault handling, with faults handled in the smallest context capable of doing so. A fault that affects a node should be signaled to that node, if it is capable of initiating a recovery action. For example, memory corruption that affects only application data should be handled first by the node kernel. The model should provide mechanisms for escalating errors. If a node is not responsive or cannot handle an error affecting it, then the fault should be escalated to an error handler for the parallel application using that node. If this error handler is not responsive or cannot handle the error, then the error should be escalated to a system error handler.

Improved fault prediction, containment, detection, notification, and recovery research requires major efforts in one area: the *detection* of silent (undetected) errors. Indeed, exascale is highly unlikely to be viable without error detection.

Significant uncertainty remains about the likely frequency of SDC in future systems; one can hope that work on the characterization of hardware faults will reduce this uncertainty. Much uncertainty also remains about the impact of SDCs on the executing software. We do not have, at this point, technologies that can cope with frequent SDCs, other than the brute force solutions of duplication or triplication.

Arguably, significant research has been done on algorithmic methods for handling SDCs. Current research, however, focuses on methods that apply to specific kernels that are part of specific libraries. We do not have general solutions, and we do not know whether current approaches could cover a significant fraction of computation cycles on future supercomputers. It is

imperative to develop general detection/recovery techniques or show how to combine algorithmic methods with other methods.

Research on algorithmic methods considers only certain types of errors, for example, the corruption of data values, but not the corruption of the executing code, or hardware errors that affect the interrupt logic. It is important to understand whether such omission is justified by the relative frequency of the different types of errors or, if not, what mechanisms can be used to cope with errors that are not within the scope of algorithmic methods.

Moreover, fault detection mechanisms that are not algorithmic specific but use compiler and runtime techniques may be harder, but they would have a higher return because they would apply to all application codes.

Fault *containment* tries to keep the damage caused by the fault from propagating to other cores, to other nodes, and potentially to the corruption of checkpointed data, rendering recovery impossible. Successful containment requires early detection of faults before many computations are done and before that data is transmitted to other parts of the system. Once the fault has propagated, local recovery is no longer viable; and if detection does not occur before an application checkpoint, even global recovery may be impossible. Successful fault containment is key to low-cost recovery.

Fault *prediction* does not replace fault detection and correction, but it can significantly increase the effective system and application MTBF, thus significantly decreasing time wasted to checkpoint and recovery. Furthermore, techniques developed for fault prediction help root cause analysis and thus reduce maintenance time.

Fault *notification* is an essential component of the previously described fault-handling model. The provision of robust, accurate, and scalable notification mechanisms for the HPC environment will require new techniques.

Current overheads for *recovery* from system failures are significant; a system may take hours to return to service. Clearly, the time for system recovery must be reduced.

Programming abstractions for resilience will be able to grow out of a standardized fault handling model. Many programming abstractions have been explored, and several examples were described in previous sections. These research explorations have not shown a single programming abstraction for resilience that works for all cases. In fact, they show that several programming abstractions will need to be developed and supported in order to develop resilient exascale applications.

The development of fault-tolerant algorithms requires various resilience services: For example, some parts of the computation must execute correctly, while other parts can tolerate errors; some data structures must be persistent; others may be lost or corrupted, provided that errors are detected before corrupted values are used; and still other data structures can tolerate limited amounts of corruptions.

More fundamentally, one needs semantics for programs that enable us to express their reliability properties. The development of efficient fault-tolerant algorithms requires the programmer to think of computations as stochastic processes, with performance and outcome dependent on the distribution of faults. The validation or testing of such stochastic programs requires new methods as well.

Resilience services and appropriate semantics would also facilitate the development of system code. Many system failures, in particular failures in parallel file systems, are due to various forms of resource exhaustion, as servers become overloaded or short in memory and fail to respond

in a timely manner. The proper configuration of such systems is a trial-and-error process. One would prefer systems that are resilient by design.

Standardized evaluation of fault tolerance approaches will provide a way to measure the efficiency of a new approach compared with other known approaches. It will also provide a way to measure the effectiveness of an approach on different architectures and at different scales. The latter will be important to determine whether the approach can scale to exascale. Even fault-tolerant Monte Carlo algorithms have been shown to have problems scaling to millions of processors [45]. Standardized evaluation will involve development of a portable, scalable test suite that simulates all the errors from the fault model and measures the recovery time, services required, and the resources used for a given resilient exascale application.

Acknowledgments: We thank Estaban Meneses for his help in the software section. This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, and under Award DESC0004131. This work was also supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-FG02-13ER26138/DE-SC0010049. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number ACI 1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

References

1. The Blue Waters super system for super science. *Contemporary High Performance Computing From Petascale toward Exascale, Jeffrey S. Vetter, editor, Chapman and Hall/CRC, pages 339–366, ISBN: 978-1-4665-6834-1, 2013.*

2. Guillaume Aupy, Anne Benoit, Thomas Hérault, Yves Robert, and Jack Dongarra. Optimal checkpointing period: Time vs. energy. *CoRR*, abs/1310.8456, 2013.

3. A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

4. 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. 2011 Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC11)*. ACM, 2011.

5. Austin R Benson, Sven Schmit, and Robert Schreiber. Silent error detection in numerical time-stepping schemes. *International Journal of High Performance Computing Applications*, April, 2014.

6. Susmit Biswas, Bronis R. de Supinski, Martin Schulz, Diana Franklin, Timothy Sherwood, and Frederic T. Chong. Exploiting data similarity to reduce memory footprints. In *Proceedings of IEEE IPDPS*, pages 152–163, 2011.

7. W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Extending the scope of the checkpoint-on-failure protocol for forward recovery in standard MPI, concurrency and computation: Practice and experience, special issue: Euro-par 2012. July 2013.

8. Wesley Bland. User level failure mitigation in MPI. In *Euro-Par 2012: Parallel Processing Workshops*, pages 499–504. Springer, 2013.

9. G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale, concurrency and computation: Practice and experience. November 2013.

10. Mohamed-Slim Bouguerra, Ana Gainaru, Leonardo Arturo Bautista-Gomez, Franck Cappello, Satoshi Matsuoka, and Naoya Maruyama. Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing. In *Proceedings of IEEE IPDPS*, pages 501–512, 2013.

11. A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Correlated set coordination in fault tolerant message logging protocols, concurrency and computation: Practice and experience. Vol. 25, No. 4:pp. 572–585, 2013.

12. Aurelien Bouteiller, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Hrault, and Yves Robert. Multi-criteria checkpointing strategies: Response-time versus resource utilization. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 420–431. Springer Berlin Heidelberg, 2013.

13. P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. Fault-tolerant linear solvers via selective reliability. *ArXiv e-prints*, June 2012.

14. PatrickG. Bridges, Mark Hoemmen, KurtB. Ferreira, MichaelA. Heroux, Philip Soltero, and Ron Brightwell. Cooperative application/OS DRAM fault recovery. In Michael Alexander, Pasqua DAmbra, Adam Belloum, George Bosilca, Mario Cannataro, Marco Danelutto, Beniamino Martino, Michael Gerndt, Emmanuel Jeannot, Raymond Namyst, Jean Roman, StephenL. Scott, JesperLarsson Traff, Geoffroy Valle, and Josef Weidendorfer, editors, *Euro-Par 2011: Parallel Processing Workshops*, volume 7156 of *Lecture Notes in Computer Science*, pages 241–250. Springer Berlin Heidelberg, 2012.

15. Allan G Bromley. Charles Babbage's analytical engine, 1838. *Annals of the History of Computing*, 4(3):196–217, 1982.

16. Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 155–164. ACM, 2008.

17. Greg Bronevetsky, Daniel J. Marques, Keshav K. Pingali, Radu Rugina, and Sally A. McKee. Compiler-enhanced incremental checkpointing for openmp applications. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 275–276, New York, NY, USA, 2008. ACM.

18. Franck Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

19. Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

20. Franck Cappello, Amina Guermouche, and Marc Snir. On communication determinism in parallel hpc applications. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–8. IEEE, 2010.

21. Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

22. Sayantan Chakravorty and L. V. Kale. A fault tolerance protocol with fast fault recovery. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

23. Zizhong Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 167–176, New York, NY, USA, 2013. ACM.

24. Zizhong Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Parallel and Distributed Processing Symposium, International*, page 76, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

25. Zizhong Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *Parallel and Distributed Systems, IEEE Transactions on*, 19(12):1628–1641, Dec 2008.

26. Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 213–223, New York, NY, USA, 2005. ACM.

27. Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10. IEEE, 2013.

28. Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *the Proceedings of SC12*, November 2012.

29. John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

30. Teresa Davies and Zizhong Chen. Correcting soft errors online in lu factorization. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 167–178, New York, NY, USA, 2013. ACM.

31. Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: A fault tolerant implementation without checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 162–171, New York, NY, USA, 2011. ACM.

32. N. DeBardeleben, J. Laros, J. Daly, S. Scott, C. Engelmann, and W. Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. Technical Report LA-UR-10-00030, DARPA, January 2010.

33. Nathan DeBardeleben, Sean Blanchard, Qiang Guan, Ziming Zhang, and Song Fu. Experimental framework for injecting logic errors in a virtual machine to profile applications for soft error resilience. In *Proceedings of the 2011 International Conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 282–291, Berlin, Heidelberg, 2012. Springer-Verlag.

34. Catello Di Martino, F Baccanico, W Kramer, J Fullop, Z Kalbarczyk, and R Iyer. Lessons learned from the analysis of system failures at petascale: The case of Blue Waters. In *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, 2014.

35. R.W. Downing, J.S. Nowak, and L.S. Tuomenoksa. No. 1 ESS maintenance plan. *Bell System Technical Journal*, 43:5:1961–2019, 1964.

36. Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 225–234, New York, NY, USA, 2012. ACM.

37. Peter D. Dben, Jaume Joven, Avinash Lingamneni, Hugh McNamara, Giovanni De Michel, Krishna V. Palem, and T. N. Palmer. Low-cost concurrent error detection for floating-point unit (fpu) controllers. *Philosophical Transactions of the Royal Society A, 20130276*, 372(2018), 2014.

38. John Daly (editor), Bob Adolf, Shekhar Borkar, Nathan DeBardeleben, Mootaz Elnozahy, Mike Heroux, David Rogers, Rob Ross, Vivek Sarkar, Martin Schulz, Marc Snir, and Paul Woodward. Inter Agency Workshop on HPC Resilience at Extreme Scale. `http://institute.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf`, February 2012.

39. Mootaz Elnozahy (editor), Ricardo Bianchini, Tarek El-Ghazawi, Armando Fox, Forest Godfrey, Adolfy Hoisie, Kathryn McKinley, Rami Melhem, James Plank, Partha Ranganathan, and Josh Simons. System resilience at extreme scale. Technical report, Defense Advanced Research Project Agency (DARPA), 2009.

40. Patrick J Eibl, Andrew D Cook, and Daniel J Sorin. Reduced precision checking for a floating point adder. In *Defect and Fault Tolerance in VLSI Systems, 2009. DFT'09. 24th IEEE International Symposium on*, pages 145–152. IEEE, 2009.

41. Mohammed el Mehdi Diouri, Olivier Glück, Laurent Lefèvre, and Franck Cappello. Energy considerations in checkpointing and fault tolerance protocols. In *Proceedings of FTXS workshop, IEEE/IFIP DSN'12*, pages 1–6, 2012.

42. J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for hpc. In *Proceedings of the IEEE 32nd International Conference on Distributed Computing Systems ICDCS*, pages 615–626, June 2012.

43. James Elliott, Mark Hoemme, and Frank Mueller. Evaluating the impact of SDC on the GMRES iterative solver4. In *Proceedings of International Parallel and Distributed Processing Symposium, IPDPS'14*, 2014.

44. Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

45. Christian Engelmann. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Generation Computer Systems (FGCS)*, 30(0):59–65, January 2014.

46. Christian Engelmann and Swen Böhm. Redundant execution of HPC applications with MR-MPI. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011*, pages 31–38, 2011.

47. Christian Engelmann, Geoffroy R. Vallée, Thomas Naughton, and Stephen L. Scott. Proactive fault tolerance using preemptive migration. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP) 2009*, pages 252–257, February 18-20, 2009.

48. Pete Beckman et al. Argo: An exascale operating system. In *http://www.mcs.anl.gov/project/argo-exascale-operating-system.*

49. Ron Brightwell et al. Hobbes - an operating system for extreme-scale systems. In *http://xstack.sandia.gov/hobbes/.*

50. Graham E. Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, UK, 2000. Springer-Verlag.

51. Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, and Ron Brightwell. rMPI: increasing fault resiliency in a message-passing environment. Technical Report SAND2011-2488, Sandia National Laboratories, Albuquerque, NM, 2011.

52. Kurt B Ferreira, Rolf Riesen, Patrick Bridges, Dorian Arnold, Jon Stearley, H Laros III James, Ron Oldfield, Kevin Pedretti, and Ron Brightwell. Evaluating the viability of process replication reliability for exascale systems. In *ACM/IEEE Conference on Supercomputing (SC11)*, Nov 2011.

53. David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

54. Ana Gainaru, Franck Cappello, and William Kramer. Taming of the shrew: modeling the normal and faulty behaviour of large-scale hpc systems. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 1168–1179. IEEE, 2012.

55. Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Fault prediction under the microscope: A closer look into HPC systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE/ACM SC'12*, page 77. IEEE Computer Society Press, 2012.

56. Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Failure prediction for HPC systems and applications current situation and open issues. *International Journal of High Performance Computing Applications*, 27(3):273–282, 2013.

57. Al Geist. Private communication, 2012.

58. Al Geist, Bob Lucas, Marc Snir, Shekhar Borkar, Eric Roman, Mootaz Elnozahy, Bert Still, Andrew Chien, Robert Clay, John Wu, Christian Engelmann, Nathan DeBardeleben, Rob Ross Larry Kaplan Martin Schulz, Mike Heroux, Sriram Krishnamoorthy, Lucy Nowell, Abhinav Vishnu, and Lee-Ann Talley. U.S. Department of Energy fault management workshop. Technical report, DOE, 2012.

59. William Gropp, Robert Ross, and Neill Miller. Providing efficient I/O redundancy in MPI environments. In Dieter Kranzlmüller, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number LNCS3241 in Lecture Notes in Computer Science, pages 77–86. Springer Verlag, 2004. 11th European PVM/MPI User's Group Meeting, Budapest, Hungary.

60. William D. Gropp and Ewing Lusk. Fault tolerance in MPI programs. *International Journal of High Performance Computer Applications*, 18(3):363–372, 2004.

61. Amina Guermouche, Thomas Ropars, Marc Snir, and Franck Cappello. Hydee: Failure containment without event logging for large scale send-deterministic MPI applications. In *Proceedings of IEEE IPDPS*, pages 1216–1227, 2012.

62. Rinku Gupta, Pete Beckman, B-H Park, Ewing Lusk, Paul Hargrove, Al Geist, Dhabaleswar K Panda, Andrew Lumsdaine, and Jack Dongarra. CIFTS: A coordinated infrastructure for fault-tolerant systems. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 237–245. IEEE, 2009.

63. Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for Linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

64. Chen-Han Ho, Marc de Kruijf, Karthikeyan Sankaralingam, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Mechanisms and evaluation of cross-layer fault-tolerance for supercomputing. In *Proceedings of ICPP*, pages 510–519, 2012.

65. Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

66. Joshua Hursey, Richard L. Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G. Solt. Run-through stabilization: An MPI proposal for process fault tolerance. In *Proceedings of EuroMPI*, pages 329–332, 2011.

67. Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. de Supinski, and Rudolf Eigenmann. Mcrengine: A scalable checkpointing system using data-aware aggregation and compression. *Scientific Programming*, 21(3-4):149–163, 2013.

68. William M. Jones, John T. Daly, and Nathan DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 276–279, New York, NY, USA, 2010. ACM.

69. D. S. Katz, J. Daly, N. DeBardeleben, M. Elnozahy, B. Kramer, L. Lathrop, N. Nystrom, K. Milfeld, S. Sanielevici, S. Cott, , and L. Votta. 2009 fault tolerance for extreme-scale computing workshop, Albuquerque, NM - March 19-20, 2009. Technical Report Technical Memorandum ANL/MCS-TM-312, MCS, ANL, December 2009.

70. D.S. Katz and R.R. Some. NASA advanced robotic space exploration. *Computer*, 36(1):52–61, 2003.

71. Ikhwan Lee, Michael Sullivan, Evgeni Krimer, Dong Wan Kim, Mehmet Basoglu, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Survey of error and fault detection mechanisms v2. Technical Report TR-LPH-2012-001, LPH Group, Department of Electrical and Computer Engineering, The University of Texas at Austin, December 2012.

72. Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S Vetter. Rethinking Algorithm-Based Fault Tolerance with a Cooperative Software-Hardware Approach. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

73. Dong Li, Jeffrey S Vetter, and Weikuan Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *SC12: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, Salt Lake City, 11/2012 2012.

74. Antonina Litvinova, Christian Engelmann, and Stephen L. Scott. A proactive fault tolerance framework for high-performance computing. In *Proceedings of the 9th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2010*, 2010.

75. M. Maniatakos, P. Kudva, B.M. Fleischer, and Y. Makris. Low-cost concurrent error detection for floating-point unit (FPU) controllers. *Computers, IEEE Transactions on*, 62(7):1376–1388, July 2013.

76. E. Meneses, O. Sarood, and L.V. Kalé. Energy profile of rollback-recovery strategies in high performance computing. *Parallel Computing*, 2014.

77. Esteban Meneses, Laxmikant V. Kalé, and Greg Bronevetsky. Dynamic load balance for optimized message logging in fault tolerant HPC applications. In *Proceedings of IEEE Cluster*, pages 281–289, 2011.

78. A. Moody, G. Bronevetsky, K. Mohror, and B.R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 International Conference on High Performance Computing, Networking, Storage and Analysis (SC10)*, pages 1–11, 2010.

79. Xiang Ni, Esteban Meneses, Nikhil Jain, and Laxmikant V. Kale. ACR: Automatic checkpoint/restart for soft and hard error protection. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13. IEEE Computer Society, November 2013.

80. Xiang Ni, Esteban Meneses, and Laxmikant V. Kalé. Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm. In *Proceedings of IEEE Cluster'12*, Beijing, China, September 2012.

81. Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. Nvcr: A transparent checkpoint-restart library for nvidia cuda. In *IPDPS Workshops*, pages 104–113, 2011.

82. Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications. Optimization of multi-level checkpoint model for large scale HPC applications. In *Proceedings of IEEE IPDPS 2014*, 2014.

83. A. Rezaei, G. Coviello, CH. Li, S. Chakradhar, and F Mueller. Snapify: Capturing snapshots of offload applications on Xeon Phi manycore processors. In *Proceedings of High-Performance Parallel and Distributed Computing, HPDC'14*, 2014.

84. Thomas Ropars, Tatiana V. Martsinkevich, Amina Guermouche, Andre Schiper, and Franck Cappello. Spbc: leveraging the characteristics of MPI HPC applications for scalable checkpointing. In *Proceedings of IEEE/ACM SC*, page 8, 2013.

85. Takafumi Saito, Kento Sato, Hitoshi Sato, and Satoshi Matsuoka. Energy-aware I/O optimization for checkpoint and restart on a NAND flash memory system. In *In Proceedings of the Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, pages 41–48, 2013.

86. Osman Sarood, Esteban Meneses, and L. V. Kale. A cool way of improving the reliability of HPC machines. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE/ACM SC'13*, Denver, CO, USA, November 2013.

87. Kento Sato, Naoya Maruyama, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R. de Supinski, and Satoshi Matsuoka. Design and modeling of a non-blocking checkpointing system. In *Proceedings of IEEE/ACM SC'12*, page 19, 2012.

88. Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. *42rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

89. Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

90. Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):129–173, May 2014.

91. L. Spainhower and T.A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5.6):863–873, 1999.

92. Omer Subasi, Javier Arias, Jesus Labarta, Osman Unsal, Adrian Cristal, and Barcelona Supercomputing Center. Leveraging a task-based asynchronous dataflow substrate for efficient and scalable resiliency, research report of polythecnic university of catalonia - computer architecture department. num: Upc-dac-rr-cap-2013-12. 2014.

93. Alexander Randall V. The Eckert tapes: Computer pioneer says ENIAC team couldn't afford to fail - and didn't. *Computerworld*, 40(8), February 2006.

94. Chao Wang, F. Mueller, C. Engelmann, and S.L. Scott. Hybrid checkpointing for MPI jobs in HPC environments. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 524–533, Dec 2010.

95. Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration and back migration in HPC environments. *J. Parallel Distrib. Comput.*, 72(2):254–267, February 2012.

96. Gulay Yalcin, Osman Sabri Unsal, and Adrian Cristal. Fault tolerance for multi-threaded applications by leveraging hardware transactional memory. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 4. ACM, 2013.

97. John W Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

98. Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for ampi and charm++. *SIGOPS Oper. Syst. Rev.*, 40(2):90–99, April 2006.

99. Gengbin Zheng, Xiang Ni, and L. V. Kale. A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale, in Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS). Boston, USA, June 2012.

100. Ziming Zheng, Andrew A. Chien, and Keita Teranishi. Fault tolerance in an inner-outer solver: a GVR-enabled case study. In *Proceedings of VECPAR 2014, Lecture Notes in Computer Science*, 2014.

101. Ziming Zheng, Zhiling Lan, Rinku Gupta, Susan Coghlan, and Peter Beckman. A practical failure prediction with location and lead time for Blue Gene/P. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pages 15–22. IEEE, 2010.

102. Ziming Zheng, Li Yu, Wei Tang, Zhiling Lan, Rinku Gupta, Narayan Desai, Susan Coghlan, and Daniel Buettner. Co-analysis of RAS log and job log on Blue Gene/P. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 840–851. IEEE, 2011.