Addressing Failures in Exascale Computing

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 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
Eric Van Hensbergen

Executive Summary

The current approach to resilience for large high-performance computing (HPC) machines is based on global application checkpoint/restart. The state of each application is checkpointed periodically; if the application fails, then it is restarted from the last checkpoint. Preserving this approach is highly desirable because it requires no change in application software.

The success of this method depends crucially on the following assumptions:

1. The time to checkpoint is $\ll$ mean time before failure (MTBF).
2. The time to restart (which includes the time to restore the system to a consistent state) is $\ll$ MTBF;
3. The checkpoint is correct—errors that could corrupt the checkpointed state are detected before the checkpoint is committed.
4. Committed output data is correct (output is committed when it is read).

It was not clear that these assumptions are currently satisfied. In particular, can one ignore silent data corruptions (SDCs)? It is clear that satisfying these assumptions will be harder in the future for the following reasons:

- MTBF is decreasing faster than disk checkpoint time.
- MTBF is decreasing faster than recovery time—especially recovery from global system failures.
- Silent data corruptions may become too frequent, and errors will not be detected in time.
- The output of the application may be used in real time.

Each of these obstacles can be overcome in a different way: (1) we can checkpoint in RAM, rather than disk; (2) we can build global operating systems that fail less frequently or recover faster; (3) we can design hardware with lower SDC rates or, alternatively, use software to detect SDCs or tolerate them; and (4) we can use replication for the relatively rare real-time supercomputing applications.

The different approaches are associated with different costs, risks, and uncertainties; we do not have enough information to choose one approach now. Therefore, we considered the following three design points: (1) business as usual, (2) system-level resilience, and (3) application-level resilience.

Design point 1: Business as Usual This approach continues to use global checkpoint/restart. Hybrid checkpoint methods (using DRAM or NVRAM, as well as disk) can provide fast checkpoint and application restart time and can accommodate failure rates that are an order of magnitude higher than today's failure rates. The additional power consumption is low, but the acquisition cost of platforms will rise because of the need for additional memory.

Two key technologies are needed for this approach to be feasible. (1) low SDC frequency (same as now) and low frequency of system failures or an order of magnitude improvement in system recovery time.

Maintaining the current rate of hardware SDC seems possible, at the expense of $<$20% of additional silicon and energy; and vendor research can further lower the overhead. However, supercomputing needs both low power and low SDC rate. It is not clear that there is a large market for this combination; hence it is not clear that this combination will appear in lower-cost volume products.

Silent hardware errors can be masked in software—the simple approach is to duplicate computations and compare results. Since most compute time and compute energy are spent moving data, a good hardware/software combination should enable the duplication of computation at a cost that is much less than

100%, as data movement need not be replicated. Such combined solutions might be more if they provide the user with a choice of higher performance or higher reliability.

Reducing system failures or system recovery time seems feasible but requires new research. This includes development of fault-tolerance mechanisms that avoid or mitigate failures due to hardware or software. A significant body of knowledge in this area exists but has not been applied to HPC; research should focus on the new problems that are likely to arise with this new application.

Design point 2: System-Level Resilience This approach assumes that vendors will not provide sufficiently low SDC rates at an acceptable acquisition and operation cost but that a combination of hardware and software technologies can hide the increased failure rates from the application.

Various techniques are available for detecting SDCs in software, and applications can tolerate some SDCs. This design point is not fundamentally different from the first one. It does, however, give more leeway to vendors on selecting where they apply hardware and where they apply software solutions. It also calls on the wider system software community to develop resilience capability for runtimes and libraries.

Design point 3: Application-Level Resilience This approach assumes that application codes will need to be modified in order to handle the increased failure rate, since neither hardware nor firmware will be efficient enough in avoiding SDCs. The critical need is for codes that can tolerate SDCs or detect SDCs.

Significant research does exist in this area, but the research is focused on specific algorithms and methods. It is not clear that current approaches can cover a large fraction of DOE's workloads.

Recommendations: Any future solution to the resilience problem must be rooted in a clear understanding of the current situation: What is the cause of failures, and what is the frequency of SDC? This information is not readily available. Thus, we make the following recommendations.

Recommendation 1: Perform experiments to estimate SDC rates on current platforms. Collect, in a consistent manner, information on detected failures on current large DOE systems and make it available to researchers in a suitable form.

The main downside of the first design point is the potential cost of platforms using this approach; the second and, especially, the third design points have risks in terms of problems with no known solutions. The three design points motivate fairly different investment streams.

Recommendation 2: Work with industry to refine estimates on cost (design complexity, transistors, power) of keeping SDC rates low and to understand the market opportunities for low-power, high-resilience technology. Aim for an early decision on levels of error detection and correction that will be provided by hardware. Invest in research on combined hardware/software error detection that is transparent to the user.

Some technologies are required no matter what design point is picked; some are beneficial no matter what design point is picked. In the first category is the need for more robust system software infrastructure and for faster system recovery, as well as good support for hybrid checkpoints. In the second category is fault prediction that increases the effective MTBF and, therefore, reduces the cost of fault tolerance.

Recommendation 3: Invest first in R&D for technologies that are required or beneficial no matter what design point is picked.

Solutions must apply to all or to the large majority of DOE's workloads. A solution that is specific to a subset of these workloads can, at best, accelerate the execution of these workloads but will not avoid the need for developing a general solution.

Recommendation 4: Focus research on application-level error handling to solutions that can apply to all or to the large majority DOE workloads. Point solutions that address specific codes should be a second priority.

Contents

1 Introduction

> "The problems are solved, not by giving new information, but by arranging what we have known since long." – Ludwig Wittgenstein, *Philosophical Investigations*

This white paper is the result of the workshop on "Addressing Failures in Exascale Computing" held in Park City, Utah, August 4–11, 2012. The workshop was sponsored by the Institute for Computing in Science (ICiS). More information about ICiS activities can be found at http://www.icis.anl.gov/about. The charter of this workshop was to establish a common taxonomy about resilience across all the levels in a computing system; to use that common language in order to discuss existing knowledge on resilience across the various hardware and software layers of an exascale system; and then to build on those results, examining potential solutions from both a hardware and software perspective and focusing on a combined approach.

The workshop brought together participants with expertise in applications, system software, and hardware; they came from industry, government, and academia; and their interests ranged from theory to implementation. The combination allowed broad and comprehensive discussions and led to this document, which summarizes and builds on those discussions.

The document is organized as follows. In the rest of the introduction, we define resilience and describe the problem of resilience in the exascale era. In Section 2, we present a consistent framework and terms used in the rest of the document. Sections 3 and 4 describe the sources and rates for hardware and software errors. Section 5 examines classes of software capability in preventing, detecting, and recovering from errors. Section 6 takes a systemwide view and describes possible ways of achieving resilience. Section 7 presents possible scenarios and how to handle failures. Section 8 provides suggested actions.

1.1 The Problem of Resilience at Exascale

DOE and other agencies are engaged in an effort to enable exascale supercomputing performance early in the next decade. Extreme-scale computing is essential for progress in many scientific and engineering areas and for national security. However, progress from current top high-performance computing (HPC) systems (at tens of petaflops peak performance and roughly 1 PF sustained performance) to systems 1,000 times more powerful will encounter obstacles. One of the main roadblocks to exascale is the likelihood of much higher error rates, resulting in systems that fail frequently and make little progress in computations or in systems that may return erroneous results. Although such systems might achieve high nominal performance, they would be useless.

Higher error rates will be due to a confluence of many factors:

- Hardware failures are expected to be more frequent (discussed in more detail in Section 3). Errors undetected by hardware may be frequent enough to affect many computations.
- As hardware becomes more complex (heterogeneous cores, deep memory hierarchies, complex topologies, etc.), software will become more complex and hence more error-prone. Failure and energy management also add complexity. 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 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, tolerate asynchrony, and tolerate failures results in more complex algorithms. The more complex libraries and application codes are more error-prone. Software error rates are discussed in Section 4 in more detail.

1.2 Applicable Technologies

The solution to the problem of resilience at exascale will require a synergistic use of multiple hardware and software technologies.

Avoidance: for reducing the occurrence of errors

Detection: for detecting errors as soon as possible after their occurrence

Containment: for limiting the impact of errors

Recovery: for overcoming detected errors

Diagnosis: for identifying the root cause of a detected error

Repair: for repairing or replacing failed components

We discuss potential hardware approaches in Section 3 and potential software solutions to resilience in Section 5.

1.3 The Solution Domain

The current approach to resilience assumes that silent errors are rare and can be ignored. Applications checkpoint periodically; when an error is detected, system components are either restored to a consistent state or restarted; applications are restarted from the latest checkpoint. We divide the set of possible solutions for resilience at exascale into three categories.

Base Option: Use the same approach as today. This would require the least effort in porting current applications but may have a cost in terms of added hardware and added power consumption. We discuss in Section 7.1 what improvements are needed in hardware and system software in order to carry this approach into the exascale range, and we consider what costs will be incurred.

System Option: Use a combination of hardware and system software to handle resiliency in a manner that is transparent to the application developer. This approach will require no change in application codes and is therefore equivalent to the base option from the viewpoint of application developers. The relative cost of hardware changes vs. system software changes will dictate preferences between the base option and the system option. We discuss this option in Section 7.2.

Application Option: Require application developers to handle resilience as part of their application code. The approach is more invasive from the viewpoint of application developers but may reduce the cost of exascale platforms and their energy consumption. We further subdivide this option into two suboptions.

- **Application-Level Error Detection** Application code is responsible for error detection; recovery is done, as today, by restarting from a checkpoint. That is, the only added burden on application developers is to provide a checkpoint validation routine. We discuss this option in Section 7.3.
- **Application-Level Error Correction** Application code is also written so as to avoid the need for global checkpoint and global restart, thus possibly reducing the overheads entailed by this approach. We discuss this option in Section 5.6.4.

We find that some technologies are essential no matter which approach is chosen. For example, it is essential to reduce the frequency of system crashes and to reduce the time to recover from system crashes. Other technologies are "no brainers" in that they improve the resilience of systems with little added cost. This is true, for example, of failure prediction and avoidance, as discussed in Section 5.1.

The three options are not mutually exclusive. The system option will still require adequate hardware support, and the application option will require adequate hardware and system software support. Design choices will need to consider the maturity of various technologies and the relative cost of the different choices of higher platform acquisition cost, higher power consumption, or higher cost for application code development and porting. The balance may change over time and may well not be the same for today's 10 PF machines as for a 100 PF system or an exascale system. To be able to make the tradeoffs requires understanding the costs based on the expected and possible capabilities at each layer. Thus, we discuss in Section 8 the commonality between these options, pointing out technologies that are clearly needed no matter what path is taken, and the research, observations, and experiments that can help us choose the appropriate path.

1.4 Previous Reports

Our work leverages several recent reports on resilience.

A DARPA white paper on system resilience at extreme scale was issued in 2008 [40]. It points out that current high-end system wastes 20% of its computing capacity on failure and recovery. The white paper outlines possible evolutionary and revolutionary research with the goal of bringing this number down to 2%.

A DOE/DOD report issued in 2009 [32] identifies resilience as a major emerging issue for high-end computing (HEC) that requires new approaches. It calls for a national effort and proposes research in five thrust areas: theoretical foundations, enabling infrastructure, fault prediction and detection, monitoring and control, and end-to-end data integrity. This report considers resilience to be "concerned with reliability of information in lieu of, or even at the expense of, reliability of the system."

A DOE/DOD report issued in 2012 [37] identifies six high priorities: fault characterization, detection, fault-tolerant algorithms, fault-tolerant programming models, fault-tolerant system services, and tools.

The Computing Community Consortium (CCC) organized in 2011 a Cross-Layer Reliability Visioning Study [6]. This study, while not focused on high-performance computing, makes many relevant points. It suggests a research and education program with eight components: repairable hardware architectures; cross-layer information sharing; multilayer error filtering; multilayer tradeoffs for error handling; differential reliability; techniques, theories, and platforms that are scalable and adaptive to a wide range of error rates and error types; graceful degradation; and embedding of reliability and immunologics engineering into electrical engineering, computer engineering, and computer science curricula.

A recent DOE workshop [56] focused on resilience from the perspective of DOE, with the following goals: (1) describe the required HPC resilience for critical DOE mission needs; (2) detail what HPC resilience research is already being done at the DOE national laboratories and is expected to be done by industry or other groups; (3) determine what fault management research is a priority for DOEs Office of Science and NNSA over the next five years; and (4) develop a roadmap for getting the necessary research accomplished We included in this list only recent reports. We note, however,

that research on fault-tolerant computing is as old as computers are. Frequent failures were a major problem in the earliest computers: ENIAC had an MTTF of two days [148]. Major advances in this area occurred in the 1950s and 1960, for example, in the context of digital telephone switches [36] and mainframes [143]. Bibliographical research must be an important component of a research program in resilience.

2 Taxonomy of Terms

Editor: Jon Stearley
Contributors: Rinku Gupta

"Clear language engenders clear thought." – Richard Mitchell, *The Underground Grammarian*

The absence of agreed upon definitions and metrics for supercomputer reliability, availability, and serviceability has, in the past, obscured meaningful discussion of the issues involved and has hindered their solution [145]. In order to avoid similar confusion, several of the participants were asked to address terminology a few weeks ahead of the workshop. In subsequent email discussions, it was proposed that the taxonomy of Aviženis [9] be considered as a widely used standard to build upon (having roughly 2,000 citations). A summary sheet was prepared and then discussed on the first day of the workshop; it is appended to this report. Several omissions and corrections were identified during the discussion, but there was wide consensus that the taxonomy was sufficiently clear and complete to be used as a basis of our discussion of addressing failures at exascale. Additional revisions were made during the week based on subsequent discussions, and the resulting definitions appear below.

2.1 Dependability

The definitions in this section are based almost entirely on [9].

System: an entity that interacts with other entities

Component/subsystem: a system that is part of a larger system

Atomic component: the point at which system/component recursion stops, by desire or discernability

Functional specification: description of system functionality and performance, defining the threshold between a *correct* and an *incorrect* service (acceptable vs unacceptable)

Service: a system's externally perceived behavior

Quality of service (QoS): guarantees provided by the system on the performance and reliability of the service it provides

Behavior: what a system does to implement its function, described by a series of states

Total state: a system's computation, communication, stored information, interconnection, and physical condition

Dependability: the ability to avoid service failures that are more frequent and more severe than is acceptable

Dependence: the extent to which a system's dependability is affected by another's

Trust: accepted dependence

The terms fault, error, and failure are sometimes used synonymously, but we believe that more distinctive use, as defined in [9], is beneficial:

Fault: the cause of an error (e.g., a bug, stuck bit, alpha particle)

Error: the part of total state that *may* lead to a failure (e.g., a bad value)

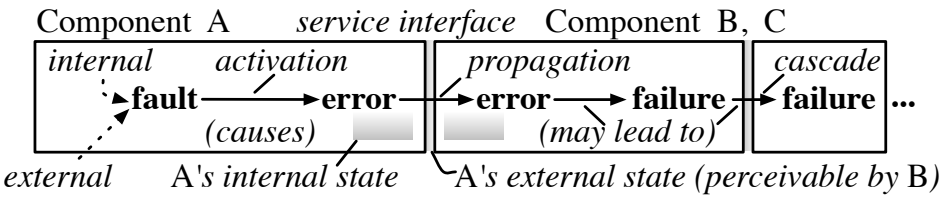


Figure 1: Error propagation and cascading failures

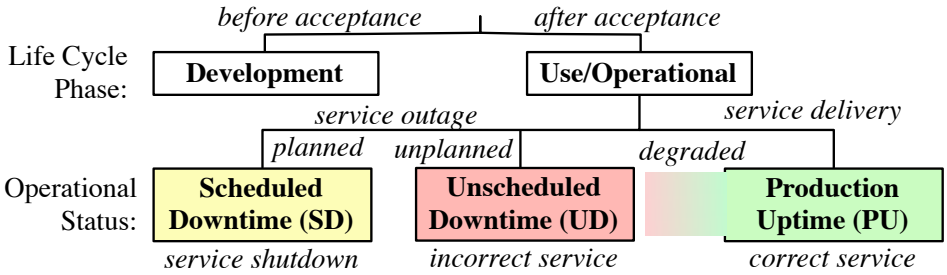


Figure 2: System's operational status

Failure: a transition to incorrect service (an event, e.g., the start of an unplanned service outage)

Degraded mode/partial failure the failure of a subset of services

Faults can be *active* or *inactive*, meaning actually causing errors or not. A fault is generally local to a single component, as distinct from errors that may propagate from component to component. Similarly, the failure of one component may lead to the failure of another (e.g., "cascading" failures), as shown in Figure 1.

For example, consider a cracked wire inside a cable. The crack is the fault, and it does not move from cable to cable. Because of the crack, a certain bit may be incorrectly flipped during transmission, resulting in an error (an incorrect bit value). The cable failed to provide correct service. The error may continue to propagate from device to device, perhaps leading to incorrect results (a failure). Or that flipped bit may have no effect on final results (no failure).

2.2 Life Cycle and Operational Status

After acceptance, a system is, at any time, in one of the operational states shown in Figure 2.

2.3 Failure Characteristics

Domain: What has failed. The failure can be involve the wrong *content* (incorrect state) or wrong *timing*—service not provided in a timely manner.

Persistence: A failed system may *halt* (fail-stop) or may exhibit an *erratic* behavior.

Detectability: A failure can be *signaled* once it is detected and a warning is generated; otherwise, it is *unsignaled*. The detection and signaling mechanism can fail, resulting in *false positives* (a false alarm) or a *false negative* (a failure that did not generate an alarm). The *precision* of a detection mechanism is the fraction of signaled failures that were actual failures, and *recall* is the fraction of failures that were detected and signaled: $\text{precision} = 1 - \text{false\_positives}/\text{signalled}$; and $\text{recall} = 1 - \text{false\_negatives}/\text{failures}$.

Consistency: A failure is *consistent* if it is perceived identically by all users; it is *inconsistent* (or Byzantine) if it is perceived differently by different users. Fail-stop errors are normally consistent, whereas erratic failures can lead to Byzantine behavior.

2.4 Fault Characteristics

Active: Fault causes an error.

Dormant: Fault does not cause an error. The dormant fault is *activated* when it causes an error.

Permanent Presence is continuous in time.

Transient Presence is temporary.

Intermittent Fault is transient and reappears.

Hard/solid: Activation is systematically reproducible.

Soft/elusive: Activation is not systematically reproducible.

The distinction between hard and soft faults is not a strict one: Faults may be due to a complex combination of internal state and external conditions that occur rarely and are difficult to reproduce; they appear as soft faults; a root cause analysis may identify the precise circumstances of the fault, enabling systematic reproduction.

2.5 Error Characteristics

Detected: indicated by error message or signal

Latent/silent: not detected

Masked: not causing a failure

Soft: due to a transient fault

2.6 Means of Dealing with Faults

Forecasting: to estimate the present number, future incidence, and likely consequences of faults

Prevention: to prevent fault occurrence

Removal: to reduce fault number and severity

Tolerance: to avoid service failures in the presence of faults

2.7 Fault Tolerance Techniques

Error detection: identify the presence of an error

Concurrent: occurs during service delivery

Preemptive: occurs during planned service outage

Recovery: prevent faults from causing failures

Error handling: eliminate errors

Rollback: revert to previous correct state (e.g., checkpoint, retry)

Rollforward: move forward to a new correct state

Compensation: correct the error (e.g., via redundancy)

Fault handling: prevents faults from reactivating

Diagnosis: identifies fault location and type

Isolation: excludes from interaction with other components

Reconfiguration: replaces component or moves work elsewhere

Reinitialization: performs a pristine reset of state (e.g., reboot)

Error detection identifies the presence of an error but does not necessarily identify which part of the system state is incorrect, and what fault caused this error. By definition, every fault causes an error. Almost always, the fault is detected by detecting the error this fault caused. Therefore, "fault detection" and "error detection" are often used synonymously.

"Full diagnosis" identifies the root cause of a failure—the original fault or faults that caused this failure; on the other hand,"partial diagnosis" traces back the error to previous events in the causality chain but does not necessarily identify the original fault. Thus, failure of a software system may be traced back to a hardware error, such as a bit flip, without identifying the fault that caused this bit flip.

2.8 Metrics

If you can not measure it, you can not improve it. – Kelvin

We cannot optimize resilience without measuring it. We discuss here two metrics: workload and availability.

2.9 Workload

A key metric is the ratio of the ideal time to solution on an ideal, fault-free system (T_{solve}) to the actual runtime in a real system ($T_{wallclock}$):

Workload Efficiency $= T_{solve}/T_{wallclock}$.

In the general case, where the system is running a mix of jobs, we can define workload efficiency as the ratio between the ideal time to solution for this job mix on a fault-free system and the actual running time. The difference between $T_{wallclock}$ and T_{solve} is the *overhead* associated with dealing with faults, errors, and failures, including scheduled downtime, unscheduled downtime, and the cost of detection, diagnosis, repair, compensation, and time lost because of degraded performance.

Typically, workload efficiency is measured with respect to "system faults" and includes all faults underlying applications that impact solution correctness or solution time: software bugs, hardware bugs, hardware faults, and so forth. It does not include faults such as application bugs or user errors. However, the workload efficiency does depend on the application code. For example, it depends on how frequently the user checkpoints and how efficient the checkpoint and restart code are. If failure handling will require increased user involvement in the future, then workload efficiency will increasingly depend on the user code, but the overhead due to user code that handles failures will be increasingly hard to measure.

The workload efficiency metric is an "instantaneous metric." The workload efficiency of a system will vary over time: failure rates are higher on a new system or on a system close to the end of its lifetime. Better system design and better testing procedures may reduce the time needed to stabilize a system and raise the workload efficiency faster. Therefore, it is also useful to define a total workload efficiency metric

up-time – time when the system functions as specified scheduled down-time unscheduled down-time

Figure 3: System history

that integrates workload efficiency over the lifetime of a system. The definition of such an integrated metric has to take into account that computers depreciate rapidly: a flop now is twice as valuable as a flop in two to three years; hence overhead now is twice as expensive as overhead in two to three years. Given a depreciation rate, it is easy to compute a depreciated total workload efficiency.

The definition of workload efficiency considers time as the critical system resource. If energy is the critical resource, then workload efficiency can be defined as the ratio of the energy needed to solve a problem in an ideal, fault-free system, to the energy needed in reality. Considering the impact of wasted energy is important: Some of the techniques for recovery discussed in this report could have little effect on total wall-clock time but could significantly reduce power consumption.

In practice, both time and energy are important resources, as are the acquisition cost of the system and the additional program development effort needed to handle failures. The contribution of resilience technology to the value of supercomputers can be measured by a "total factor productivity" (TFP) metric, as the ratio between the cost of inputs (acquisition price, salaries, electricity bills) and the value of outputs (scientific results) [141]). Unfortunately, it is hard to properly estimate the cost of various inputs (e.g., programming time), even harder to separate the contribution of resilience technology from the contribution of other technologies, and practically impossible to put a price on the output of supercomputers.

2.10 Availability

Availability metrics are similar in spirit but more operational in nature. For example, a system may be defined to be "down" when more than 5% of the compute nodes are down or the file system is down; downtime may be considered "unscheduled" if notification occurs less than 12 hours in advnace[105].

Consider the time series in Figure 3 of system states, where numbers indicate duration in days. We tabulate the data into sets and obtain the following statistics:

Set X	$\sum X$	$\lvert X\rvert$
Uptime periods={6, 2, 3.7, 3, 2.6}	$Uptime = 17.3$	$NumUptimes = 5$
Scheduled downtime periods={1, 1, 2}	$Scheduled\_Downtime = 4$	$NumSchedDown = 3$
Unscheduled downtime periods={.3, .4}	$Unscheduled\_Downtime = .7$	$NumInterrupts = 2$
	$Total\_Time = 22$	

The following metric is recommended as a *control* (specified) metric [11]:

- *Scheduled Availability =Total_Time - Scheduled_Downtime/Total_Time*.

In our example, *Scheduled Availability*$= (22 - 4)/22 = 81.8\%$.

The following metric is recommended as an *observed* metric:

- *Actual Availability =Uptime/Total_Time*.

In our example, *Total Availability*$= 17.3/22 = 78.6\%$

We are using *interrupt* as synonymous with detected failure, so *mean time between interrupts* (MTBI) is equal to *mean time between failures* (MTBF). In our example, *MTBI* $= TotalTime/NumInterrupts = 22/2 = 11$ days.

Similarly, if MTTI is the *mean time to interrupt*, then ***MTTI*** $= Uptime/NumInterrupts = 17.3/2 =$ 8.65 days.

The *mean time to repair* (MTTR) is the average length of a unscheduled downtime period. In our example, ***MTTR*** $= Unscheduled\_Downtime/NumInterrupts = .7/2 = .35$ days.

The *mean uptime* is the average length of an uptime period. In our example, ***MeanUptime*** $= Uptime/NumUptimes = 17.3/5 = 3.46$ days.

2.11 Subsystem

When discussing faults, errors, and failures, one must carefully identify what "system" is being referred to. In the previous example, the cable can be considered a system (of wires, solder connections, pins, etc.), the transmission network a whole can be considered a "system" (of cables, switches, network cards), and the entire collection can be considered a "system" (compute nodes, I/O nodes, network, disks, etc.).

The taxonomy [9] was developed to address both dependability and security, so the definitions are extremely broad. For example, "system" can refer not only to computing equipment but also to a hacker or group of collaborating hackers. We found it important to identify what is meant by "system" and to identify when that definition changes during the discussion, such as "full system" versus "I/O system." Some uses of "system" include applications, users, and administrators; but the majority of participants referred to "full system" as the collection of components *underlying* the application (not including the application or elements above it, such as users).

Unique acronyms can increase clarity. For example, Sandia and Los Alamos National Laboratories prepend an "S" (e.g., $SMTTI$) to metrics that apply to the full system and‘ other prefixes to identify subsystems [2, 145]. JMTTI, the *job mean time to interrupt*, is defined as ***JMTTI*** $= (Uptime \times NumJobs)/Num\ JobInterrupts$, where ***NumJobs*** is the total number of jobs run and ***NumJobInterrupts*** is the total number of jobs terminated as a result of any failure. NMTTI, *node mean time to interrupt*, is defined as ***NMTTI*** $= Uptime \times NumNodes/NumNodeFailures$, where ***NumNodes*** is the total number of nodes and ***NumNodeFailures*** is the total number of node failures.

2.12 Statistical Models

Analyses of failures and recovery algorithms assume that failures occur according to a probabilistic process that has a closed-form description. A typical assumption is that failures are independent, that is, failure intervals are independent, identically distributed random variables. This assumption is clearly false over long periods, since failures are more frequent on a new system or on a system close to the end of its expected lifetime (this leads to a so-called bathtub distribution of failures). It is not clear to what extent the assumption is valid over short time periods, since many phenomena may cause correlated failures. In particular, even if faults are independent, some faults may cause cascading failures of many components. For example, a power or cooling fault can cause the failure of a large number of nodes.

It is often assumed that between-failure intervals have an *exponential distribution*, with a cumulative distribution function (CDF) $F(t) = 1 - e^{-t/M}$, where M is the MTBI. Such a distribution is implied by the assumption that failures occur according to a Poisson process: The probability that a failure occur during a time interval depends only on the length of this interval. A *Weibull distribution*, with a CDF of $F(t) = 1 - e^{-(t/M)^k}$, can be used to model a decreasing failure rate ($k < 1$), constant failure rate ($k = 1$), or increasing failure rate ($k > 1$),

An empirical study of HPC failure data from Los Alamos National Laboratory showed a poor fit to an exponential distribution, whereas gamma or Weibull distributions with decreasing failure rates (.7-.8) fit well [135]. Surprisingly this study showed that the Weibull distribution fit better in the outer years of the observed system, while no distribution fit well in the first years. These results could interpreted as meaning

that failures in HPC systems are chaotic during the long period it takes for the system to stabilize and that the system keeps improving its reliability through its lifetime. Such an interpretation is consistent with the observation that most failures are due to software.

See http://cfdr.usenix.org for this and other data.

2.13 Resilience, Fault Tolerance, and Dependability

Unitl now, we have been using the key term "resilience" without clearly defining it. Several reports [38, 33, 112, 37] have used different definitions; and debate continues about how, or whether, resilience differs from "fault tolerance" or "dependability." Avizienis et al. [9] considered it synonymous with "fault tolerance" and defined it as a wide collection of techniques. The authors defined "dependability" as the"ability to avoid services failures that are more frequent and more severe than is acceptable." In HPC, service failure has two aspects: (1) failure to run a program or incorrect answer and (2) computation taking too long. The second criterion is quantitative and can be measured in various ways, in particular by using the workload efficiency metric defined earlier: A system fails if its workload efficiency is below a certain threshold. Accordingly, *resilience* can be defined as follows:

The collection of techniques for keeping applications running to a correct solution in a timely and efficient manner despite underlying system faults

"Correct," "timely," and "efficient" are context-dependent. In some contexts "correct" may mean "bit reproducible"; in another context, it could mean "within a rounding error"; in yet another context, we could be content with a system that frequently provides a correct solution to a problem—provided that we can efficiently verify solutions. "Timely" and "efficient" are relative rather than absolute (as in before the hurricane arrives and within our power budget). The definition of "efficient" also depends on what we consider to be the total system—for example, are programming costs included?

3 Sources and Rates of Hardware Faults and Errors

Editor: Mattan Erez
Contributors: Pradip Bose, Paul Coteus, Al Geist, Subhasish Mitra, Rob Schreiber

In this section we describe a generic HPC machine along with the various hardware errors and failures that can occur while it is executing an application. We focus on hardware aspects and do not account for any masking or handling in software. We summarize the rates at which these errors and component failures occur on current systems and then discuss models for the underlying fault mechanisms, project these models to future 11 nm technology, and recommend possible mitigation techniques and their overheads.

3.1 Generic Machine Model and Associated Errors and Failures

Figure 4 describes a generic exascale machine, patterned after the current generation of HPC machines at Argonne, Los Alamos, Lawrence Livermore, and Oak Ridge National Laboratories and similar leading supercomputing centers. Faults can occur in any part of the machine, with differing consequences. Some failures (fans, power converters) are masked by redundant hardware. Other failures (nodes) will cause an application to crash and restart from the last checkpoint with a new set of nodes but will not cause the system to crash. Some failures cause the entire system to crash and have to be rebooted. The severity of different failures can be measured by the loss of machine time they cause. The masked failure of a fan slightly increases scheduled downtime; a system crash causes the entire machine to be down for half an hour or more.

To expand on the hierarchy, we imagine that the nodes, servers, and switches of the machine are composed of *field replaceable units* (FRUs): processors, memory DIMMs, various circuit cards, power and

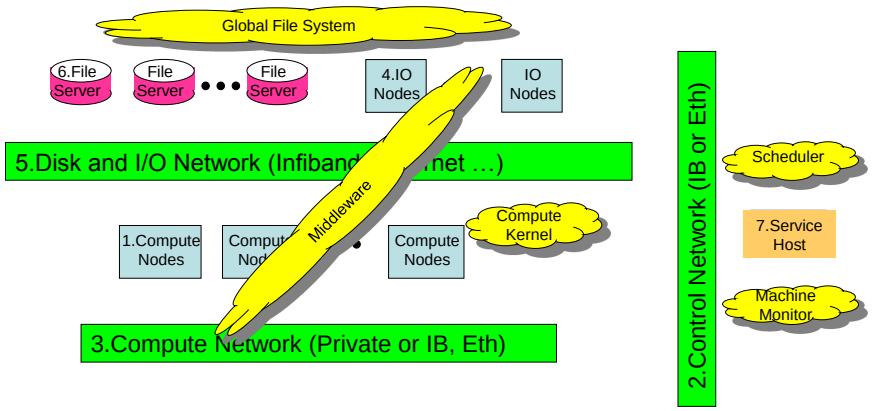


Fig 1: Generic Exascale Machine.

Fails can occur in any system 1-7 with differing consequences, 1 being the least troublesome and 7 being the most.

Figure 4: Generic exascale machine

fan modules, and the like, which are usually collected into removable and field serviceable drawers. Sets of drawers may form chassis, and multiple chassis form racks. Typically, but not always, communication is highest between FRUs on a processor node (formed of one or more processor sockets sharing coherent memory and with at least one network connection). This then is a natural fault containment region. Further, groups of nodes may share some common resource—a network adapter card, power supply, or fan module—making this group of nodes in a drawer (or chassis) a higher-level containment region. The entire rack, perhaps sharing a common resource such as a power cord and bulk AC-DC power supply, could form an even higher containment region. However, HPC applications are tightly coupled, so that errors propagate quickly across components. Software help is needed in order to avoid error propagation and transform physical fault containment units into logical error containment units.

3.2 Classification of Errors and Failures

Hardware faults can result in errors and failures that may be grouped into three categories: (1) detected and corrected by hardware (DCE), (2) detected in hardware but flagged as being uncorrectable (DUE), and (3) silent (SE). A silent fault may be masked; a *silent data corruption* (SDC) is an error caused by an unmasked silent fault. We describe these categories below and discuss the possibility that faults may lead to operating with degraded performance, efficiency, and/or fault protection capability.

Examples of DCE: (a) a detected error in an error checking and correcting (ECC) protected SRAM/DRAM array that is corrected "in place" before being passed on to a unit that consumes that piece of data and (b) a detected parity error in the processor pipeline that triggers an instruction retry mechanism, resulting in recovery of an uncorrupted, prior-architected register state and re-execution from that point. In the latter case, the recovery mechanism must ensure that leakage of potentially corrupted data to the system's memory or I/O state is prevented during the whole "detection and recovery" process. The system can be architected such that DCEs are usually transparent to the user (application) program and possibly even to supervisory system software (e.g., operating system). In some cases the supervisory system or operating system is invoked in order to help record DCE statistics in system memory for later error analysis. In such cases, the DCE is still transparent to the user application. Usually, hardware has autonomous (software-transparent) mechanisms to record DCE statistics in hardware trace (debug) arrays for later diagnostics. Note that frequent DCEs will slow the system and could, in extreme cases, cause timing errors.

Examples of DUE: (a) a double-bit error, detected during the attempted reading of a SECDED ECC-protected SRAM/DRAM array datum, that could not be corrected "in place" and (b) a detected parity error in the processor pipeline that cannot inform the on-chip recovery mechanism within a stipulated deadline, which is an architected parameter designed to ensure that known (potentially) corrupted data is not released to system memory or I/O state. Usually, all DUEs are flagged as an exception to system software by the hardware. Depending on the nature (severity) of the DUE and the capability of the system, the software should be able to handle the hardware-raised exception in a manner that enables one of the following three actions: (i) restart of the processor execution from a local or global checkpoint; (ii) application checkstop that terminates the application, without crashing the node; or (iii) system checkstop that results in a machine check (requiring "reboot") for the particular node or, in the worst case, perhaps even the whole system. In some cases it might be preferable to simply mark corrupted values as invalid, or poisoned, and allow the application itself to handle the error. An example is to use `NaN` values to prevent incorrect data from silently corrupting results, while still allowing for potential application-level masking or handling.

Examples of SDC: (a) an undetected arithmetic computation error, within an integer or floating-point data path pipeline, that makes it into architected register state (and eventually perhaps the system memory state) without triggering any error alert at the hardware level; (b) an undetected control error that results in a premature termination of an iterative loop computation that may result in a datum held in register or memory state to contain a value that is incorrect from a program-intended perspective; and (c) incorrect memory and network transfers that were not detected by the error protection mechanisms (e.g., triple-bit errors with SECDED protection). Such SDCs may eventually be detected within a self-checking application program or as a result of a triggered DUE, but such a detection could happen many thousands, millions, or billions of cycles beyond the point of the original occurrence of the SDC. Thus, a sophisticated "root cause analysis" of a DUE may later point to a an originating (causative) SDC when it comes to proper accounting statistics of various categories of errors in the hardware.

As a consequence of errors originating from hardware sources, and the associated error-handling hierarchy in hardware and/or software, the overall computing system may manifest degraded levels of quality of service as viewed by the end user. For example, if the system encounters a node failure, even if the system or application can recover from the failure, the system will operate at a degraded performance level during the period of system reconfiguration (via updates in the routing tables, etc.). Similarly, an escalated sequence of ECC memory errors may eventually result in a memory "chipkill" that reduces the amount of available system memory (before the defective memory module is replaced), thereby degrading performance. Similarly, certain other repair actions resulting from the flagging of hardware errors may reduce the capability of hardware in terms of being able to detect the full range of errors that the system was originally designed for.

3.3 Quantification of Component Errors and Failures

Table 1 shows the hardware error and failure data for 382 days of the Intrepid system at Argonne National Laboratory. This 40-rack, 557 TF Blue Gene/P system currently shows a mean time to (hardware) interrupt of 7.5 days. Thus, the total of any detected hardware failure, including compute nodes, I/O nodes, compute node interconnect, control hosts, and file servers, was roughly 1 per 7.5 days. This was extremely close to the 7-day MTBF predicted for the machine back in 2006, well before installation, and shows that one can accurately predict hardware failure rates on a large system before its construction. We point out that this agreement was obtained only after wholesale replacement of two minor but problematic elements of the machine—the 10 Gb/s optical transceiver modules on the I/O links and early versions of the bulk power supply modules. This experience is consistent with the LANL study discussed in Section 2.12: In the beginning there is chaos; statistical regularity takes over when the system matures.

Table 1: Error and failure rates for the Intrepid Blue Gene/P system.

Detected Uncorrectable	**Predicted % Fails per Repair Period**	**Intrepid (ANL 40 racks) Observed failures per Repair Period**	**Intrepid without I/O Failures per Repair Period**
Compute cards	90%	0.648	0.648
Node boards	5%	0.137	0.137
I/O cards	2%	0.785	0.000
Link cards	2%	0.020	0.020
Service cards	1%	0.098	0.098
Fans	0%	0.000	0.000
Bulk power	0%	0.000	0.000
Mid-planes	0%	0.000	0.000
Clock card	0%	0.000	0.000
		1.69	**0.90**
Detected and Corrected/Marked			
Compute Cards: (80% DRAM)	58%	2.003	2.003
Node boards	28%	0.491	0.491
IO Cards	0%	0.000	0.000
Link Cards	2%	0.059	0.059
Service cards	1%	0.196	0.196
Fans	4%	0.079	0.079
Bulk Power	6%	0.884	0.884
Mid-planes	0%	0.000	0.000
Clock card	0%	0.000	0.000
		3.71	**3.71**

Not all failures have the same impact. A node board failure affects all 32 compute cards sitting on it (each card contains a 4-core processor and attached memory). The failure of an I/O card can affect all compute cards on the board containing the I/O card. The failure of a link card affects an entire partition or set of nodes that are assigned to a running job.

3.4 Hardware Fault, Error, and Failure Models and Projections

To project the hardware error and failure rates expected in an exascale machine, one must understand the root cause of these events. While reasonably good models exist for some faults in some components, important gaps remain in the projections we will be able to make. We summarize our best-effort models below.

3.4.1 Compute Node Soft Errors

Soft errors in the compute node (processor and memory only; network, power, and cooling are discussed later in this section) are most often a result of events that are entirely external to the system and cannot be

replicated. By far the most significant source of transient faults is energetic particles that interact with the silicon substrate and either flip the state of a storage element or disrupt the operation of a combinational logic circuit. The two common sources of particle strike faults are alpha particles that originate within the package and high-energy neutrons. Alpha particles are charged and may directly create electron-hole pairs. When a high-energy neutron interacts with the silicon die, it creates a stream of secondary charged particles. These charged particles then further interact with the semi-conductor material, freeing electron-hole pairs. If the charged particle creates the electron-hole pairs within the active region of a transistor, a current pulse is formed. This current pulse can directly change the state of a storage device or can manifest as a wrong value at the end of a combinational logic chain.

To analyze the impact a particle strike has on a compute node, we model the effect on each node component separately, namely, SRAM, latches, combinational logic, DRAM, and NVRAM. We then determine a rough estimate for the number of units of each component within the node. We use this estimate to provide a rough order-of-magnitude fault rate for the compute node. We also briefly mention how such faults are handled in processors today, and we discuss how advances in process technology are expected to affect these soft faults. We make projections for the impact of particle-strike soft errors on a future 11 nm node, as well as present an estimate of the overhead/error-rate tradeoffs at the hardware level. The estimates are based on the models below and on some assumptions about the components of a node, as shown in Table 2. First, however, we give a few important caveats about the models and projections.

- The numbers summarized in Table 2 do not include errors due to hard faults or to transient faults other than particle strikes. We expect those to be a significant contributor to software-visible errors and failures.
- We do not have access to good models for the susceptibility of near-threshold circuits and do not consider such designs.
- We give only a rough, order-of-magnitude (at best) estimate; many important factors remain unknown with respect to a 11 nm technology node.

We expect that, over the next few years, ongoing research at microelectronic companies, research labs, and academia will provide more accurate estimates.

Table 2: Summary of assumptions on the components of a 45 nm node and estimates of scaling to 11 nm.

	45 nm	**11 nm**
Cores	8	128
Scattered latches per core	200,000	200,000
Scattered latches out of cores	$\frac{\sqrt{n_{cores}} \times 1.25}{n_{cores}} = 0.44$	$\frac{\sqrt{n_{cores}} \times 1.25}{n_{cores}} = 0.11$
FIT per latch	10^{-1}	10^{-1}
Arrays per core (MB)	1	1
FIT per SRAM cell	10^{-4}	10^{-4}
Logic FIT / latch FIT	0.1–0.5	0.1–0.5
DRAM FIT (per node)	50	50

SRAM. Large SRAM arrays dominate the raw particle-strike fault rate of a processor silicon die. When a particle strike releases charge within an active region of a transistor in an SRAM cell, the charge collected may exceed the charge required to change the value stored in the cell, causing a *single event upset* (SEU).

Table 3: Summary of per-processor particle-strike soft error characteristics within a compute node (sea level, equator). Note that other sources of transient faults cannot be ignored.

	Array Interleaving and SECDED (Baseline)					
	DCE [FIT]		DUE [FIT]		**SE [FIT]**	
	45 nm	11 nm	45 nm	11 nm	45 nm	11 nm
Arrays	5000	100000	50	20000	1	1000
Scattered latches	200	4000	N/A	N/A	20	400
Combinational logic	20	400	N/A	N/A	0	4
DRAM	50	50	0.5	0.5	0.005	0.005
Total	**1000–5000**	**100000**	**10–100**	**5000–20000**	**10–50**	**500–5000**
	Array Interleaving and ¿SECDED (11nm overhead: $\sim 1\%$ area and $< 5\%$ power)					
	DCE [FIT]		DUE [FIT]		**SE [FIT]**	
	45 nm	11 nm	45 nm	11 nm	45 nm	11 nm
Arrays	5000	100000	50	1000	1	5
Scattered latches	200	4000	N/A	N/A	20	400
Combinational logic	20	400	N/A	N/A	0.2	5
DRAM	50	50	0.5	0.5	0.005	0.005
Total	**1500–6500**	**100000**	**10–50**	**500–5000**	**10–50**	**100–500**
	Array Interleaving and ¿SECDED + latch parity (45 nm overhead $\sim 10\%$; 11nm overhead: $\sim 20\%$ area and $\sim 25\%$ power)					
	DCE [FIT]		DUE [FIT]		**SE [FIT]**	
	45 nm	11 nm	45 nm	11 nm	45 nm	11 nm
Arrays	5000	100000	50	1000	1	5
Scattered latches	200	4000	20	400	0.01	0.5
Combinational logic	20	400	N/A	N/A	0.2	5
DRAM	0	0	0.1	0.0	0.100	0.001
Total	**1500–6500**	**100000**	**25–100**	**2000–10000**	**1**	**5–20**

An SEU may impact a single SRAM cell or may change the values of multiple adjacent cells. Such *multicell upsets* (MCUs) are also called *burst errors*. A reasonable ballpark figure for SRAM particle-strike upset rate is 1 upset every 10^7 hours for 1 Mb of capacity, which is a rate of 10^{-4} FIT/bit [139]. Our best estimates indicate that the SEU rate for SRAM will remain roughly constant as technology scales. While many complex phenomena impact susceptibility, the current roadmap of changes to devices, operating voltage, and scale do not point to extreme changes in susceptibility. What is expected to change is the distribution of MCUs, with a single upset more likely to affect a larger number of cells at smaller scales.

Because the raw FIT/chip from SRAM is high (estimated at roughly 0.5 upsets per year per chip, or multiple upsets an hour in a large-scale HPC system), large arrays are protected with error detection and error correction capabilities. An approach in use today is a combination of physical word interleaving coupled with an error detection code or with ECC mechanisms. Given the distribution of MCUs today, 4-way interleaving with SECDED capabilities per array line is sufficient. Stronger capabilities will likely be needed in the future, but their energy and area overhead are expected to be low (see Table 3). Note that

our estimates assume that 4-bit or larger bursts increase from 1% of all SEUs to 10% or higher between 45 nm and 11 nm technology and that the rate of bursts of 8 bits or larger increases from 0.01% of all SEUs to 1% of all SEUs [81].

Note that alternative storage technology with much lower particle-strike error rates is possible. Some current processors use embedded DRAM for large arrays, and future processors may use on-chip arrays of nonvolatile storage. Embedded DRAM has a 100 times or more lower error rate than does SRAM. Nonvolatile storage cells are immune to particle strikes but do display some soft-error fault mechanisms (see discussion below).

Latches. The error mechanisms and trends for latches are similar to those of SRAM, and the per-latch SEU rate is expected to remain roughly 10^{-4}–10^{-3} FIT/bit [35]. Given the smaller number of latch cells in a processor today compared with SRAM cells, the overall contribution to error rate of latches is much smaller as well. Future processors will contain a much larger number of latch cells, and protection may be necessary. The protection mechanisms and overheads of latches depend on how the latch is used. Some latches are organized in arrays, like SRAM arrays, while other latches are scattered within logic blocks. Array latches can be protected with interleaving and ECC, although such latches are often accessed with finer granularity than large SRAM arrays, which increases the relative overhead of protection. We include this extra cost in Table 3 and project a higher power overhead than area overhead for protecting arrays in order to account for the added protection of latch arrays that may be needed in future processors.

"Scattered latches" are more difficult to protect, on the one had, because the overhead of interleaving and ECC is exorbitant without the regularity of an array. On the other hand, an error in a scattered latch is often masked by the natural operation of the circuit it is part of. Various estimates exist for the derating factor that should be applied for this natural masking, typically ranging from 90 to 95%. The masking rate may depend on the application and also on the architecture, with more streamlined architecture potentially having a lower rate of masked latch errors. If needed, scattered latches can be protected against particle-strike-induced upsets. The two main techniques that can be applied are hardened latches or a combination of parity prediction from logic with parity checking on a collection of latch bits. Both techniques can be effective but potentially have high overhead if a large fraction of latches must be protected. We show the impact of this overhead in Table 3.

Combinational Logic. The trends we expect for particle-strike-induced soft errors in combinational logic are again consistent with those for SRAM and latches. The raw SEU rate associated with combinational logic can reasonably be estimated at 0.1–0.5 FIT for every 1 FIT contributed by scattered latches within logic blocks [57]. Note that this is the raw upset rate and does not account for logical masking effects. Similar to latches, even if an output of a logic gate is changed, this change is highly unlikely to impact the final result of the circuit. Because the output of a combinational logic path is always a latch, the overall masking rate of combinational logic upsets is most likely close to 99%.

Note that the raw upset rate quoted above already accounts for electrical masking, which results from the SEU current pulse being attenuated as it passes through multiple gates, and for timing or latch masking, which results from the output of the combinational logic being observed for only a fraction of the cycle. As with scattered latches, we expect the raw fault rate to stay roughly constant as technology scales, and application and architecture may impact masking rates. The parity-prediction mechanism that can be used to detect errors in latches, will also detect a large fraction of logic errors. Other techniques for detecting combinational logic soft faults at the hardware level include those based on arithmetic coding [122, 94, 8, 95] and replication [7, 140, 133]. Moreover, electrical masking can be increased by using less area and power-efficient gate designs [72, 99].

DRAM. As a rule, DRAM exhibits a fixed rate of particle-strike soft errors per DRAM die, regardless of technology. This rate is roughly 10–20 FIT/device, and a significant fraction affects multiple bits and entire rows, columns, or banks of the DRAM device [144]. Many DRAM devices are required for the capacity of each node. Recent studies have shown that the error rate of DRAM is far higher than the particle-strike soft errors, indicating that hard faults in either the peripheral or signaling circuits are the main cause of problems [136, 79, 144].

Regardless of the fault mechanism, DRAM is protected with ECC, with large-scale systems typically supporting some form of chipkill-level ECC, which is effective against hard errors as well. We expect that even if new ECC schemes are needed in the future, their overhead will overall be similar to the overhead observed today for most applications.

NVRAM. We cover several technologies: NAND Flash, spin-transfer torque memory, phase-change memory, and resistive memories such as memristor.

NAND Flash is vulnerable to soft errors. The FIT rate per bit is growing with process shrinks. Currently it is 10^{-5} FITs. It was 10^{-8} FITs at the 100 nm technology node. ECC is needed and used to cope with this rate, which exceeds that of DRAM. NAND Flash wears out after approximately 10^6 rewrite cycles. Many architectural techniques are used to spread the load across the cells of a chip, a technique called wear leveling. Wear-out is not a major issue in consumer storage devices such as media cards. It may be an issue in solid-state disks, but it is clearly manageable there. As main memory and cache, Flash is unsuitable for this and other reasons.

Spin-Transfer Torque (STT), the leading magnetoresistive random-access memory (MRAM) technology, is under development by Toshiba and Hynix, which have made prototypes at 30 nm. Samsung has made a device at 17 nm. It is dense ($6F^2$ feature size). Speed and energy cost are good. Chips of 1 Gb are under development and may reach the market in 2014. Wear-out does not appear to be a concern for STT. It also seems that STT bits cannot be flipped by particle strikes. Thermal noise seems to cause something similar to soft errors—errors due to external stimuli, not internal imperfections. A FIT rate of 10^{-10} FITs per bit has been reported. Hard errors are an issue. It is said that "imperfections in the fabrication process greatly affect the reliability of data in STT-MRAM. Process variability causes variation in the tunneling oxide thickness and cross-section area, which affects both the static and dynamic behaviors of magnetic tunnel junctions, resulting in cell errors" [82]. Appropriate responses could include testing and map-around for bad cells, spare cells, and ECC. (These comments likely apply to all the memory technologies we consider.)

Phase-Change Memory (PCM) is resistive memory in which the state of a chalcogenide glass is changed between crystalline and amorphous by heating and either slow or fast cooling. The resulting change in the electrical resistance determines the state. Multilevel cells are possible with perhaps two bits per cell, but possibly fewer (as when three resistance levels are used). Micron is marketing 45 nm PCM for consumer applications today. PCM has better endurance than Flash, but it may wear out after as few as 10^6 up to a high estimate of 10^9 cycles because of the physical stresses of repeated heating and cooling. It appears to be invulnerable to particle-induced soft errors. The resistance of the PCM cell changes with time. Thermal disturbance due to the heating required for reset of a nearby cell is a chief cause of resistance drift, and this limits cell density. The decay of the stored data is similar to the charge leakage in the DRAM capacitor and, like it, may cause errors. Some combination of refresh and ECC can cope with drift. Because of the necessity for refresh to arrest drift, it is not clear that PCM is as nonvolatile as necessary for use in offline storage. The rate of required refresh will depend on the degree to which storage density is boosted by using multilevel cells; the tighter the level spacing, the more frequently the cell must be refreshed. In particular, a one-bit cell with only two levels would have no drift problem. There is thus a complex design space in which density, the cost of mitigating the resistance drift, the data retention time, and the error rate are in competition. Optimization of the PCM cell and its required refresh and error correction architecture is an

area of ongoing research.

The *memristor* is a new technology under development at HP and Hynix and in other laboratories and companies. A memristor stores information in the resistance of a cell (like PCM), that resistance being a function of the past flux (the integral of current) that has passed through (not by heating, as in PCM). HP and Hynix have explored memristive systems in which a metal oxide (often titanium oxide but recently also oxides of tantalum, zirconium, and hafnium) sandwiched between electrodes is electrochemically changed by the passage of current. Memristors are two terminal devices. Like other resistive random-access memory, memristors appear invulnerable to particle-strike soft errors, but a nonrecurring or transient error mechanism in memristors may exist. Recent experimental studies and models show a tendency to fail to remain in the lowest resistance state, randomly, with a probability that is strongly temperature related. At 150 degrees Celsius, half of all cells may fail if left unchecked for ten days (see,for example, [52, 155]). It is not clear whether these errors are due to cell deficiencies and can be reduced by mapping out bad cells or are totally random and need to be handled by ECC and scrubbing, or both. Memristors can wear out, but the wear-out mechanisms are not as clearly understood as they are for PCM. The ultimate durability of memristors is still to be determined. In new work on tantalum oxide memristors, endurance of over 10 billion cycles has been demonstrated [152]. Hard error vulnerabilities appear to be due to wear-out, manufacturing issues, and interface/ communication issues and may be comparable to those of PCM.

Nonvolatile Memory as a Resilience Enhancer. Nonvolatile memory (NVM) is often less vulnerable to soft error due to cosmic rays than is DRAM, but this is almost totally irrelevant to our discussion (since other errors predominate, and NVM has its own sources of errors.) Thus, replacing DRAM with NVM will not, in and of itself, enhance resilience. Checkpointing, normally at the application level, is the current default for preserving the state of an ongoing computation in order to protect it from a subsequent failure. Because of the growing size of application state and the failure of disk-based file systems to provide proportionally growing bandwidth, checkpointing to shared disk is not seen as a sustainable approach at exascale. We expect that on-node nonvolatile memory will appear, for many reasons. One reason is to serve as fast checkpoint storage, since write bandwidth will be superior to disk. In order to cope with node hardware failure, the checkpoint NVM may be "two-tailed (capable of being read by a service node or another compute node following node failure). Alternatively, the checkpoints may need to be delocalized, stored on a buddy node, or made recoverable by another scheme. DRAM can also be used for delocalized checkpoints. It will survive node failures but not global power failures. Since such DRAM will be on standby mode most of the time, there is no significant difference in power consumption.

NVM may serve other resilience functions, in part simply by providing enough memory to do more or as the top level in a hierarchy of nonvolatile storage components. For example, it can be used for logging messages, in order to support local, uncoordinated checkpointing, or for holding file system caches.

3.4.2 Compute Node Hard Errors and Failures

While we could provide rough quantitative projections of particle-strike-induced soft error rates, we cannot ignore possible failures and errors (detected and undetected) due to hard faults. Because of the complexity of designing and efficiently operating future processors, some failures and errors may be intermittent and manifest only with certain environmental conditions or specific execution characteristics. Major concerns include increased early-life failure rate, permanent and intermittent faults associated with device degradation, and increased storage element error rates because of low-voltage operation. Quantitative data on how such hard fault sources will evolve over technology generations is difficult to predict. But the effects can be enormous. We briefly discuss the issues below.

Early-Life Failures (Infant Mortality). Burn-in for screening early-life failures is becoming increasingly challenging [17, 24, 84, 113, 149]. Major challenges include power dissipation, cost, and possibly reduced effectiveness and coverage of the burn-in test techniques. Burn-in alternatives, for example, Iddq testing (measuring the supply current, or Idd, in the quiescent state) and very low voltage testing [**?**, 55, 68, **?**], are also experiencing limitations, including high leakage, process variations, and reduced voltage margins. At a highly scaled technology node with minimal reliance of burn-in, the effects of early-life failures can be significant: on the order of several thousands of defective parts per million. Such a high rate of failures is roughly equivalent to adding 10^3–10^4 FIT to the node failure rate. More aggressive on-line techniques for detecting these failures may become necessary.

Device Degradation (Aging). Device degradation induced by degradation mechanisms such as *bias temperature instability* (BTI) [3, 123, 158], *hot-carrier injection*, *time-dependent dielectric breakdown*, or metal electromigration is becoming important. While design margins (guard bands) are being squeezed to achieve higher energy efficiency, expanded design margins are required to cope with aging. Hence, traditional speed or voltage margins to overcome degradation may become too expensive. Some projections predict that beyond the 14 nm technology node, guard bands due to BTI degradation may grow to 20% or more, degrading efficiency and performance by a similar amount. Such guard bands are highly dependent on the workload, and quantitative projections can be highly pessimistic for worst-case workloads. Moreover, for near-threshold voltages of operation, a huge dilemma arises: while low-voltage operation can reduce the amount of aging, high-voltage turbo modes of operation or fast execution followed by low-voltage operation for energy efficiency can significantly exacerbate this aging effect. Techniques that dynamically adjust guard bands and improve performance and efficiency have been suggested, but their impact on intermittent failures and errors has not been fully evaluated. Here, the difference between exascale and commodity small-scale systems is vast, because of the scale multiplier of base rates and the impact of large variances on tightly coupled systems.

Low-Voltage Storage-Element Stability. As supply voltage is reduced in order to improve energy efficiency and reduce power consumption, maintaining the integrity of storage elements, including latches, flip-flops, and SRAM cells, is challenging. For example, Vcc_{min}-related errors can induce so-called goldilocks failures [111]—failures that appear hard but are, in fact, caused by phenomena typically associated with soft failures. Such failures are expected to become more problematic with increasingly complex circuilts and lower voltage supplies, affecting other circuit structures besides SRAM. At present, the only viable way to deal with Vcc_{min} errors in sequential elements is to rely on (expensive) circuit-design techniques or resort to high-voltage operation, resulting in poor energy efficiency.

Possible Mitigation Techniques. Understanding the effects of such failures is not enough. The question is, how do we mitigate them, especially for silent errors that may lead to silent data corruption? Techniques in the literature that can be useful include (1) on-line self-test and diagnostics, (2) concurrent error detection techniques (similar to soft errors), (3) adaptive self-tuning and on-line optimization, and (4) on-line self-repair. However, these techniques are generally not supported extensively for existing processors. If the U.S. Department of Energy has to rely on COTs components, chances of all these techniques being supported get even lower. That brings up the question, what hardware and software support is required for future exascale systems?

3.4.3 Network

The transport layer of the network, whether electrical or optical, can be instrumented for error detection and correction with quantifiable cost. Thus, for example, on Blue Gene/Q, a combination of CRC, Reed-

Solomon codes, and Hamming codes, along with a retry mechanism for detected but uncorrected errors, reduces the possibility of an error escape to 10^{50} [42]. Thus, network transport errors are containable. Network logic, on the other hand, comprises SRAM, latches, and logic, as described above, with their failure modes and correction techniques. Errors that result in data sent to a wrong destination are potentially the most damaging but may be mitigated with hardware or software techniques that use knowledge of the desired and actual recipient to trap errors before data corruption occurs. Networks that support superior error detection and correction, with tailored mechanisms to ensure correct delivery, will surely be a part of exascale systems.

3.4.4 I/O

The increased density of disks results in increased error rates, including an increase in undetected disk errors—those that are not detected by current techniques (RAID 6 included). Various techniques are available for detecting such errors and correcting them—mostly in the form of added redundancy [65]. In addition, disk failure rates are often higher than the nominal MTBF would indicate, with 2%–4% yearly failure rate common [134]; one parity block (RAID 5) is not sufficient, since the probability of two disk failures within the same group is too high. The IBM GPFS system implements in software a RAID 6 scheme (two parity blocks) that can overcome two disk failures [43].

While these techniques can practically eliminate the risk of data loss, they come at a cost: the disk storage system of a large supercomputer will have continuous I/O background activity due to RAID reconstruction after disk failure. The problem is worsened by the increasing gap between disk capacity and disk bandwidth, which results in increasing reconstruction time—or the need to spread reconstruction across more disks. This background activity will reduce the effective I/O bandwidth and cause significant I/O performance jitter.

3.5 Commercial Trends

The technology analysis in this section provides insight into the cost of producing components with acceptably low failure rates; it does not tell us what will be the price of processors that incorporate these technologies. While predicting component prices a decade ahead may be infeasible, we point out that market trends are not favorable. High levels of resilience are important for high-end servers, such as mainframes or RISC/Unix servers (Sun, Power, Itanium). For many other markets (mobile, clouds) vendors are likely to accept lower reliability in order to achieve lower cost and lower energy consumption. Unfortunately, the market for high-end servers is currently shrinking; the decline is particularly sharp for high-end RISC Unix servers. While some of this decline may be attributed to the current state of the economy, this sector clearly is an increasingly small fraction of the IT industry. Furthermore, this sector is likely to be less price sensitive than other sectors. Buyers of mainframes or high-end Unix servers have been willing to accept large markups on price per performance, in order to achieve higher reliability levels. These two trends are likely to lead to an increasing cost differential between low-reliability components and high-reliability components. Systems built with high-end RISC processors (Sparc64, Power7) are already sparse in the Top500 list. In addition, low-power components may not be available with high reliability.

3.6 Shielding

The impact of particle strike can be reduced by shielding (an area where DOE has significant expertise). The atmosphere is a natural shield, with higher locations suffering from higher strike rates; a computer at sea level will fail less frequently than one at a high-altitude location. Natural or artificial shielding can further reduce the neutron flux. For example, 2 meters of concrete will reduce the impact of 10 Mev neutron

radiation by three orders of magnitude [137]; less energetic neutrons are attenuated much more. On the other hand, neutrons with energies above 10 MeV carry a very small fraction of the total energy of cosmic-ray neutrons [75]. Hence, the cheapest way of avoiding the effect of cosmic radiation-induced errors may be to locate future exascale systems in abandoned SSD tunnels or repurposed atomic shelters.

4 Sources and Rates of Software Faults and Errors

Editor: Saurabh Bagchi;
Contributors: Jon Stearley, Eric Van Hensbergen, Al Geist

A large fraction of system failures is due to software, rather than hardware. A study of major DOE supercomputers in 2004–2006 showed that about 65% of failures can be attributed to software [115], whereas a study of failures in 2012 on Intrepid, the BG/P system at Argonne National Laboratory, showed that less than 16% of job crashes were due to hardware problems [5]. Results of studies show variance, however; a study by Schroeder and Gibson in 2010 showed that failures attributable to hardware ranged from 30%–60% [135].)

Moreover, the statistics do not include failures due to application software faults. Computer centers typically keep statistics only for the failures they see themselves responsible for. With application software failures included, the fraction of failures due to software faults is likely to be much higher.

Unfortunately, failures due to software are less well tracked and characterized. While statistics may indicate which subsystem crashed (e.g., file system), they do not indicate why the file system crashed. Therefore, much of the discussion in this section is qualitative.

4.1 Classes of Software Faults

Software faults can be grouped into three categories: pure software problems, hardware problems mishanlded by software, and software causing a hardware problem.

4.1.1 Class 1: Pure software problems

Some of the software faults in the first category are "classical" correctness issues: unhandled exceptions, incorrect return values, including null objects; and incorrect control flows, such as some function not being called or called under a different condition from what was desired. Such errors are likely to be frequent in the exascale system software stack. It is well known that system software is harder to develop than application software; kernel software is harder to debug than user software; and *reactive* software, where execution is driven by asynchronous events, is harder to get right than is *transformational* software, such as scientific software, that transforms an input into an output through a long sequence of (mostly) deterministic transformations.

Large scale is worsening the frequency or impact of two other types of software errors: concurrency and performance.

Concurrency errors: Subsystems such as a parallel file system are large, concurrent applications. Concurrent code is hard to develop because programmers have difficulty comprehending the possible interactions between a large number of agents. Humans often are said to be able to conceive of concurrency only at a limited scale (roughly up to 10), much less than the scale of large supercomputers. Concurrent code also is hard to debug because of the large number of possible interleavings of actions. Debugging tools typically are designed to handle bugs caused by the interaction of only two or a few agents. Because of the large number of agents in supercomputers and their tight interaction, failures due to subtle interactions between many agents become more frequent. The problem is compounded by stringent performance requirements

that prevent the use of simple, coarse-grained synchronization. As an example, early versions of the Luster file system would occasionally corrupt the data written on files [73].

Performance errors: By "performance errors" we mean failures due to resource (time, memory, etc.) exhaustion. These manifest themselves in the form of unacceptable performance, or actual crashes, due to timeouts ("time overflow") or buffer overflows. Current programming models and programming methodologies do not provide good ways to manage the performance of large, distributed systems. Estimating the average load on different nodes is relatively easy, but understanding the tail of a distribution and evaluating the frequency of rare events are much harder. A large system is a "black swan detector": events that occur rarely on one node are much more frequent with 1,000,000 nodes. Unfortunately, humans are not good at handling the impact of "black swans" [146]. The Luster file system has suffered from multiple performance errors when deployed at large scale [138]. Some of the problems were due to a lack of clarity on the "acceptable performance behavior" of applications. Programming models do not prevent applications from bring a system down by taxing particular resources. In the case of Luster, one "Achilles heel" was a limited ability to handle metadata operations. The designers of Luster assumed that no application would open or close tens of thousands of files each second; some applications did the unthinkable.

4.1.2 Class 2: Hardware propagating up to software and software not handling it correctly

Examples of the second category are a node failure not being handled by software at other nodes (node goes down, the RAS system notices it, but the application did not take that into account) and a disk failure causing file system failure. These kinds of failure can be seen as software faults (bugs) because the software is supposed to overcome such hardware failures. In practice, however, many failures seem to be due hardware errors that were mishandled by software. One plausible reason is that testing code that handles failures is difficult. Another is that software is typically designed to handle clean, fail-stop hardware failures but will be taxed by messy, intermittent errors or other strange hardware behavior.

4.1.3 Class 3: Software creating a problem for the hardware

Incorrect firmware, for example, misbehaving thermal control firmware, can damage hardware; this can be seen a firmware fault. Software can trigger an unusual usage pattern for the hardware, causing hardware errors; this can be seen as a hardware fault. In both cases, however, the software is actually the culprit.

4.2 Severity of Software Faults

Not all software errors are equally bad. The syslog standard (RFC 3164 / RFC 5424) [61] defines eight levels of severity.

0 Emergency: system unusable

1 Alert: immediate action required

2 Critical: critical conditions

3 Error: error conditions

4 Warning: warning conditions

5 Notice: normal but significant condition

6 Informational: informational messages

7 Debug: debug-level messages

Other dimensions are important as well: In particular, one must understand the scope of an error: how the error propagates and what it affects. Errors with a local effect are much easier to handle than errors that have a global effect. In software, we want to avoid as much as possible errors that corrupt large, global system state, where recovery will involve the entire system and may take a long time.

4.3 Evolution of Failure Types and Rates at the Exascale

We expect a significant increase in software faults as we move to exascale. The software stack will become more complex as it has to handle more issues (such as power management, resilience, and heterogeneity) and has to face ever more stringent performance constraints (including memory footprint). Correctness bugs will be more numerous. The increasing scale of such systems will certainly increase the frequency of concurrency errors and of performance errors.

The problem is compounded by obstacles due to the development process for extreme-scale software. Supercomputing is a small market; the development of software for the largest systems is usually underfunded and understaffed. Furthermore, software for the largest systems is never tested at full scale before they are deployed: Vendors cannot afford to stand test systems at full scale, and full-scale testing is done on premise. As systems keep increasing in size, new software errors will surface with each new generation of systems, even if the software does not change.

5 Error Prevention, Detection, and Recovery

Editor: Eric V Van Hensbergen
Contributors: Subhasish Mitra, Jacob A. Abraham, Sriram Krishnamoorthy, Franck Cappello, Sven Leyffer, Todd Munson, Mattan Erez, Marc Snir, Sarita Adve, Saurab Bagchi

In the preceding two sections we discussed sources of errors. Error handling can be categorized under several headings.

Prevention While an error-free system is not within the realm of possibility, various techniques can reduce the occurrence of errors.

Prediction Certain patterns of behavior can indicate future errors. If future errors are predicted with high precision, then preventive actions can be used to avoid them.

Tolerance Various techniques can be used to ensure that errors do not lead to failures—even if they are not detected.

Detection If an error cannot be tolerated, then it must be detected before it can be corrected.

Containment Error handling is eased if errors are contained so that they affect only a small part of the system.

Recovery Once an error is detected, forward or backward recovery is used to bring the system back to a correct state. Recovery most often will be automated.

Diagnosis As part of error detection and recovery, or at a later time, diagnosis activities can narrow down the likely cause of an error.

Repair The recurrence of errors can be avoided by replacing components, updating software, changing configuration parameters, and so on.

We address each of these approaches in the following subsections.

5.1 Prevention

We discussed in Section 3 mechanisms for hardening hardware and avoiding hardware errors. Suitable codes can be used to detect and correct errors in memory, caches, and buses. Errors in combinatorial circuits and latches can be detected and corrected by re-executing instructions.

Such prevention mechanisms can be used selectively. For example, one could have more reliable or less reliable cores—using either different designs or different operation parameters (clock speed, voltage); one could have the ability to run cores in tandem, comparing their outputs (to the L2 shared cache) in order to detect errors; one could modify mechanisms for thread-level speculation or for transactional execution so as to allow reexecution of code blocks when an error is detected; and one could have more reliable or less reliable memory. Some of these choices (e.g., types of memory or cores) need to be made when hardware is configured. Others (e.g., voltage levels, clock speeds, or duplicate execution) can be selected dynamically.

Automatic compensation mechanisms for hardware faults sometimes lead to poor overall system performance. Examples of such scenarios can be found in prior anecdotal fault analysis of large-scale systems. Sandia's Redstorm large-scale runs were plagued by slower than expected performance due to several CPUs running at 2.0 GHz instead of 2.2 GHz. Another Sandia system, Thunderbird, experienced poor system performance due to several InfiniBand links silently degrading to 256 MB/s instead of 1 GB/s. The tightly coupled nature of supercomputers exacerbates these issues, leading to the entire system experiencing performance loss as a result of a small set of degraded components.

One proposed approach is pervasive self-test diagnostics that run before and potentially during application execution in order to ascertain the health of system components as well as the overall system [83]. Similar diagnostics are run during system bring-up and in some cases weekly as part of scheduled maintenance windows—but systemic errors and performance degradation caused by transient faults happen at a much finer granularity because of various causes, including environmental variability, certain workloads exercising components of the system in unusual ways, and human error. The more pervasive use of such diagnostics would enable a consistent performance environment from run to run, eliminating significant variability in application performance resulting from latent undiagnosed system issues.

The tradeoff here would be the overhead of running diagnostics at boot time and periodically during execution versus the possibility of performance degradation. Some of this overhead may be mitigated by a "+1" core whose operation will not significantly interfere with the actual workload running on the other cores. Finding the right balance between background monitoring, periodic health diagnostics, and other forms of online self-test will be an important aspect of co-design research on extreme-scale systems.

A complementary approach to software-based errors would be to adopt better design and testing methodologies. For example, performance errors could be avoided by adopting techniques used in the design of real-time software for avoiding overcommitment of resources. Alternatively, resource exhaustion could be avoided by the use of properly designed feedback mechanisms—a topic that will appeal to control theory.

5.2 Prediction

A failure can be prevented by predicting the faults that cause the failure and evading the failure. For example, if one can predict that a node is likely to fail, then one can prevent job failure by vacating the node and migrating its workload to another node before the failure occurs. To do so, one needs to understands which faults are most likely to cause failures; and one needs to predict the occurrence of such faults based on past observations. The prediction should be timely: it is easy, but not very useful, to predict that each piece of hardware will eventually fail. Conversely, if the prediction is too close in time to the failure, then there may not be enough time for evading the failure. Failure prediction is used successfully for a wide range of complex systems, including railroads [114], nuclear power plants [159], and aircraft engines [78]. Many different techniques can be used to forecast failures. A fairly complete survey of these techniques is

presented in [132].

Several studies suggest that failure can be predicted in HPC systems. For example, a memory device tends to show, for a given address, multiple repetitive correctable errors before showing an incorrectable error [80]. Correlations in time have also been observed between soft errors and hard errors. Another recent study [74] has observed correlation in space. The predictability of hard drive failure is at the origin of the SMART (Self-Monitoring and Reporting Tech.) technology used in many disks.

In HPC systems, the overall failure prediction workflow based on event analysis, its limitations, and needed improvements are reasonably well identified. HPC systems are producing events related to the state of their software and hardware components. Events of the same type can be clustered into groups. Event correlation analysis allows establishing stochastic propagation chains between events of the same group or/and of different groups. Stochastic propagation chains essentially contain two categories of events: precursors and critical events. When a critical event is in a propagation chain, all previous events in the chain are called precursors (precursors potentially also include critical events). In the past two years, several key results have demonstrated that recent advances in event clustering [**?**], anomaly detection, [50], event correlation [51], propagation chain construction [74], and online detection of propagation chains [49] can provide precise failure prediction. The time lag observed for the most efficient prediction approaches is consistent with the time taken by proactive actions.

Current predictors can achieve a precision of over 90%, so that preventive actions will be superfluous in only one-tenth of the cases; acting on such predictions is usually worthwhile (see analysis in Appendix B). On the other hand, the recall is still low and stays below 50% even for the most advanced prediction approaches: fault prediction can effectively double the MTBI but cannot replace other methods, by itself. The main reasons for the low recall are the lack of precursor events (some failures have no identified precursors) and the precision losses at each step of the failure prediction workflow. Thus, an identified research objective is to improve the whole failure prediction workflow to increase the failure prediction coverage from 50% to 80% or 90%.

5.3 Tolerance

For some applications, we may not need to recover from node failures at all. For example, in derivative-free parameter estimation of a complex simulation, a node failure could be ignored and treated as a simulation failure. However, not all simulation failures are the same. A graceful failure can yield partial information that could be used when determining the next experiment to perform for the optimization. Structured simulation-based optimization techniques can use this partial information to build partial interpolation models and thus become resilient to node failures. Similarly, we could use partial solutions for simulations at a looser tolerance as long as we account for the truncation error in the model and optimization.

This approach can be likened to controlling the noise in simulations [107]. For stochastic noise, model-based optimization methods have been developed that specify both a candidate point and the number of replications needed to obtain sufficient accuracy. Parallel replications at a fixed point can be used to control stochastic noise but not deterministic noise. For deterministic functions one could use nearby points and Taylor's theorem to bound the noise in the simulation. By neighborhood sampling one could reduce the noise in many settings, and these samples may already be available from previous computations of the algorithm.

An alternative approach is to use insight into the application to reduce the probability of failure by reducing the memory footprint of an application. Variable precision arithmetic can help in this approach by using bounds on the precision requirements for Newton solves to compute low-precision steps initially. Analysis tools, such as those developed for automatic differentiation and estimating computational noise, could identify blocks in the code for which higher precision would lead to improved precision in function evaluations. Based on this identification, one could restructure the computation of a function so that the least-precision

arithmetic was used in each block to obtain the required precision in the overall function evaluation. Similar ideas could be applied for gradient and Hessian evaluations. User-provided and automatically generated codes for quantities derived from function values (such as derivatives) can be significantly less precise than the underlying function. Analysis of the underlying computational graph (for example, as done by Kubota) could provide insight into reformulations of the derived code that yield both function and derived values to specified precision.

5.4 Detection

Mechanisms for detecting hardware errors, such as ECC and circuit-level redundancy, are briefly described in Section 3. Here we focus on software-driven detection and application-level detection.

5.4.1 Software-Driven Detection of Hardware Errors

Conventional hardware detectors either have relied on expensive redundancy-based solutions or have focused on specific fault models and hardware components. Recently, considerable work has been done on software-driven solutions that are oblivious to the fault model and potentially provide larger hardware coverage at low cost. The key observation underlying these techniques is that the hardware reliability solution needs to handle only those faults that become observable to software. This class of solutions, therefore, focuses on detecting hardware faults by monitoring for anomalous software behavior or symptoms. Much research has shown that such monitors (implemented in software and/or hardware) can be inexpensive and detect a wide range of hardware faults [34, 58, 71, 91, 100, 101, 118, 120, 151]. Moreover, this strategy treats hardware faults analogous to software bugs, potentially leveraging software reliability techniques and further amortizing overhead.

A software anomaly- or symptom-based detection strategy must be viewed in the context of a holistic reliability solution. First, since the hardware fault is detected through software symptoms, the latency from the activation of the fault to detection can be high (relative to traditional hardware-driven techniques). This requires a sophisticated diagnosis strategy to determine the root cause of the symptom, namely, whether it was a hardware or a software fault; in the case of a hardware fault, whether it was a permanent or a transient fault; and in the case of a permanent fault, in which field reconfigurable unit the fault occurred so as to trigger appropriate repair/reconfiguration and recovery. Simplifying detection in exchange for more complex diagnosis is a reasonable tradeoff since the former is "always on," whereas the latter is invoked in the relatively infrequent case of a fault detection.

Second, the longer detection latency also impacts recovery. Software-driven detection techniques rely on backward error recovery, typically checkpoint/rollback-based recovery. Therefore, the detection latency should be short enough to ensure that a fault-free (recoverable) checkpoint is available on detection. Another constraint comes from the need to buffer outputs until they are known to be fault-free; the detection latency should be short enough to ensure that this buffering time does not degrade performance.

Much recent work has been done on individual components of the above approach [34, 58, 71, 91, 131, 100, 101, 118, 120, 151, 21, 90, 142, 119, 109, 121]. Recent work on the SWAT (SoftWare Anomaly Treatment) project [91, 90, 131, 71, 121] has developed an integrated framework for all components of such a resiliency solution with promising results. It performs software anomaly detections using both hardware monitors (e.g., fatal traps that require no added cost or more explicit hardware out-of-bounds detectors that detect addressing anomalies) and software monitors (e.g., the kernel panic routine that involves zero cost or more explicit application-level invariant checkers). The detectors invoke a thin firmware layer that diagnoses the root cause of the symptom, leveraging the rollback/replay mechanism available for recovery. Repeated replays on different cores and units are used to systematically narrow down the source of the fault.

Once the root cause is understood and eliminated or repaired, recovery is invoked, and application execution continues.

The software-driven approach described has several advantages: (1) *generality*: it is oblivious to specific failure modes and microarchitectural or circuit details; (2) *masked faults ignored*: it naturally ignores all faults masked at the software level; (3) *customizability:* the software layer in charge of resilience can be customized to the application and system in various ways; and (4) *amortization of overheads:* the approach is inspired by online software bug detection [41, 67] and can leverage similar techniques, thereby amortizing overheads toward a holistic view of system reliability.

A key limitation of the approach is that some faults could corrupt application state in undesirable ways but escape detection. Such silent data corruptions could potentially be catastrophic, and much research is required to mitigate their effects. A key problem is that the conventional method to quantify the presence or impact of SDCs relies on fault (or error) injection campaigns (using real applications; see, for example, [91, 100, 124, 151]). With the above approach, the impact of a fault depends on the application and where in the application the fault was injected. A brute-force fault injection campaign might require trillions of fault injections (one fault per application and hardware fault site) even for simple benchmarks and hardware fault models and is clearly impractical. Therefore, statistical fault injection campaigns are used where a random sample of application instructions (and hardware sites) is selected for fault injection, but these do not provide any insight on where (if) SDCs might occur in the rest of the application. Without such knowledge, it is difficult to design protection mechanisms for the SDC vulnerable parts of the application.

Significant progress has been made recently in addressing this problem. For example, recent work on Relyzer [70, 69] proposes methods to determine when application-level transient faults are equivalent, enabling comprehensive analysis by injecting (transient) faults in only one instruction per equivalence class. Relyzer is able to both determine all SDC vulnerable fault sites with relatively high accuracy for the studied fault model and identify the reason for the SDC (i.e., the fault propagation path). The latter motivates low-cost, application-specific detectors designed to protect only those instructions that are vulnerable, thereby enabling selective, frugal, and customizable placement of detectors. The approach promises quantifiable resiliency vs. overhead tradeoff curves that can be used as appropriate by the system designer or application writer. Another project, SymPLFIED [117], takes a complementary approach of understanding the impact of different errors in the same application site without performing different fault injections for each. SymPLFIED inserts a symbolic error value and uses model checking to explore all execution paths with this value, ensuring that all paths that result in corruptions are detected and, if not, to motivate detectors. This approach has been tried only for relatively small programs, however, because model checking is resource intensive. The Shoestring project [45] has developed a pure static analysis that identifies instructions where faults are likely to be detected quickly enough (e.g., there is a short path in the data-flow graph from such a fault to enough potentially symptom generating instructions) without requiring fault injections. The rest of the faults are considered vulnerable and protected by using selective instruction duplication. Shoestring reduces the SDC rate significantly but is not yet able to eliminate SDCs or comprehensively identify where the remaining ones are.

Despite this progress, much research remains to be done to convert ideas such as these into a practical workflow that can be demonstrated for all fault models of interest and that can drive automatic derivation and insertion of detectors according to customizable resiliency vs. overhead tradeoff requirements.

5.4.2 Application-Level Detection of Hardware Errors

At the application-software level, we can develop a taxonomy of errors similar to the one presented in Section 2. We separate errors into detectable and undetectable errors. An example of an undetectable error is a corrupted matrix/vector dimension before we invoke a checksum. We can further subdivide each category into irrelevant errors (such as errors in out-of-date data that will not be used further), correctable errors

(such as a single corrupted matrix element that can be corrected using checksum), and uncorrectable errors. The key message is that although application-level detection can handle some hardware errors, it cannot, on its own, ensure resiliency. At the same time, application-level detection can mitigate the overhead of error correction in hardware or lower-level system software and thus forms part of an integrated approach to resiliency.

Application-level error detection schemes have been developed in the context of solvers for linear systems [77, 13, 12, 19, 26] and certain iterative methods for solving partial-differential equations (PDEs) [130]. These schemes are based on computing checksums of the rows and/or columns of the matrix (discretized PDE). The checksums can be shown to preserve a range of common matrix operations such as addition, multiplication, scalar product, and LU and QR decomposition (i.e., matrix inversion or solves). Checksums can thus detect errors in common matrix operations, although strictly speaking we can detect only the fact that the checksum is inconsistent, which may indicate a corrupted matrix element or an error during the checksum or matrix operation (a common misconception). With this caveat, a single erroneous matrix element can be corrected by using checksums (more elements can be corrected if the matrix decomposes and the errors occur in independent partitions). Unfortunately, the checksums have not been generalized to multigrid methods [64, 147] for solving PDEs, which are optimal in terms of flop counts compared with the SOR method described in [130].

Application-level detection in other areas of applied mathematics is less well developed (in part, this situation may be because other areas such as optimization or differential equations can be build on resilient linear algebra routines, provided the remaining computations are performed in a resilient manner). However, additional opportunities exist at higher levels of abstraction to design resilient algorithms at potentially reduced overhead. For example, when we are solving a nonlinear system of equations, $F(x) = 0$, with Newton's method, we typically promote convergence by enforcing descent in a merit function such as $p(x) = \|F(x)\|_2^2$ for the Newton step, s_k, at iteration k, obtained by solving the linear system $\nabla F(x_k)s_k = -F(x_k)$. Solvers assess progress by ensuring a sufficient reduction condition in the merit function (e.g., [47]) such as

$$p(x_k) - p(x_k + s_k) \geq \sigma \left(\|F(x_k)\|_2^2 - \|\nabla F(x_k)s_k + F(x_k)\|_2^2 \right),$$

where $\sigma \in (0, 1)$. We can use this condition to detect errors during the computation of the (approximate) Newton step. If the right-hand side is negative, then the solve failed. If the sufficient reduction condition fails, then we recompute the Newton step inside a reduced trust-region (e.g., [29]).

Other application-level error-detection schemes can be built around invariants. For example, we can detect errors in the gradient computation, $\nabla F(x_k)$, by recomputing gradients of $p(x_k)$ at a cost that is comparable to a single function evaluation using automatic differentiation [59] to detect errors ($\nabla p(x_k) = \nabla F(x_k)F(x_k)$). Stochastic optimization [16] and stochastic PDEs [27] also provide error detection schemes. In both cases, we typically solve an ensemble of systems and compute expected values. Thus, we can use the deviation from the expected value to detect potential errors in individual ensembles. However, such a scheme cannot detect all errors (e.g., those that are close to the expected value). An interesting challenge is the integration of hardware error models into the convergence analysis of these sampling methods.

Invariants can also be derived from the physics of the simulated system: wind speed is positive and does not exceed the speed of sound; nearby values cannot be too different; system energy is preserved. Programmers often check such invariants in order to debug their codes; using such checks to catch hardware errors may not add much coding effort.

The two approaches described in the preceding two sections (software-driven detection and application-level detection) are nicely complementary. Software-driven detection is most effective for errors that affect the system state or the control state of an application (e.g., wrong jump) or affect break the language abstractions (e.g., corrupt pointers); application state detection is most applicable when the application computation proceeds unperturbed, but data values are incorrect. Research is needed to further study this complementarity and understand the coverage obtained when both methods are used.

5.4.3 Behavioral-Based Detection

The number of cores used in large-scale systems already exceeds a million cores. As a result, the challenge of developing correct, high-performance applications is also growing. When an application does not complete or completes with incorrect results, the developer must identify the offending task (such as an MPI task) and then the portion of the code in that task that caused the error. Traditional parallel debugging tools [126, 92, 97, 108] often perform poorly at large task counts. Hence, research is actively under way to develop a detection toolchain that can identify the offending task and, to a customizable granularity, the relevant portion of code within the task responsible for the error.

Several debugging tools detect bugs in large-scale applications without relying on extensive manual effort demand by debuggers such as gdb, DDT, or TotalView. These more sophisticated debugging tools typically focus on detecting violations of deterministic and statistical properties of the applications. Deterministic tools can validate certain properties at runtime; any violation of these properties during an execution is reported as an anomaly. For example, FlowChecker [54] focuses on communication-related bugs in MPI libraries. It extracts information on the application's intentions of message passing (e.g., by matching MPI Sends with MPI Receives) and at runtime checks whether the data movement conforms to these intentions. Bug localization follows directly: the data movement function that caused a discrepancy is the location of the bug.

Statistical tools [53, 103] detect bugs by deriving the application's normal behavior and looking for deviations from it. For example, if the behavior of a process is similar to the aggregate behavior of a large number of other processes, then it is considered correct, and different behaviors are considered incorrect. Mirgorodskiy et al. [103] monitor the application's timing behaviors and focus the developer on tasks and code regions that exhibit unusual behaviors. This approach centers on function call traces in order to identify the trace that is most different from other traces. DMTracker [53] uses data movement related invariants, tracking the frequency of data movement and the chain of processes through which data moves.

While these tools are effective in their own domains, their primary weakness is that their designs do not consider scalability. Typically, these tools collect trace data during the application's execution and write it to a central location. They then process the data in order to detect potential problems. Some recent work has tried to rectify this problem by analyzing the application's behavior online, without any central bottlenecks. One such work is STAT [4, 88, 89], which provides scalable detection of task equivalence classes based on the functions that the processes execute. STAT uses MRNet [129], a tree-based overlay network, to gather and merge stack traces across tasks and presents the traces in a call-graph prefix tree that identifies task equivalence classes. STAT removes problems associated with a central bottleneck by reducing the trace data as part of a computation being performed within the overlay network through a custom reduction plug-in.

Another work of this type is AutomaDeD [22, 86, 85], which performs runtime monitoring of a parallel application to build a statistical model of the application's typical timing and control flow behavior. AutomaDeD models the control flow and timing behavior of application tasks as semi-Markov models (SMMs) and detects faults that affect these behaviors. AutomaDeD examines how each task's SMM changes over time and relates to the SMMs of other tasks in order to identify the task and code region where a given fault is first manifested. AutomaDeD detects which time period in the execution of the application is likely erroneous. Next, it clusters task SMMs of that period and performs cluster isolation, which uses a novel similarity measure to identify the task(s) suffering from the fault. Then, transition isolation detects the transitions that were affected by the fault more strongly or earlier than others, thus identifying the code region where the fault is first manifested. STAT focuses primarily on the state of the application once an error manifests itself, whereas AutomaDeD focuses on scalable analysis of the entire application execution.

Further work is needed to make the behavior-based detection tools robust enough to rely on in production systems. One question is how a detection system should deal with changing workload patterns, and corresponding discontinuous, but legitimate, changes in the correlation patterns. A related question is how

a detection system should handle noise in the execution environment, such as that resulting from congestion on the network switches due to competing applications executing on other nodes. For purposes of scalability, tools compress models for comparison. It is tempting to use lossy compression for this purpose. If so, what parts of the model can be compressed away because they are not germane to the error detection or localization activities? Moreover, are the models powerful enough to handle a wide variety of applications and their legitimate behaviors and yet simple enough that their parameters can be reliably derived through training and detection and localization can be done efficiently at runtime using the models?

5.5 Containment

Most system-level failures affect only a single node. Statistical analysis [15] shows that multinode failures, also called correlated failures, are rare. The probability distribution of multinode failures according to the number of nodes involved in a correlated failure is heavy tailed: failures involving the whole system are rare but still happen, for example, in the case of a long power outage.

On the other hand, the global checkpoint/restart approach to application recovery makes the simplifying assumption that if an error occurred, then any application state could be corrupted. In practice, by the time an error is detected, it may have propagated to only a small subset of the application state. Recovery could be faster if only this small fraction of the application data was repaired.

5.5.1 Strategies to Limit Propagation

Various containment strategies can be used to limit error propagation.

A priori containment recursively divides the resources of a parallel system and execution of a parallel program into nested disjoint *containment domains* (CDs); the goal is to limit recovery to one CD, at the finest nesting granularity possible. Any error or failure can be contained within some level of the CD tree and may be recovered by restoring only the state necessary to re-execute that CD. State is restored from explicit preservation clauses within each CD, which permit a range of preservation/restoration tradeoffs. These include preserving only a partial state, relying on regeneration routines or on state already available elsewhere in the system or at a higher CD level. Alternatively, one could use forward recovery of state where the state of a CD is corrected, for example, by extrapolating from the state of neighbors; this is discussed in Section 5.6.4. These approaches can be applied hierarchically. If recovery fails at one level of the system falls back to recovery at a higher level (which, presumably, is more expensive but more reliable) [28].

The choice of containment domains in terms of granularity, preservation/restoration options, and recovery and detection routines introduces new, flexible tradeoffs. For example, one can construct a strict CD hierarchy in which all communication occurs at a single CD context at a time, simplifying preservation and recovery. Often, however, it is preferable to relax this communication constraint in order to reduce preservation overhead and the granularity of recovery. When communication is allowed between CDs, data must first be verified for correctness in order to prevent silent data correction. Communication should also be logged in order to retain the ability to recover CDs in an uncoordinated manner (see Section 5.6.2). Overall the tradeoffs are between the cost of preserving state (lower relative overhead for larger domains) and the cost of CD recovery (which is relatively higher if containers are large).

Containment domains can be selected statically, based on the application structure and tuned automatically for optimal resilience. For example, in a multiphysics code, modules running the different physic codes are natural containment domains, with the containment done by the coupler that couples these modules. Alternatively, one can build containment domains automatically by tracking communication during a trial run and finding good separators in the communication graph [128].

Another possible approach is *a posteriori containment*. The logic of the application may constrain error propagation. For example, in an iterative algorithm with nearest-neighbor communication, an error can

propagate at most one neighbor away at each iteration. In a 3D problem, where each node holds a $k \times k \times k$ subcube, a bit flip in a data element will have propagated to at most $\lceil n/k \rceil^3$ nodes after n iterations. An algorithm that checks periodically for corrupted values can compute a posteriori the domain that could be affected by their error and use localized recovery.

Another form of a posteriori containment is the retention of multiple checkpoints [76, 98] and recovery based on analysis or more extensive error checking than one would normally incur at each checkpoint. When an error is manifested and the system proceeds to recovery, then based on application semantics, the contents of multiple checkpoints can be analyzed to find the most recent one that has a correct application state that can be resumed. This is a particularly powerful technique for tolerating silent errors and may avoid the need for checking checkpoints for correctness as they are committed. Such multiversion checkpoints are likely to be most viable for applications that have modest main memory requirements, when application-level determination of critical state is employed, or when additional resources such as NVRAM are available.

5.5.2 System Software

Application recovery requires a correct functioning of multiple global system services (resource managers, parallel file system, etc.). Failures in these systems are much harder to recover from and often require a time-consuming reboot. Thus, containment techniques are important for OS functions.

One approach to this problem is to partition system software in such a way that even when corruption occurs in systems code, it can be contained, and failures in a particular core do not impact other cores. Such a partitioning can be accomplished within an OS image by taking a formally verified microkernel approach with the system software [150], by using a hypervisor such as Palacios [87] or a hybrid kernel approach such as those proposed by NIX [102] or FusedOS [116]. All of these approaches create strict boundaries between different software components of the system, which can be used to facilitate the creation of containment domains within the system services and applications. Selective restart or fail-over of those partitions can refine the granularity of recovery to improve efficiency.

5.6 Recovery

Recovery will return the system to a valid state. *Backward recovery* returns the system to a previous state (a previous checkpoint), whereas *forward recovery* evolves the system to a new, correct state. Currently, in high-performance computers, system state is recovered by forward recovery, while application state is recovered by backward recovery. Checkpoint/restart is advantageous when large parts of the computation state change rapidly; this is the case with application variables in a scientific computation. Replication at MPI process level has been explored [46]. Its cost is high, and this approach is competitive against checkpoint/restart only in extreme situations. Full-node replication has not been explored in the HPC domain as far as we are aware. Its cost in development and overhead would be even more expensive than replication at the level of MPI processes.

Forward recovery makes sense when a relatively small part of the state changes; this is the case with a file system and with the system state (most of which does not change during a scientific computation). Forward recovery requires sufficient redundancy in stored state that a correct state can be recreated if part of it was lost. It also requires the use of update mechanisms that ensure that a failure in the midst of an update will not corrupt the state. Commit protocols and transaction logging for replay are two examples.

Research efforts in this area focus on avoiding the need for a global checkpoint/restart for applications, by ensuring that errors are contained and recovery can be performed locally. In addition, if the OS and runtime have a more dynamic behavior (e.g., resources added or deleted during a computation, processes migrated), then forward recovery of the OS and runtime will require additional effort.

5.6.1 Restart

The classical checkpoint/restart strategy for resilience used in most large-scale executions in petascale systems has two main limitations: (1) the time to save the state of the execution (checkpoint) is becoming unacceptable compared ith the system MTBF, and (2) all processes involved in the execution are restarted from the last checkpoint even if only one process fails. Recent results in multilevel checkpointing and in fault tolerance protocol show that these two limiting factors could be addressed and make checkpoint/restart a viable approach for exascale resilience for errors that are quickly detected—detected in less time than it takes to commit a checkpoint.

Multilevel checkpointing (*hybrid checkpointing*) consists of using multiple storage resources with different characteristics in terms of speed and reliability in order to respond to different failure scenarios. The main scenarios to consider are the crash of a process that can be restarted on the same node, the failure of a node that make that node unavailable for restart, and the failure of the entire system.

Multilevel checkpoint restart uses local storage resources (nonvolatile memory, HDD or SSD devices) as a first level of storage for execution checkpoints. A second level could use the storage resources of remote nodes. If a node failed, even if it cannot be restarted, the execution context of that node could be restarted from the checkpoint stored on remote node. Local, persistent storage can also handle node failures if it is *twin-tailed*, that is, remotely accessible even after a node failure. A third level of checkpoint considers an encoding of several process checkpoint and a distributed storage of the encoding result on several nodes. Different encoding algorithms (Xor, Reed Solomon, etc.) can be used, leading to different levels of reliability. According to the level of reliability provided by the encoding algorithm, this third level of checkpointing can be used to tolerate simultaneous multinode failures. A fourth level of storage is the remote parallel file system. This level is relevant only for catastrophic failure scenarios that could not be covered by the previous checkpointing storage levels, such as the loss of enough nodes to make impossible the restoration of the checkpoint images. Finally, mass storage can back up disk information, enabling recovery from catastrophic failures of the file system. The available bandwidth for checkpoint storage of several levels of storage is studied in [106].

Currently, two environments provide multilevel checkpoint/restart: SCR (scalable checkpoint/restart) [106] and FTI (fault tolerance interface) [14]. While SCR is keeping the file interface abstraction, FTI is providing a data structure abstraction, masking from the programmer how the data to be saved is actually managed by the library. Recent results show that a process context of 1 GB can be saved in 2–3 seconds in local SSD (2 SSDs mounted in RAID0). Such checkpoint speed is orders of magnitude faster than checkpointing on a remote file system, which requires tens of minutes on current petascale systems (about 30 minutes if the full system memory is dumped in the remote file system). An experiment with FTI on a current large-scale execution (.5 million GPU cores) of an earthquake simulation on a hybrid system composed of CPU and GPUs demonstrates very low overhead on the execution time (less than 10%) when using a checkpoint strategy compared with a computation that does not checkpoint. Other research results demonstrate that checkpointing on remote node memory is even faster than on local HDD or SSD [157]. Research still is needed, however, in order to understand how to take advantage of new storage technologies such as phase change memory. Europe has a project called AMFT to test this approach with several storage technologies; the objective is to include multilevel checkpoint restart in the PRACE software stack and to prepare for exascale.

5.6.2 Localized Restart

Checkpoint/restart is usually done at the application level. Applications periodically save state onto storage and provide a callback function to restore the computation from saved state. For most applications, the checkpoint size is a fraction of the system memory. Checkpointing is *coordinated*: the involved processes

synchronize before checkpointing and ensure that no message is in flight during the checkpoint operation.

If the computation is restarted, then all processes restart from the last checkpoint, even if only one process has failed. In general, this situation cannot be avoided. If the computation is nondeterministic, the computation after restart could follow a different path from that followed before the failure occurred; the computations of the "healthy" processes may not be valid anymore. However, most HPC scientific codes are "piecewise deterministic": the execution consists of long deterministic phases, with nondeterminism occurring at a small (possibly empty) set of execution points. THus, the opportunity exists to use message-logging protocols in order to avoid global restarts. During the fault-free execution, all messages contents and nondeterministic events (reception orders) are recorded. When a failure occurs, only the failed process restarts; its state is reconstructed by sending it the messages recorded before the failure and by forcing the message deliveries in the same order. Many variants of message-logging protocols have been developed [39]. However, they all share two limitations: (1) the contents of all the messages need to be saved, requiring a significant amount of storage; and (2) the nondeterministic events (reception orders) also need to be stored, thus impacting either the communication latency or the communication bandwidth, depending on the message-logging protocol.

A recent analysis of communications patterns in HPC applications shows two important properties: (1) communication patterns are either deterministic (the order and outcome of communication operations at each process are fixed) or send-deterministic (whatever is the order of reception for each process, the sequence of send operations is identical in any correct execution) [23]; and (2) communications show strong spatial and temporal localities and form clusters, which can be observed manually for some applications and extracted automatically with graph-partitioning tools [127]. These two properties can be leveraged to develop new fault-tolerant protocols having excellent properties in the HPC context: no global restart, no need to log all message content, no need to store reception orders, no risk of restart from the beginning. Two fault-tolerant protocols have been proposed in the literature [62, 63] from these principles. A hybrid protocol can use coordinated checkpointing inside clusters and message logging between clusters. This protocol is a good match for HPC applications built of independent modules, such as the CESM climate simulation code [1]: checkpoint/restart can be done independently for each module (cluster), and logging (within the coupling toolkit) handles interaction within modules. For real applications, the number of messages to log is a small fraction (10%) of all the messages sent during the execution [63]. Other hierarchical, hybrid fault-tolerant protocols, combining coordinated checkpointing with some form of message logging), have been proposed that do not consider communication determinism [20]. They require logging, in some way, the message reception orders of all messages.

While this progress is encouraging, many research questions remain open: how to form clusters to reduce the number of messages to log, how to adapt clusters to the different communication patterns seen during the execution, how to prove the deterministic or send deterministic nature of communication patterns automatically, how to organize a fully distributed recovery, how to better understand sources of nondeterminism in applications that show nondeterministic or send-deterministic communication patterns, and how to address them.

Localized restart reduces the total I/O volume needed to restart, but it may not reduce the restart time if all nodes have to wait for the failed node to be restarted. Nevertheless, it still may result in lowered power consumption, since the waiting nodes can reduce their power intake. Furthermore, the restart can be accelerated by being distributed across multiple nodes.

5.6.3 Fault-Tolerant Data Structures

Between application-level and restart schemes, there are runtime-level techniques for redundancy and repair. These techniques can operate at the data structure level, below even a typical application abstraction, and by encoding additional information into the data structures enable them to be reconstructed in case of

error. Common examples include i-nodes in filesystems, redundant virtual-physical mapping information in operating system page tables, trees and lists with multiply linked structures, and redundancy-encoded arrays and data structures. These techniques offer the potential for significant recovery capability under software (compiler, runtime, OS, even application) control, and they support selective and flexible usage. One example of such structures has been proposed in the Global View Resilience (GVR) system [48].

5.6.4 Application and Algorithmic Recovery

Application and recovery techniques can use the *algorithmic redundancy* available in many parallel algorithms, in order to recreate a valid computation state if the loss of a (small) part of the state has been detected. Many simulations use iterative methods on meshes. When a catastrophic node failure occurs and is communicated to the remaining nodes, such a method could approximate the missing information and continue with the computation, by extrapolating the missing information from the remaining information. If the method had suitable convergence properties, then the error thus introduced would be smoothed out, possibly at the price of additional iterations. More sophisticated recovery methods that use a hierarchy of meshes generated for multigrid methods could also be developed. These methods would traverse from fine to coarse and back using restriction and interpolation operations. By moving to a coarser level, one could estimate the numerical values of the computational node that failed, using the interpolation operation and neighboring values, and then construct a new mesh for the missing patch and apply interpolation operations. Such an approach requires knowledge of how to remesh and recover the mesh hierarchy, and possibly a rebalancing of the computations to prevent neighboring nodes from becoming a computational bottleneck.

In applications with different types of structure, the recovery mechanism might be less intrusive. For example, in branch-and-bound methods for mixed-integer optimization, which recursively subdivide the domain and solve optimization problems on each subdomain, a tree structure maintains the current state. As long as the tree structure was available, if a solve on a subdomain did not complete because of node failure, that subdomain could be recovered from the tree and the optimization problem solved on a different node. The same approach applies to any functional execution model, where variables are not mutable: if the evaluation of a function fails to complete, it can be just recomputed. This approach is heavily used by Hadoop to provide resilience [31].

This discussion also suggests that algorithm-level checkpointing can be more efficient than system-level checkpointing. For example, to ensure that function, gradient, and Hessian computations are correct, one needs only to checkpoint the computational graph of the nonlinear functions, which is orders of magnitude less information than the values and sparsity patterns. Similarly, branch-and-bound schemes need only to checkpoint the root node of each distributed solve, which can be stored by using two binary vectors.

Another advantage of algorithm-based recovery is that it may not be necessary to replay the MPI messages since the last checkpoint. For example, if a node fails during a distributed solve of $F(x) = 0$, we can simply resume the Newton iterations from the checkpoint, because we are not interested in the sequence of iterates, x_k, but the final solution. This opens the door for new hybrid methods that combine Newton with Gauss-Jacobi steps. The analysis of such methods remains an open problem. In the context of stochastic optimization, asynchronous techniques already exist that can accommodate missing subproblem solves [93].

5.6.5 Fault-Tolerant MPI

Application-level recovery needs to be preceded by system-level recovery. For example, if a node has failed, then the application has to be made aware of the failure and has to continue its execution in a well-defined environment, in order to execute the recovery code. This problem is usually addressed in the context of MPI. Can we ensure that the failure of one node will not cause processes running on other nodes to crash? How can we inform the other processes of the failure? What is the state of MPI after the crash? Projects

aimed at providing a fault-tolerant MPI have been going on for over a decade [44, 18], and several prototype implementations of fault-tolerant versions of MPI exist. However, the MPI forum has not agreed yet on a standard for fault-tolerant MPI.

A similar problem occurs for any library that provides a view of collective operations and consistent state across multiple nodes. These include mathematical libraries and I/O libraries. For each of these, we need to define what the state of the system is after a failure and how the information about the failure is propagated.

5.6.6 Accommodation of Errors Based on Naturally Redundant Information

For some applications, we may not need to recover from node failures at all. For example, in derivative-free parameter estimation of a complex simulation, a node failure could be ignored and treated as a simulation failure. However, not all simulation failures are the same. A graceful failure can yield partial information that could be used when determining the next experiment to perform for the optimization. Structured simulation-based optimization techniques can use this partial information to build partial interpolation models and thus become resilient to node failures. Similarly, we could use partial solutions to simulations at a looser tolerance as long as we account for the truncation error in the model and optimization.

This approach can be likened to controlling the noise in simulations [107]. For stochastic noise, model-based optimization methods have been developed that specify both a candidate point and the number of replications needed to obtain sufficient accuracy. Parallel replications at a fixed point can be used to control stochastic noise but not deterministic noise. For deterministic functions we could use nearby points and Taylor's theorem to bound the noise in the simulation. By neighborhood sampling we could reduce the noise in many settings; these samples may already be available from previous computations of the algorithm.

An alternative approach is to use insight into the application in order to reduce the probability of failure by reducing the memory footprint of an application. Variable-precision arithmetic can help in this approach by using bounds on the precision requirements for Newton solves in order to compute low-precision steps initially. Analysis tools, such as those developed for automatic differentiation and estimating computational noise, could identify blocks in the code for which higher precision would lead to improved precision in function evaluations. Based on this identification, one could restructure the computation of a function so that the least-precision arithmetic is used in each block to obtain the required precision in the overall function evaluation. Similar ideas could be applied for gradient and Hessian evaluations. User-provided and automatically generated codes for quantities derived from function values (such as derivatives) can be significantly less precise than is the underlying function. Analysis of the underlying computational graph (for example, as done by Kubota) could provide insight into reformulations of the derived code that yield both function and derived values to specified precision.

5.6.7 Rejuvenation

Software rejuvenation is meant to mitigate the problem of software aging, in which the state of the software system degrades with time [25]. The primary causes of this degradation are the exhaustion of operating system resources (such as file handles or network sockets), data corruption, and numerical error accumulation. Eventually, software aging leads to performance degradation or correctness problems such as hang or crash failure. Some typical causes of this degradation are memory bloating and leaking, unterminated threads, unreleased file-locks, data corruption, storage-space fragmentation, and accumulation of round-off errors. These causes also affect HPC applications, and hence software rejuvenation is a relevant technique in our toolchest. In particular, accumulation of round-off errors is a problem in some numerical computations that appear in HPC applications [66].

Software rejuvenation essentially involves occasionally terminating an application or a system, cleaning its internal state, and restarting it. This process removes the accumulated errors and frees up operating system resources, thus preventing in a proactive manner an unplanned and potentially expensive system outage due to the software aging. Much research has been done in order to determine optimal times to do software rejuvenation [10, 60], for example, when the load on the system is low, the amount of corrupted state is likely to be small, or a failure is impending. With an appropriate choice, the cost of system downtime can be reduced significantly compared with reactive recovery from failure.

Surprisingly, software rejuvenation has not been widely used in HPC applications. In [110], the authors argue that rejuvenation should be tried in HPC applications only at the level of individual OS kernel, rather than the entire system. They propose three scheduling strategies for rejuvenation: using the MTTF, the median of TTFs, and the reliability of the system. Based on failure data extracted from System 20 at Los Alamos National Laboratory [96], they evaluate the hypothesis that rejuvenation together with checkpoint/restart can reduce the lost computation over simply checkpoint/restart. The verdict is mixed. Only by a careful estimate of TTFs can rejuvenation give benefits. Not surprisingly, more rejuvenations quickly reach a point where they hurt overall performance.

Nevertheless, it seems worthwhile to further explore the application of rejuvenation in HPC applications. The first issue that needs to be considered is what state should be saved and "rejuvenated." Related to this is how that state should be compartmentalized so that a quick rejuvenation is possible. The second issue is when to trigger the rejuvenation. In addition to the factors that have already been explored in non-HPC domains, here one must also consider interactions of that node being rejuvenated with all the other nodes in the cluster on which the application is running. Done right, software rejuvenation holds the promise of extending the MTBF and reducing the frequency of checkpoint/restart.

6 System View of Resilience

Editor: Rinku Gupta
Contributors: Pedro Diniz, Pavan Balaji, Pradip Bose, Subhasish Mitra, Dean Liberty, Jon Stearley

We discussed in the preceding section mechanisms for detecting hardware errors at the system or application level. A similar interplay between the various system layers applies to all aspects of resilience. Proper interfaces between the different layers are required in order to propagate information about faults, errors, and failures in various subsystems to the subsystems that will be involved in managing them: the subsystems that need to act upon the information to contain and recover from the errors and the subsystems and will be further involved in diagnosis and repair. Furthermore, resilience techniques are often based on the assumption that a single fault will occur at a time. It is hard enough to address in a systematic manner all possible faults and practically impossible to address in a systematic manner all possible combinations of multiple faults. The "single fault" assumption is statistically valid if errors are rare and are cleared rapidly. It also requires the error-handling infrastructure to be flawless. Therefore, the correctness and the performance of the fault-handling software are paramount considerations.

6.1 Fault and Error Management

Each layer of running software should be able to optionally specify its dependencies—namely, which errors in other subsystems may affect it and the designated error handlers for different types of errors, whether internal or external. Operational error handling may also dump local data in support of later fault management activities (diagnosis and repair). In general, the invocation of error handlers must be carefully ordered. For simplicity, let us consider each higher-layer (or procedurally deeper) error handler as being pushed onto a stack. When passing error-handling control to successive error handlers, the system will invoke the topmost

handler on the stack. When returning, the handler will indicate whether the error was successfully handled. If it was not, the next handler on the stack is invoked. In this way, error handling passes from the most specific to the most general handler, with increasingly general actions attempted to recover from the error. The problem is complicated by the existence of a horizontal as well as a vertical organization: The error handler can be invoked on a node different from the node that signaled the error; the error can be signaled in a place different from the place where it occurred; and errors may be signaled multiple times, through different mechanisms. For example, the failure of a node can lead to an error being signaled through the hardware monitoring infrastructure to the system console; it may cause communication timeouts, generating error messages at other nodes that communicate through the failed node; and it may generate a timeout on a system or application heartbeat. We need to ensure that recovery actions are not duplicated and are properly ordered.

As information on faults or errors propagate through the system, it is also important to properly map their semantics from level to level, into terms meaningful to each level and to the recovery abilities of each level. For example, if a bit switched in memory, the hardware layer will want to know the physical address of the affected location and will want to further localize the failure to a hardware subcomponent, such as CPU, cache, or memory. The system layer will want to know how far the error could have propagated; the application level will want to know which variables may be corrupted; and so on. Therefore, it is useful to define at each level the set of conditions that can be signaled, so that a fairly generic, portable error interface can be used to program error handlers at each level. Having such a generic classification of error types for applications will allow a more portable programming model and a simpler evaluation of the effects of errors on application execution.

Diagnosis and repair may involve more elaborate actions that have to be coordinated across layers. For example, a node failure is recovered by replacing the node (possibly involving the global resource manager), updating routing tables and MPI structures, and restarting from the last checkpoint. Later diagnosis and recovery actions may include running detailed diagnostics and replacing the node.

A viable model for diagnosis and repair could be a software repository that allows subscriptions to fault management updates, thus allowing arbitrarily complex recovery and repair actions. In addition, these actions need information about static and dynamic configuration of the system: what the hardware and system configuration is, which applications run on which nodes, what the software configuration of the application (source code, compiler versions, library versions, etc.) was, and so on. Today, this information is typically distributed across multiple databases or is not captured at all. As a result, root cause analysis is much more painful than it should be. All configuration changes should be captured and configuration information stored in a repository, using schemata that reflect the logical system organization.

6.2 Reporting of Software-Detected Errors

The various software layers, including the top-level application, can detect errors that were not caught by the lower-level layers. The application code may detect outliers, for example, that may indicate silent data corruption. Therefore, reporting can also move information downward. This approach is complicated, however, since the information cannot be fully trusted (is the algorithm sure that a silent data corruption happened, or could this be the effect of a data race?) and the information comes with a lower level of detail than information produced by lower-level detection mechanisms (the algorithm may not know where and when was a bit flipped). The passing of such information is likely to invoke a complex procedure that evaluates the reliability of the information, based on other information available to the recipient (e.g., information about the sender) and triggers activities to isolate and diagnose potentially faulty components. Such a level of activity is probably more appropriate as part of diagnosis and repair, when more complex, trainable logic can be used.

6.2.1 Error Management: Algorithm Hints and Watchpoints

Different parts of the software stack typically have different capabilities for handling a propagated fault or error. For example, in many situations, an application and its runtime may be able to validate its results and recover from an underlying error or fault. Similarly, a communication library may be able to establish a dynamic alternative connection on detection of a lost communication error. In such cases, an interface would be useful that allows the different software to express their inherent fault tolerance capabilities. Algorithmic hints would also allow lower-level software to understand what level of error semantics is useful to the upper application layers and what level of fault information could be conveyed to them. We explore algorithm hints in greater detail in Sections 6.5 and 6.6.

In many situations, the application or the lower-level subsystem can recover from an underlying error but needs to execute recovery code for that purpose. For example, an application may tolerate a corrupted data value by interpolating a replacement value from neighbor points in a mesh. This approach is most efficient if the error is detected and the recovery action enacted as soon as possible after the error occurred. In this scenario, the application (or underlying runtime) will register its ability to handle some types of error and will register the exception handler to be invoked when such an error occurs. For example, the algorithm could identify memory regions that it wants to "watch" along with the recovery procedure for errors in this region. Then when an error is found by the hardware and translated up the software stack, it will trigger the appropriate exception handler, passing to it the location and type of error. The compiler and runtime need to ensure that the granularity of error reporting by hardware (e.g., ECC block) matches the granularity of software objects.

6.2.2 Error Management: Communication Errors

Communication errors require added attention because their effect can be global. A misrouted message could corrupt state at any node in the system. This can be handled in a variety of ways.

We can provide sufficient levels of error handling in hardware to ensure that communication errors are fail-stop errors, where the communication fails but no incorrect message is delivered. An end-to-end protocol (a variant of the sliding window protocol) can ensure that message deletions are detected and corrected for point-to-point communication channels. However, the support for correction may require additional buffering space (to save message copies) and additional latency (to receive acknowledgments and ensure that messages will be transmitted in the right order, even in the face of errors). The problem is harder to manage for collective communications or one-sided communications. Application hints that relax message-passing semantics (e.g., relax ordering requirements) could be used to improve communication performance.

6.3 Responding to and Handling of Faults/Errors

Various components of a system (whether software or hardware) can receive information about a fault or error occurring in a specific part of the system. Several of these components could independently be interested in handling this fault and initiating a recovery section. These different recovery actions may be interdependent: they need to occur in a correct order, and the recovery procedure for a component may depend on the outcome of previous recovery procedures. For example, a partition may require a new node to replace a failed node. The application recovery could follow different paths if the request succeeded or failed.

Response prioritization is an inherent part of response negotiation. The systemwide resilience infrastructure will need to support mechanisms that will allow declaring response priorities of various components for the variety of faults that they might receive. In addition, interfaces are needed to allow components to specify the outcome of their recovery procedure. On failure of failure handling by the first component, the infrastructure should be able to delegate the responsibility to the next component on the list. Response

negotiation can become more complicated when response priorities for components differ between various job executions.

6.4 Fault/Error Propagation and Security Implications

While a large amount of information related to faults and errors is present on systems, not all of it can be made available to all consumers because of security reasons. Often, low-level hardware/system fault information is made available only to administrators of the system. A systemwide fault-sharing infrastructure needs to have mechanisms and interfaces to control access to different types of information, for example, by using capability-based security.

6.5 Top-Down View of Errors

Higher-level algorithms may not require notification or recovery from certain types of errors, since the normal course of computation will overcome the error. A prototypical example is solving a nonlinear system of equations using Newton's method. The basic steps of Newton's method are to compute the residual, solve a linear system of equations with the residual on the right-hand side to obtain a direction, determine a step length along the direction, and update the iterate. For well-scaled problems, Newton's method can ignore errors in many parts of the computations without suffering ill effects.

We will often tolerate errors in the least significant digits of the mantissa of floating point numbers (say, the last 8 digits), as these would be analogous to rounding errors; but we would need to detect and correct errors in the sign, exponent, or most significant mantissa digits. The recovery may be recomputation in the case of the residual or switching to using the steepest descent direction in the step length computation. We cannot tolerate errors in the the sparsity structure of the Jacobian matrix, but we can easily tolerate low-order errors in the nonzero values of the Jacobian matrix. High-order errors in the sign, exponent, or most significant digits can be also be tolerated, but there are consequences, such as degradation in the convergence rate of Newton's method and requiring extra iterations to converge to an acceptable solution. As long as the Jacobian matrix is correct or suffers from only low-order errors a large fraction of the time, then there is little impact on performance, and one may need to perform only a single extra iteration. We do note that a change in the sign or magnitude of the values can result in a positive definite matrix becoming indefinite and thus impacting linear solvers, such as the conjugate gradient method. Errors in the computation of the norm of the residual can be tolerated in the step length calculation. uch errors may not be tolerated in the convergence tests, however, if the error results in a smaller norm of the residual that triggers premature termination of the algorithm.

Similar observations can be made about other types of algorithms and about other software subsystems. These suggest the need for an interface that enables an application or a software subsystem to provide hints and describe which lower-level errors it can tolerate. The lower software layers and the hardware could then provide differentiated levels of resilience, protecting state that the application cannot repair, if corrupted. These could include using more resilient memory, duplicating critical computation (done automatically by the compiler and runtime), or checking double-precision calculations with (cheaper) single-precision ones. Providing increasing levels of resilience would come at higher costs, with tradeoffs of both power and performance, thus requiring that the provided set of interfaces be expressive enough to allow upper-layer software to specify their tradeoff preferences.

6.6 Bottom-Up View of Errors

This section focuses on issues related to exposing error semantics upstream (to higher-level libraries or applications), the amount of information to be exposed, and the information to expose. Cost is a big challenge

in detecting and correcting errors in the underlying hardware. The challenge is how to minimize the power and performance costs of highly effective error detection. Can we make use of high-level (e.g., application level or high-level system stack) information to minimize this cost?

Error Semantics and Translation. While an error can influence multiple layers of the hardware/software stack, how an error is interpreted can differ for each layer of the stack. For example, a fault on flight #1508 might be relevant at the hardware layer to correct or work around, but it might not have much semantic meaning for the application. Similarly, the fact that memory variable "X" is corrupted might be relevant for an application, but it might not have much semantic meaning for the hardware developer, unless the virtual memory address and eventually the appropriate physical memory address translation are known.

Amount and Type of Error Information Exposed. The amount of error information propagated to upper layers needs to be tunable. While some upper layers can benefit from having information on every ECC error (corrected or detected but not corrected) that the hardware encounters, other upper layers might be interested only in uncorrected errors. Similarly, an application might not necessarily care about errors on all of its memory regions. For example, as discussed in Section 6.5, if a higher-level library can correct memory faults on a region of memory, it might not care about the lower level of the stack returning errors for that region of memory. Such a model should also allow software architects to define the contract or expectations they have from the lower layers of the stack.

How can such hints on criticality be generated? Does the hardware need to provide low-level information to the higher layers so that the critical hints can be generated? Once hints are generated and passed down, several opportunities can exist.

Example 1. The Built-In Soft Error Resilience (BISER) technique [104, 156] can be configured, during system operation, to operate in one of two modes: an error-resilient mode in which BISER protection is turned on, and an economy mode in which BISER protection is turned off. Such configurability can be implemented in hardware and may be activated through software orchestration. It can minimize the system-level power cost of BISER by turning on the error-resilient mode only for critical computation. However, dynamic reliability management across multiple abstraction layers and orchestration of information flow across abstraction layers to utilize such configurability during system operation are open research questions. For BISER, one can piggyback on existing scannable signals available on-chip, but a general question concerns the costs that are incurred for such configurability at the hardware level. Can such configurability be implemented for arbitrary techniques (e.g., easy for core/thread duplication)? Is it easy for inline checking techniques such as parity prediction? What is the level of configurability that should be supported?

Example 2. One can combine software-level error resilience techniques with circuit-level techniques using a "temporal combination" approach. For a memory controller unit (MCU) in a multicore SoC, for example, we can start with request duplication with BISER flip-flops in economy mode. We then switch the BISER flip-flops into error-resilient mode (i.e., incurring high power costs) and turn off request duplication when the systems stalls because of pending requests (which indicate high-traffic situations). We switch back to request duplication with BISER flip-flops switched to economy mode when all queues have only a few entries (to indicate low-traffic situations). Such "temporal combination" simultaneously incurs very small performance cost (performance impact similar to that of BISER-only and far better than request duplication-only) and small energy cost (similar to request duplication-only and far better than BISER).

Example 3. Depending on workload, temperature sensors, and so forth, the fault-sharing framework can pass on the information to hardware to initiate fault management, for example, on-line circuit failure pre-

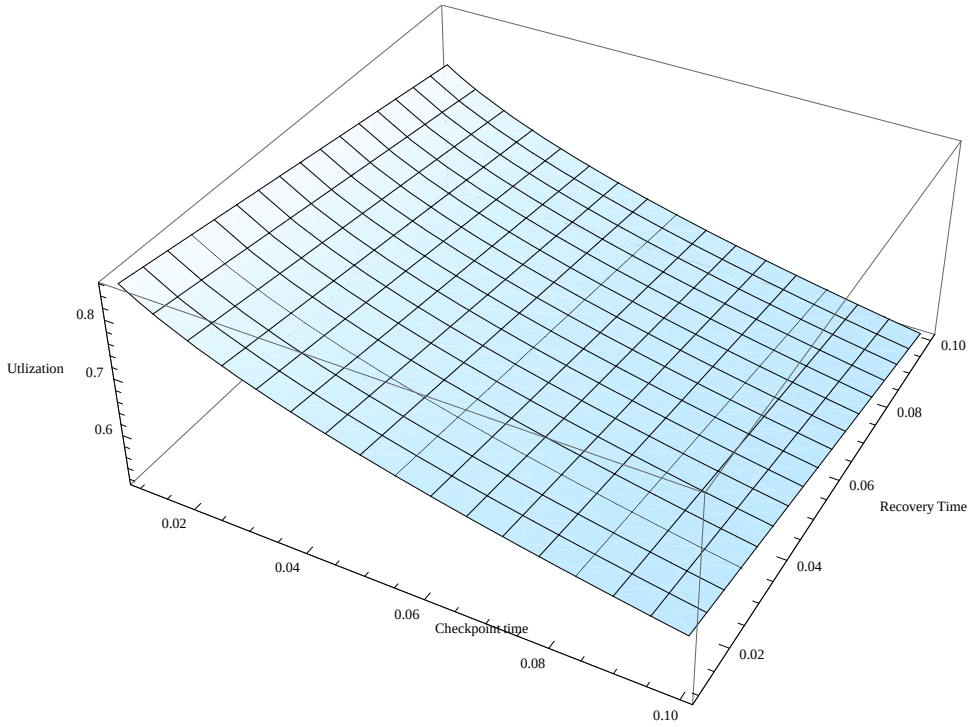


Figure 5: System utilization as a function of checkpoint and recovery time

diction through reactive on-line self-test and diagnostics. This approach minimizes any side effects and can initiate proactive self-repair.

7 Possible Scenarios

We present in this section several possible scenarios for handling failures at exascale, describe their pros and cons, and discuss technologies needed to support each scenario.

7.1 Base Scenario

In the base scenario, errors are handled the same way they are handled now: applications use global checkpoint/restart, and system software is either restarted upon failure or handles its own recovery. The obvious advantage of this scenario is that it requires (almost) no change in current application codes and requires no changes on the overall infrastructure for error recovery. (One required change will be more frequent checkpoints; with high-frequency checkpoints, it is unlikely that checkpoints will be identical to the output that goes to long-term storage or to in situ analysis.)

The performance of global checkpoint/restart schemes has been analyzed by multiple authors [153, 30]. We recapitulate the analysis in Appendix B. This analysis enables us to compute an optimal checkpoint interval, given checkpoint time and mean time to failure (MTTF); next we can compute the *utilization* of such systems, namely, the fraction of the total computer time that is usefully applied to computation, rather than used for checkpointing and restart or wasted because of failures.

We plot in Figure 5 utilization as function of checkpoint time and recovery time. Utilization depends on the length of checkpoint and recovery relative to MTTF; if all three parameters are increased or decreased by the same ratio, then utilization is unchanged. Therefore, we express checkpoint time and recovery time

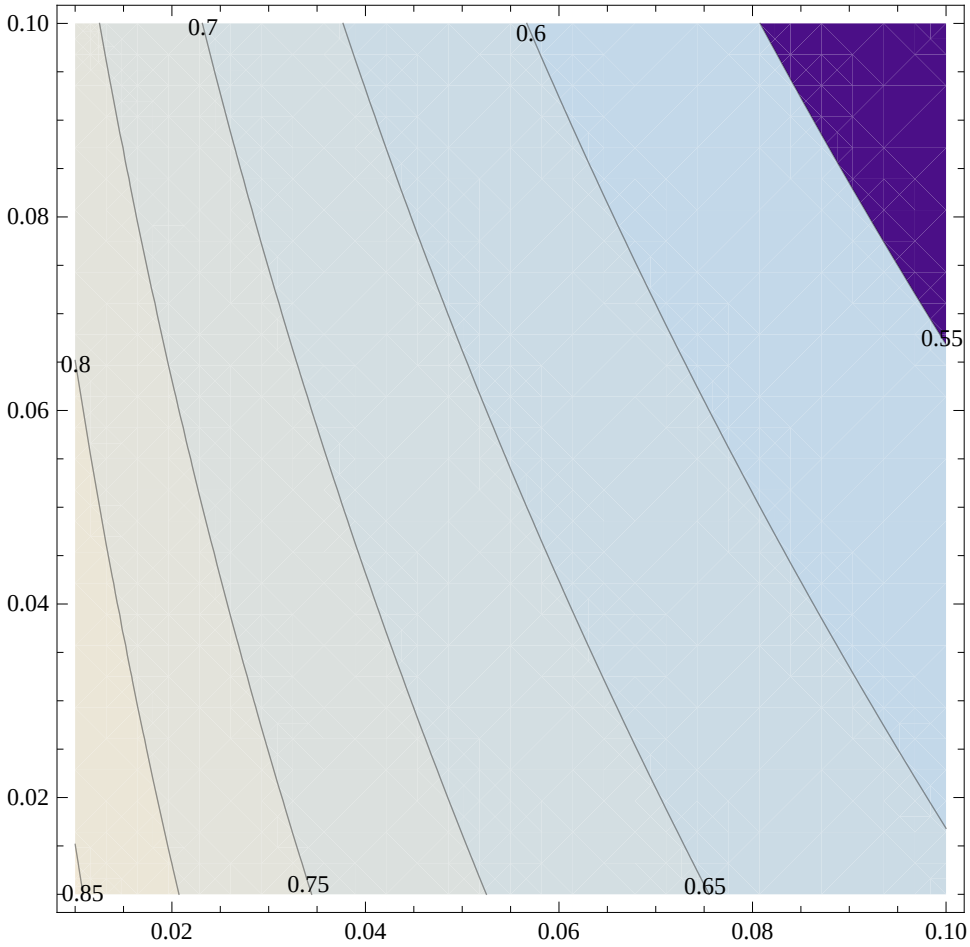


Figure 6: System utilization as a function of checkpoint and recovery time – contour map

as a fraction of MTTF. Figure 6 shows the same data, in the form of a contour map.

Suppose we want to achieve a utilization of more than 80%. Then Figure 6 indicates that we need to keep checkpoint time at 1%–2% of MTTF and recovery time at 2%–5% at MTTF. Assume that the MTTF of an exascale system is 30 minutes. Then global checkpoint should be done in less than 20 seconds, and recovery in about a minute. It does not seem feasible to checkpoint so fast on disk, but it is feasible to checkpoint in a few seconds in RAM. Several schemes have been proposed for *hybrid* or *multilevel checkpointing* where frequent checkpoints are done in memory or, less frequently, on disk [106, 14]. One can use either nonvolatile RAM to store checkpoints or volatile RAM with a "RAID" scheme that allows recovery from the failure of one (or more) nodes. (The second option may be constrained by the limited number of write cycles supported by various NVRAM technologies.)

Such a scheme has an obvious cost: the need to significantly increase the amount of memory by, say, 50%. This will have a significant impact on the system acquisition cost. Note, however, that the increase in power consumption is negligible. This is obvious for NVRAM but true also for DRAM, since checkpoint memory would be in standby mode most of the time.

The two main obstacles to this approach are the need to detect errors in a timely manner and the need for fast recovery. We looked in Section 3 at soft errors due to particle strikes and estimated that current technologies could be used to keep their frequency to current levels at a cost of $< 20\%$ additional silicon and power. However, these numbers involve considerable uncertainty. Particle strikes are only one of

multiple potential new sources of errors, and the impact of near-threshold logic was not taken into account. Furthermore, for reasons explained in Section 3.5, there is no certainty that the market will produce the low-energy, high-resilience components that would be needed to avoid silent errors in hardware at an acceptable price. If silent errors can propagate into checkpoints, then checkpoints are not of much use.

While the time for backward recovery from checkpoint at the application level is essentially gated by I/O rates, the time for forward recovery that reboots or repair various system software components is gated by the computation overhead of boot or repair code. Boot time of large systems may currently exceed 30 minutes; without a change, the boot time of an exascale supercomputer could exceed its MTBF—not a sustainable situation. This last problem is common to all envisaged scenarios for resilience. Therefore, advances that reduce boot time and repair time for the system infrastructure at exascale are essential. Also essential are advances that reduce the likelihood of system failures—in particular, software failures.

7.2 System Software Scenario

In the second scenario, hardware is assumed not to provide enough detection, and therefore silent data corruption events occur too frequently to be ignored. Instead, we assume that data corruption can be prevented, detected, and corrected or else tolerated with no change to the application software.

Not all hardware errors have the same severity. A bit flip in a large array of data may have little impact on the final answer; a bit flip in a program counter or a data pointer is likely to have stranger, less predictable impact; and a bit flip in a page table or a routing table is likely to have a catastrophic impact. Luckily, the software error detection schemes described in Section 5.4.1 are more likely to detect the "bad errors"—those that will have a significant impact on the final answer or will cause a crash. Furthermore, redundancy can be used in order to reduce the probability of "bad errors." Critical computations can be executed twice (and the redundancy can be introduced automatically by a compiler [125, 154]); more reliable memory may be used for more sensitive data; and so forth.

A plausible hypothesis is that silent hardware errors fall into two categories: "pleasant errors" that can be treated as aleatoric uncertainty in the computation and "nasty errors" that, essentially, change the computation model. The latter must be treated as epistemic errors that cannot be modeled as statistical noise and have to be avoided or corrected. Fortunately, "nasty errors" are likely to be less frequent than "pleasant errors" in large scientific codes and are easier to avoid or correct. If this hypothesis is correct, then silent data corruption events can be survived with little to no change in application codes. This hypothesis needs to be validated for all or a large fraction of large scientific workloads.

The system scenario also covers schemes for using local restart, thus reducing restart overhead—provided that the construction of node clusters (application containers) can be automated.

7.3 Application Scenarios

Handling resilience without changes in application codes may turn out to be too expensive. We envisage two subcases: those in which the application code has to handle only tolerance or detection and those in which the application code also has to handle correction.

We discussed fault-tolerant algorithms in Section 5.6.6, algorithmic fault detection in Section 5.4.2, and algorithmic recovery in Section 5.6.4. The main issue with these techniques is that they are specific to one or to a family of algorithms. We need *generic* techniques that will apply to *all* computations of interest for the exascale era and *efficient* techniques that will apply to the *large majority* of these computations.

Another issue is how to compose different approaches to resilience. If one module can tolerate silent bit flips, another module can detect them efficiently and recover using checkpoints, and yet another module needs redundant execution, how are these three modules coupled in one application?

8 Suggested Actions

We outline in this sections actions that are suggested by this workshop.

8.1 Information Gathering

The different scenarios imply very different strategies for achieving the required level of resilience: from possibly significant investments in hardware that has little use outside extreme-scale computing to possibly significant investments in recoding existing applications. At this time, we do not have enough information to choose a direction; more information gathering is essential. We propose several activities for that purpose.

8.1.1 Characterization of Sources of Failures on Current Systems

DOE has a rich source of information in the form of the message logs that are collected at each of the supercomputing centers at DOE labs. Unfortunately, most of this data is not centrally collected; also, different vendors use distinct terminologies, so that data cannot be directly compared. To the best of our knowledge, there are no vendor restrictions on the publication of data owned by the various centers. Initial discussions with vendors indicate a willingness to help analyze the data.

We propose to establish as soon as possible a centralized repository within DOE that will systematically collect event logs and other relevant information from all DOE supercomputing centers. In parallel, we propose to invest in tools to normalize these logs into a vendor-neutral notation and to anonymize them. DOE would then make these cleansed logs available to the broader research community.

We note that the paper of Schroeder and Gibson on "Understanding Failures in Petascale Computers [135] cites three repositories for computer failure data. Two (at LANL and NERSC) do not seem to be accessible on the web. The third, the Computer Failure Data Repository (CFDR) at http://cfdr.usenix.org, which is maintained by Bianca Schroeder, is easily accessible. This situation suggests that a community effort will be more productive that the individual efforts of supercomputing centers.

Event logs provide failure symptoms but do not provide a root cause for each failure. Root cause analysis is now a tedious manual process that engages much of the time of the staff at supercomputing centers. We propose two efforts on root cause analysis:

1. Develop a registration system that will facilitate recording the results of the manual root cause analysis. The gaol is to annotate event logs with the result of such analyses.

2. Develop better tools for root cause analysis. Existing software products, such as SMARTS of EMC, could be a good start for such development.

8.1.2 Study of Frequency of Silent Errors

Currently there exists a large uncertainty about the frequency of silent data corruption events. On the one hand, the practice of supercomputer users is to assume such events do not occur. On the other hand, anecdotal evidence on the nonreproducibility of computations that are supposed to be bit reproducible suggests they do occur, and occur quite frequently.

We propose to push a study on the frequency of SDC events on current supercomputers. Such a study could be effected by running a background job on as many nodes as possible on various supercomputers. The job would produce bit reproducible, testable results and be used to detect SDCs.

8.1.3 Refinement of Estimates on Future Hardware Technologies

The main uncertainty about future roadblocks to resilience concerns the frequency of hardware silent data corruption events. Our analysis showed that cosmic radiation induced SDCs could be managed at a cost of less than 20% in circuitry and in power consumption—using current methods. More research in this area could further reduce the gap. However, the study ignored other issues (subthreshold logic, aging). In any case, the main uncertainty about future hardware technologies is less about what can be done and more about what will be done by industry, given market forces. It will be useful to complement technological studies with economic studies, based on the evolution of different markets (high-end server, cloud, mobile). The key question to be addressed is the following: What is the market size for processors that have low power? have high resilience? high floating-point performance?

8.2 Research Areas

We divide research directions into three categories:

Necessary Technologies: Technologies that will be necessary for resilience at extreme scale, no matter what scenario ends up being pursued.

Generally Useful Technologies: Technologies that will be useful no matter what scenario ends up taking effect

Scenario-Specific Technologies: Technologies that will come into play only under a subset of the scenarios.

DOE investments in R&D should focus on the roadblocks we know will certainly exist, and less so on roadblocks that are still hypothetical. On the other hand, one may justify investments in scenario-specific technologies as a risk-reduction action, if the technology is necessary under some plausible scenario and the time lag from research to deployment is expected to be significant.

8.2.1 Necessary Technologies

In any scenario, it will be essential to reduce the frequency of system failures, contain them, and reduce recovery time from system failures. Some of the problems may have simple engineering solutions, for example, fast boot from nonvolatile memory. Solutions to other problems may require new structures and mechanisms for global system services. Some of the current research on error containment that is now focused on application errors could be fruitfully applied to system errors. Faster recovery form file system failures will be important.

Another critical technology is the communication infrastructure that enables recovery actions at different levels of the system. This infrastructure will need to be as resilient as the current out-of-band networks that collect hardware monitoring information and channel it to the hardware monitoring console. But the infrastructure also will need to handle software failures and avoid the sequential bottleneck of one global monitoring point.

8.2.2 Generally Useful Technologies

Some technologies are useful no matter what scenario takes effect. One example is fault prediction and avoidance—predicting node failures and migrating a node workload before the node fails. Successful fault prediction and avoidance effectively increase the system MTBF, thus increasing the system utilization.

Another example is provided by technologies for fault containment. Avoiding a global restart can reduce the time and energy consumed by restarts, thus improving system performance.

8.2.3 Scenario-Specific Technologies

Scenario-specific technologies include all the technologies that would be required if SDCs become a major problem: technologies for system software error detection, containment, and correction and technologies for application-level error tolerance, detection, containment, and correction.

Arguably, the choice between handling errors in hardware or in firmware is a vendor choice. Vendors will choose one or the other, or a mix of the two, according to relative non-recurring and recurring costs of the two approaches. Research in DOE can help in exploring firmware-level resilience solutions. We recommend a co-design collaboration between DOE research and vendors in exploring the right mix of hardware and system software approaches that would provide the appearance of a failure-free system to the application layer.

Application-level error handling is a much more significant departure from current practice, one that should be entertained only if the other options are not feasible or have a significant cost. Application-level error correction will require new services from the underlying hardware and software—for example, the ability to provide differentiated resilience quality for computations or storage, fault-tolerance at the level of MPI and other global libraries, and mechanisms for signaling errors to application code. Since these are needed for research in application-level error handling, their development should be a priority.

A main focus on application-level error handling should be on generic techniques that apply to all applications or large classes of applications. These are needed in order to avoid having to develop a unique solution for each application code.

We note that although application-level tolerance or detection of SDCs is more important than application-level correction, global/checkpoint restart is still viable at exascale, provided one can ignore or detect errors.

8.3 Integration

Much of the current research on resilience is addressing small sections of the problem, for example, how to tolerate or detect SDCs errors for a particular algorithm. Point solutions are useful only if they fit in an overall resilience architecture. For example, algorithm error-handling may assume that some system services continue to be available after an error occurred and may be able to handle some errors (a bit flip in data) while ignoring other errors (a bit flip in a pointer). These assumptions and limitations must be made explicit in order to ensure that error modes ignored by the point solution are either sufficiently rare or handled by another point solution.

We need to develop a *resilience architecture* that specifies (1) which errors are assumed to occur and which errors are assumed to be so rare as to be ignored (e.g., SDCs) and (2) what the division of labor is between the various layers of the system in handling such errors.

As long as we have not converged to one scenario, we will have multiple resilience architectures. But each of them must be brought to a reasonable level of completeness in order to make sure the different approaches are comprehensive.

Acknowledgments

Lead: Marc Snir

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. We thank the U.S. Department of Energy for its financial support of ICiS; the ICiS director and steering committee for the support provided to our workshop; and, in particular, Cheryl Zidel for her outstanding administrative support before, during, and after the workshop. We also thank Gail Pieper for her thorough editing of this report.

A Taxonomy Summary Sheet

See next page

Dependable Supercomputing

This sheet is intended to facilitate clear and consistent communication.

Dependability

This section is based almost entirely on [1].

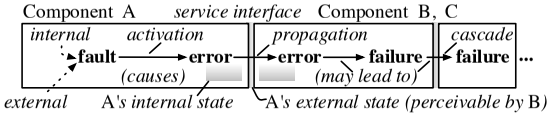
dependability - the ability to avoid service failures that are more frequent and more severe than is acceptable
dependence - the extent to which one system's dependability is affected by another's
trust - accepted dependence

system - an entity that interacts with other entities
component/subsystem - a system which is part of a larger system
atomic component - the point at which system/component recursion stops, by desire or discernability

functional specification - description of system functionality and performance, defining the threshold between *correct* and *incorrect* service (acceptable vs unacceptable)
service - a system's externally-perceived behavior
behavior - what a system does to implement its function, described by a series of states
total state - a system's computation, communication, stored information, interconnection, and physical condition

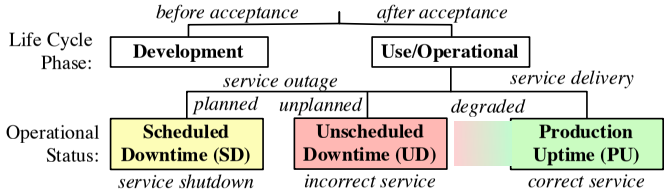
fault - the cause of an error *(e.g. a bug, stuck bit, alpha particle)*
error - the part of total state that *may* lead to a failure *(e.g. a bad value)*
failure - a transition to incorrect service (an event, *e.g. the start of an unplanned service outage*)
degraded mode / partial failure - the failure of a subset of services

Faults typically affect a single component, versus errors which may propagate, and failures which may cascade:



Life Cycle and Operational Status

After acceptance, a system is in one operational status at any time. The below blends [1] and [2], and supersedes [3].



Failure Modes

Domain: *content|timing|both:halt|erratic*
Detectability: *signalled|unsignalled*
Consistency: *consistent|inconsistent*

consistent - perceived identically by all users
inconsistent/Byzantine - perceived differently by different users

Error Characteristics

detected - indicated by error message or signal
latent/silent - not detected
masked - not causing a failure
soft - produced by an intermittent fault

Fault Classes

Creation or occurrence: *development|operational*
Phenomenological: *natural|human-made*
Capability: *accidental|incompetence*
Dimension: *software|hardware/physical*
Interface boundary: *internal|external/interaction*
Persistence: *permanent|transient*

permanent - presence is continuous in time
transient - presence is not continuous in time
intermittent - elusive development or transient physical

Fault Characteristics

active - causing an error
dormant - not causing an error

solid/hard - activation is systematically reproduceable
elusive/soft - activation is not systematically reproduceable

Means of Dealing with Faults

forecasting - to estimate the present number, future incidence, and likely consequences of faults
prevention - to prevent fault occurrence or introduction
removal - to reduce fault number and severity
tolerance - to avoid service failures in the presence of faults

Fault Tolerance Techniques

error detection - identifies the presence of an error

concurrent - occurs during service delivery
preemptive - occurs during planned service outage

recovery - prevents faults from causing failures

error handling - eliminates errors

rollback - revert to previous correct state *(e.g. checkpoint, retry)*
rollforward - state without corrected errors is a new state *(??)*
compensation - correct the error *(e.g. via redundancy)*

fault handling - prevents faults from re-activating

diagnosis - identifies fault location and type
isolation - excludes from interaction with other components
reconfiguration - replace component or move work elsewhere
reinitialization - a pristine reset of state *(e.g. reboot)*

Metrics

If you can not measure it, you can not improve it. - Kelvin

A key metric is the ratio of the ideal time to solution on a fault-free system (T_{solve}) to the actual run time including error detection and recovery ($T_{wallclock}$): **Workload Efficiency** $= \frac{T_{solve}}{T_{wallclock}}$

Full-System

Unscheduled downtime thresholds for the full system should be defined in the functional specification, e.g. the "system" is down when more than 5% of compute nodes are down (e.g. become unavailable with less than 24 hours notice [4]). Consider the below time series of system states, where numbers indicate duration in days:

6	1	2	.3	3.7	1	3	.4	2.6	2

Tabulate the data into sets:

Set X	$\sum X$	$\lvert X\rvert$
PU={6, 2, 3.7, 3, 2.6}	$17.3 = Uptime$	$5 = NumUptimes$
SD={1, 1, 2}	$4 = SDown$	3
UD={.3, .4}	$.7 = UDown$	$2 = NumInterrupts$
	$22 = TotalTime$	

Recommended [5] as a *control* (specified) metric:
Scheduled Availability $= \frac{100*Uptime}{Uptime+UDown} = \frac{100*17.3}{17.3+.7} = 96.1\%$
Recommended as an *observed* metric:
Overall Availability $= \frac{100*Uptime}{TotalTime} = \frac{100*17.3}{22} = 78.6\%$
Using **interrupt** as "a detected failure" (so $MTBI \simeq MTBF$, etc):
Mean Time *Between* Interrupt:
MTBI $= \frac{TotalTime}{NumInterrupts} = \frac{22}{2} = 11\frac{Days}{Interrupt}$
Mean Time *To* Interrupt:
MTTI $= \frac{Uptime}{NumInterrupts} = \frac{17.3}{2} = 8.65\frac{Days}{Interrupt}$
Mean Time To Repair:
MTTR $= \frac{UDown}{NumInterrupts} = \frac{.7}{2} = .35\frac{Days}{Repair}$
Mean Uptime (this accounts for both types of Downtime, unlike MTTI):
MeanUptime $= \frac{Uptime}{NumUptimes} = \frac{17.3}{5} = 3.46\frac{Days}{Uptime}$

Non-Full System

Unique acronyms can increase clarity. For example, SNL prepends an "S" (e.g. $SMTTI$) to distinguish the above from the below [3]:
Job Mean Time To Interrupt, where $NumJobInterrupts$ is the total number of jobs terminated as a result any failure granularity (e.g. full system crash, I/O subsystem failure, node failures, etc):
JMTTI $= \frac{Uptime}{NumJobInterrupts}$
Node Mean Time To Interrupt, where $NumNodeFailures$ is the total number of individual node failures:
NMTTI $= \frac{Uptime}{NumNodeFailures}$

Miscellaneous

exponential failures - time to failure distribution having a CDF of $F(t) = 1 - e^{-t/M}$, where M is the mean time to failure. An empirical study of LANL HPC failure data showed this to fit poorly, whereas gamma or Weibull distributions with decreasing hazard rates (.7-.8) fit well [6]. See http://cfdr.usenix.org for this and other data.
optimal checkpoint interval (τ_{opt}) - the time between checkpoints which maximizes workload efficiency. The below is an approximation, assuming exponential job failures, where δ is the time to write a checkpoint and R is the time to restart from checkpoint [7]:
$\tilde{\tau}_{opt} = \sqrt{2\delta(M+R)} - \delta$ for $\delta < \frac{1}{2}M$, else $\tilde{\tau}_{opt} = M$

References

(in minimal format)

[1] Avižienis et al, TSDC, 2004. [2] SEMI E10-0304, 2004. [3] Stearley, LCI, 2004. [4] Kramer, Chap6, 2008. [5] Bell et al, 2007. [6] Schroeder et al, DSN, 2006. [7] Daly, FGCS, 2006.

National Nuclear Security Administration

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.
This is SNL document number SAND2012-6495P, compiled by Jon Stearley *jrstear@sandia.gov*, version 2012-12-1.

Sandia National Laboratories

B Derivation of Optimal Checkpoint Interval

We assume a global checkpointing model: The system is periodically taking global checkpoints; after a failure, computation is restarted from the last checkpoint. We use the following parameters:

- Checkpoint time is C
- Recovery time is R
- Checkpoint interval is τ: A new checkpoint is taken time τ after the previous checkpoint started, or time τ after a failure occured.
- Probability of failure within a time interval τ is $F(\tau)$
- Time to first failure, *given that a failure occurs* within the interval τ is $W(\tau)$.

We assume that C, R, and τ are constant, while $W(\tau)$ is a random variable. We further assume that the system is *memoryless:* $F(\tau)$ and $W(\tau)$ are the same, for each time interval.

We divide the computation into *epochs*: A new epoch starts when a failure occurred, or when a checkpoint completed. Let $Comp_i$ be the amount of useful computation done in epoch i and let $Time_i$ be the amount of wallclock time consumed by epoch i. $Comp_i$ are i.i.d. random variables and $Time_i$ are i.i.d. random variables ($Comp_i$ and $Time_i$ are not independent).

We have

$$Comp = \begin{cases} \tau - C & \text{if epoch completes normally} \\ -R & \text{otherwise} \end{cases} \tag{1}$$

The $-R$ represents the fact that not only no progress was done, but the computation now requires recovery. Also

$$Time = \begin{cases} \tau & \text{if epoch completes normally} \\ W(\tau) & \text{otherwise} \end{cases} \tag{2}$$

We define the *Utilization* of the system to be ratio between compute time and wall-clock time:

$$Util = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} Comp_i}{\sum_{i=1}^{n} Time_i}$$

We have $\frac{1}{n}\sum_{i=1}^{n} Comp_i \to \mathrm{E}[Comp]$ and $\frac{1}{n}\sum_{i=1}^{n} Time_i \to \mathrm{E}[Time]$, so that

$$Util = \frac{\mathrm{E}[Comp]}{\mathrm{E}[Time]} \tag{3}$$

We derive, from Equations 1 and 2,

$$\mathrm{E}[Comp] = (1 - F(\tau))(\tau - C) - F(\tau)R$$

and

$$\mathrm{E}[Time] = (1 - F(\tau))\tau + F(\tau)\mathrm{E}[w]$$

so that

$$Util = \frac{(1-p)(\tau - C) - pR}{(1-p)\tau + p\mathrm{E}[W(\tau)]} \tag{4}$$

Note that this formula does not involve any approximation and does not depend on the distribution of between-failure intervals.

We shall assume from now on that failures occur according to a Poisson process, and normalize time so that MTTF equals to 1 (i.e., we express checkpoint time and recovery time as fractions of MMTF). Thus,

$$F(\tau) = 1 - e^{-\tau}$$

and

$$\mathrm{E}[w] = \frac{1}{p}\int_0^\tau xe^{-x}dx$$

But

$$\int_0^\tau xe^{-x}dx = -(x+1)e^{-x/1}|_0^\tau = -(\tau+1)e^{-\tau} + 1$$

Thus

$$\mathrm{E}[CT] = e^{-\tau}(\tau - C) - (1 - e^{-\tau})R = (\tau - C + R)e^{-\tau} - R,$$

$$\mathrm{E}[WT] = \tau e^{-\tau} - (\tau+1)e^{-\tau} + 1 = 1 - e^{-\tau},$$

and

$$Util = \frac{(\tau - C + R)e^{-\tau} - R}{1 - e^{-\tau}}.$$

We want to select τ that maximizes utilization. Such τ solves the equation $\frac{dUtil}{d\tau} = 0$.
We compute derivatives and obtain the equation

$$(e^{-\tau} - (\tau - C + R)e^{-\tau})(1 - e^{-\tau}) - ((\tau - C + R)e^{-\tau} - R))e^{-\tau} = 0$$

Simplifying, we obtain the equation

$$e^{-\tau} = 1 - \tau + C \tag{5}$$

and

$$Util = \frac{(\tau - C + R)(1 - \tau + C) - R}{\tau - C} = 1 - \tau + C - R \tag{6}$$

We solve Equation 5 numerically, for different values of M and R, and plug into Equation 6 in order to compute the best possible utilization, as a function of (relative) MTTI and recovery time.

Various approximations can be derived from Equation 5: If we approximate e^x with the first three terms of its Taylor expansion, then we get

$$1 - \tau - \frac{\tau^2}{2} = 1 - \tau + C \tag{7}$$

so that $\tau_{opt} = \sqrt{2C}$ and $Util = 1 - \sqrt{2C} + C - R$

If the MTBF is M (rather than 1), we get
$\tau_{opt} = M\sqrt{2C/M} = \sqrt{2CM}$ and $Util = 1 - \sqrt{2CM} + (C - R)/M$.
The approximation is valid when $C \ll M$ [153]. Higher level approximations are derived in [30].

References

[1] Community Earth System Model.

[2] Scope of work and technical specification (for the Cielo supercomputer). Technical Report Subcontract 65483-001-10 Exhibit D, Los Alamos National Laboratory, February 2010.

[3] M. Agostinelli, S. Pae, W. Yang, C. Prasad, D. Kencke, S. Ramey, E. Snyder, S. Kashyap, and M. Jones. Random charge effects for PMOS NBTI in ultra-small gate area devices. In *Reliability Physics Symposium, 2005. Proceedings. 43rd Annual. 2005 IEEE International*, pages 529–532. IEEE, 2005.

[4] Dong H. Ahn, Bronis R. De Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. Scalable Temporal Order Analysis for Large Scale Debugging. In *SC '09*, 2009.

[5] W. Allcock. Private Communication, 2013.

[6] Andre DeHon Nick Carter Heather Quinn. Final report for "CCC" cross-layer reliability visioning study. Technical report, Computing Community Consortium (CCC), 2011.

[7] Todd M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207, 1999.

[8] A. Avizienis. Arithmetic algorithms for error-coded operands. *IEEE Transactions on Computers*, C-22(6):567 –572, 1973.

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

[10] A. Avritzer, A. Bondi, M. Grottke, K.S. Trivedi, and E.J. Weyuker. Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms. In *International Conference on Dependable Systems and Networks (DSN)*, pages 435–444, 2006.

[11] Ronald Bailey, Gordon Bell (Chair), John Blondin, John Connolly, David Dean, Peter Freeman, James Hack (co chair), Steven Pieper, Douglass Post, and Steven Wolff. Petascale metrics report. Technical report, Department of Energy Office of Scient Advanced Scientific Computing Advisory Committee, 2007.

[12] P. Banerjee and J.A. Abraham. Bounds on algorithm-based fault tolerance in multiple processor systems. *Computers, IEEE Transactions on*, C-35(4):296 –306, April 1986.

[13] P. Banerjee, J.T. Rahmeh, C. Stunkel, V.S. Nair, K. Roy, V. Balasubramanian, and J.A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *Computers, IEEE Transactions on*, 39(9):1132 –1145, sep 1990.

[14] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[15] Leonardo Arturo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *SC*, page 32, 2011.

[16] J.R. Birge and F. Louveaux. *Introduction to stochastic programming*. Springer Verlag, 1997.

[17] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[18] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, et al. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29. IEEE, 2002.

[19] George Bosilca, Rmi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.

[20] A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Correlated set coordination in fault tolerant message logging protocols. *Euro-Par 2011 Parallel Processing*, pages 51–64, 2011.

[21] Fred Bower, Daniel Sorin, and Sule Ozev. Online Diagnosis of Hard Faults in Microprocessors. *ACM Trans. on Architecture and Code Optimization*, 4(2), 2007.

[22] G. Bronevetsky, I. Laguna, S. Bagchi, B.R. de Supinski, D.H. Ahn, and M. Schulz. AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, pages 231–240, 2010.

[23] F. Cappello, A. Guermouche, and M. 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.

[24] JM Carulli and T.J. Anderson. Test connections-tying application to process. In *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, page 8 pp. IEEE, 2005.

[25] Vittorio Castelli, Richard E Harper, Philip Heidelberger, Steven W Hunter, Kishor S Trivedi, Kalyanaraman Vaidyanathan, and William P Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 45(2):311–332, 2001.

[26] Zizhong Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 10 pp., April 2006.

[27] P.L. Chow. *Stochastic partial differential equations*, volume 11. Chapman & Hall/CRC, 2007.

[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] A.R. Conn, N.I.M. Gould, and P.L. Toint. *Trust region methods*, volume 1. Society for Industrial Mathematics, 1987.

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

[31] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[32] N. DeBardeleben, J. Daly, S. Scott, and W. Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path forward for research and development. In *2009 National HPC Workshop on Resilience*, 2009.

[33] 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.

[34] Martin Dimitrov and Huiyang Zhou. In *Parallel Architecture and Compilation Techniques), year=2007, pages = 73–82, title=Unified Architectural Support for Soft-Error Protection or Software Bug Detection*.

[35] Anand Dixit, Raymond Heald, and Alan Wood. Trends from ten years of soft error experimentation. In *the Workshop on Silicon Errors in Logic-System Effects (SELSE)*, March 2009.

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

[37] 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. Technical report, February 2012.

[38] 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, 2009.

[39] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[40] E.N.M. Elnozahy, R. Bianchini, T. El-Ghazawi, A. Fox, F. Godfrey, A. Hoisie, K. McKinley, R. Melhem, JS Plank, P. Ranganathan, et al. System resilience at extreme scale. Technical report, Defense Advanced Research Project Agency (DARPA), 2008.

[41] Michael D. Ernst et al. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 2007.

[42] D. Chen et al. The IBM Blue Gene/Q interconnection fabric. *IEEE Micro*, 32(1):32–42, January February 2012.

[43] Scott Fadden. An introduction to gpfs version 3.5, 2012.

[44] Graham Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, 2000.

[45] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 385–396, 2010.

[46] Kurt B. Ferreira, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin T. Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *SC*, page 44, 2011.

[47] R. Fletcher. Practical methods of optimization: Vol. 2: Constrained optimization. *John Wiley & Sons*, 1981.

[48] Hajime Fujita, Robert Schreiber, and Andrew A. Chien. Its time for new programming models for unreliable hardware. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013. Provocative Ideas Session.

[49] Ana Gainaru, Franck Cappello, Joshi Fullop, Stefan Trausan-Matu, and William Kramer. Adaptive event prediction strategy with dynamic time window for large-scale hpc systems. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, SLAML '11, pages 4:1–4:8, New York, 2011. ACM.

[50] Ana Gainaru, Franck Cappello, and William Kramer. Taming of the shrew: Modeling the normal and faulty behavior of large-scale hpc systems. In *Proceedings of IEEE IPDPS 2012*. IEEE press, 2012.

[51] Ana Gainaru, Franck Cappello, Marc Snir, and William Kramer. Fault prediction under the microscope: A closer look into hpc systems. In *Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE press, 2012.

[52] B. Gao, H. Zhang, B. Chen, L. Liu, X. Liu, R. Han, J. Kang, Z. Fang, H. Yu, B. Yu, et al. Modeling of retention failure behavior in bipolar oxide-based resistive switching memory. *Electron Device Letters, IEEE*, 32(3):276–278, 2011.

[53] Qi Gao, Feng Qin, and Dhabaleswar K. Panda. DMTracker: Finding Bugs in Large-scale Parallel Programs by Detecting Anomaly in Data Movements. In *ACM/IEEE Supercomputing Conference (SC)*, 2007.

[54] Qi Gao, Wenbin Zhang, and Feng Qin. FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking. In *ACM/IEEE Supercomputing Conference (SC)*, 2010.

[55] A. Gattiker, P. Nigh, D. Grosch, and W. Maly. Current signatures for production testing [cmos ics]. In *IDDQ Testing, 1996., IEEE International Workshop on*, pages 25–28. IEEE, 1996.

[56] 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 (ANL) Larry Kaplan (Cray) 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.

[57] B. Gill, N. Seifert, and V. Zia. Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node. In *Reliability Physics Symposium, 2009 IEEE International*, pages 199–205, April 2009.

[58] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-Error Detection Using Control Flow Assertions. In *Default and Fault Tolerance in VLSI Systems*, pages 581–588, 2003.

[59] A. Griewank and G.F. Corliss. *Automatic differentiation of algorithms: theory, implementation, and application*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991.

[60] Michael Grottke and Kishor S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *IEEE Computer*, 40(2):107–109, 2007.

[61] "Network Working Group". The syslog protocol, 2009.

[62] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 989–1000. IEEE, 2011.

[63] A. Guermouche, T. Ropars, M. Snir, and F. Cappello. Hydee: Failure containment without event logging for large scale send-deterministic MPI applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1216–1227. IEEE, 2012.

[64] W. Hackbusch. *Multi-grid methods and applications*, volume 2. Springer-Verlag Berlin, 1985.

[65] James L Hafner, Veera Deenadhayalan, Wendy Belluomini, and Krishnakumar Rao. Undetected disk errors in RAID arrays. *IBM Journal of Research and Development*, 52(4.5):413–425, 2008.

[66] Richard Hamming. *Numerical methods for scientists and engineers: Chapter 1*. Dover Publications, 1987.

[67] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. 2002.

[68] H. Hao and E.J. McCluskey. Very-low-voltage testing for weak CMOS logic ICs. In *Test Conference, 1993. Proceedings*, pages 275–284. IEEE, 1993.

[69] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. Low-cost Program-level Detectors for Reducing Silent Data Corruptions. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012.

[70] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[71] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. mSWAT: Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 122–132, 2009.

[72] P. Hazucha, T. Karnik, S. Walstraand B. Bloechel, J. Tschanzand J. Maiz, K. Soumyanath, G. Dermer, S. Narendra, V. De, and S. Borkar. Measurements and analysis of SER tolerant latch in a 90 nm dual-Vt CMOS process. In *2003 IEEE Custom Integrated Circuits Conference*, pages 617–620, September 2003.

[73] Richard Hedges, Bill Loewe, Tyce McLarty, and Chris Morrone. Parallel file system testing for the lunatic fringe: The care and feeding of restless I/O power users. In *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on*, pages 3–17. IEEE, 2005.

[74] Eric Heien, Derrick Kondo, Ana Gainaru, Dan LaPine, Bill Kramer, and Franck Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 45:1–45:11, New York, 2011. ACM.

[75] Wilmot N Hess, H Wade Patterson, Roger Wallace, and Edward L Chupp. Cosmic-ray neutron energy spectrum. *Physical Review*, 116(2):445, 1959.

[76] Sean Hogan, Jeff Hammond, and Andrew A. Chien. An evaluation of difference and threshold techniques for efficient checkpointing. In *2nd workshop on fault-tolerance for HPC at extreme scale (FTXS 2012)*, 2012.

[77] Kuang-Hua Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[78] R.C. Hunter and C.Eng. Engine Failure Prediction Techniques. *Aircraft Engineering and Aerospace Technology*, 47(3):4–14, 1975.

[79] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 111–122, March 2012.

[80] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design. *SIGARCH Comput. Archit. News*, 40(1):111–122, March 2012.

[81] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba. Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. *IEEE Transactions on Electron Devices*, 57(7):1527 –1538, July 2010.

[82] Data Storage Institute. First patent on low density parity check coding with soft decision decoding for spin-torque transfer magnetic random access memory, March 2012.

[83] D. Kerbyson, R. Rajamony, and E. Van Hensbergen. Performance health monitoring for large-scale systems. In *Second International Workshop on High-performance Infrastructure for Scalable Tools*, 2012.

[84] S. Kundu, TM Mak, and R. Galivanche. Trends in manufacturing test methods and their implications. In *Test Conference, 2004. Proceedings. ITC 2004. International*, pages 679–687. IEEE, 2004.

[85] Ignacio Laguna, Dong Ahn, Saurabh Bagchi, Bronis R de Supinski, and Todd Gamblin. Probabilistic diagnosis of performance faults in large-scale parallel applications. In *submission to the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–10, 2012.

[86] Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Anh, Martin Schulz, and Barry Rountree. Large scale debugging of parallel tasks with AutomaDeD. In *ACM/IEEE Supercomputing Conference (SC)*, pages 50:1–50:10, 2011.

[87] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, et al. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[88] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit. Lessons Learned at 208K: Towards Debugging Millions of Cores. In *ACM/IEEE Supercomputing Conference (SC)*, pages 1–9. IEEE Press, 2008.

[89] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Barton P. Miller, and Martin Schulz. Benchmarking the Stack Trace Analysis Tool for BlueGene/L. In *International Conference on Parallel Computing: Architectures, Algorithms and Applications (ParCo)*, 2007.

[90] Man-Lap Li, Pradeep Ramachandran, Swarup Sahoo, Sarita V. Adve, Vikram Adve, and Yuanyuan Zhou. Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2008.

[91] Man-Lap Li, Pradeep Ramachandran, Swarup Sahoo, Sarita V. Adve, Vikram Adve, and Yuanyuan Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *Proc. of the Intl. Conf. Architectural support for programming languages and operating systems*, pages 265–276, 2008.

[92] Karl Lindekugel, Anthony DiGirolamo, and Dan Stanzione. Architecture for an offline parallel debugger. In *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*, pages 227–235. IEEE, 2008.

[93] J. Linderoth and S. Wright. Decomposition algorithms for stochastic programming on a computational grid. *Computational Optimization and Applications*, 24(2):207–250, 2003.

[94] J.C. Lo, S. Thanawastien, and T.R.N. Rao. Concurrent error detection in arithmetic and logical operations using Berger codes. In *Proceedings of 9th Symposium on Computer Arithmetic*, pages 233 –240, September 1989.

[95] Jien-Chung Lo. Reliable floating-point arithmetic algorithms for error-coded operands. *Computers, IEEE Transactions on*, 43(4):400 –412, apr. 1994.

[96] Los Alamos National Lab. Operational Data to Support and Enable Computer Science Research. http://institutes.lanl.gov/data/fdata/.

[97] João Lourenço and José Cunha. Fiddle: A flexible distributed debugger architecture. *Computational Science-ICCS 2001*, pages 821–830, 2001.

[98] Guoming Lu, Ziming Zheng, and Andrew A. Chien. When are multiple checkpoints needed? In *2nd workshop on fault-tolerance for HPC at extreme scale (FTXS 2013)*, 2013.

[99] D. Lunardini, B. Narasimham, V. Ramachandran, V. Srinivasan, R. D. Schrimpf, and W. H. Robinson. A performance comparison between hardened-by-design and conventional-design standard cells. In *2004 Workshop on Radiation Effects on Components and Systems, Radiation Hardening Techniques and New Developments*, September 2004.

[100] G. Lyle, S. Cheny, K. Pattabiraman, Z. Kalbarczyk, and R.K. Iyer. An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 584–589, 2009.

[101] Albert Meixner, Michael E. Bauer, and Daniel J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the Annual International Symposium on Microarchitecture (MICRO)*, pages 210–222, 2007.

[102] Ron Minnich, Noah Evans, Enrique Soriano, Francisco Ballesteros, Jim McKie, Gorka Guardiola, and Charles Forsyth. High performance cloud computing is NIX. *Bell Labs Technical Conference*, 2011.

[103] Alexander V. Mirgorodskiy, Naoya Maruyama, and Barton P. Miller. Problem Diagnosis in Large-Scale Computing Environments. In *ACM/IEEE Supercomputing Conference (SC)*, New York, 2006. ACM.

[104] Subhasish Mitra, Ming Zhang, Norbert Seifert, T. M. Mak, and Keesup Sup Kim. Built-In Soft Error Resilience for Robust System Design. In *IEEE Internaional Conference on Intefrated Circuit Design and Technology*.

[105] Akbar Mokhtarani, William Kramer, and Jason Hick. Reliability results of NERSC systems. http://escholarship.org/uc/item/890104qh, 2008.

[106] A. Moody, G. Bronevetsky, K. Mohror, and B.R. De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[107] Jorge J. Moré and Stefan M. Wild. Estimating derivatives of noisy simulations. *ACM Trans. Math. Softw.*, 38(3):19:1–19:21, April 2012.

[108] MPIPlugIn. MPI Plugin for KDevelop. http://sourceforge.net/projects/mpiplugin/.

[109] Jun Nakano, Pablo Montesinos, Kourosh Gharachorloo, and Josep Torrellas. ReVive I/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2006.

[110] N. Naksinehaboon, N. Taerat, C. Leangsuksun, C.F. Chandler, and S.L. Scott. Benefits of software rejuvenation on HPC systems. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 499–506, 2010.

[111] S.R. Nassif, V.B. Kleeberger, and U. Schlichtmann. Goldilocks failures: not too soft, not too hard. In *Reliability Physics Symposium (IRPS), 2012 IEEE International*, pages 2F–1. IEEE, 2012.

[112] John Daly Stephen Scott Christian Enbelmann Bill Harrod Nathan DeBardeleben, James Laros. High-End Computing Resilience: Analysis of Issues Facing the HEC Community and PathForward for Research and Development. Technical report, January 2010.

[113] P. Nigh and A. Gattiker. Test method evaluation experiments and data. In *Test Conference, 2000. Proceedings. International*, pages 454–463. IEEE, 2000.

[114] Jutaek Oh, Simon P Washington, and Doohee Nam. Accident prediction model for railway-highway interfaces. *Accident analysis and prevention*, 38(2):346–356, 2006.

[115] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 575–584. IEEE, 2007.

[116] Y. Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. Dong Ryu, and R. Wisniewski. FusedOS: Fusing LWK performance with FWK functionality in a heterogeneous environment. In *24th International Symposium on Computer Architecture and High Performance Computing*, 2012.

[117] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2008.

[118] Karthik Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *European Dependable Computing Conference*, pages 97–108, 2006.

[119] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: Cost-Effective Arch Support for Rollback Recovery in Shared-Mem Multiprocessors. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2002.

[120] Paul Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. Perturbation-based Fault Screening. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 169–180, 2007.

[121] Pradeep Ramachandran. *Detecting and Recovering from In-Core Hardware Faults Through Software Anomaly Treatment*. PhD thesis, University of Illinois at Urbana Champaign, 2011.

[122] Thammavarapu R. N. Rao. *Error Coding for Arithmetic Processors*. Academic Press, Inc., Orlando, FL, USA, 1974.

[123] V. Reddy, A.T. Krishnan, A. Marshall, J. Rodriguez, S. Natarajan, T. Rost, and S. Krishnan. Impact of negative bias temperature instability on digital circuit reliability. *Microelectronics Reliability*, 45(1):31–38, 2005.

[124] George Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-Controlled Fault Tolerance. *ACM Trans. on Architecture and Code Optimization*, 2(4), 2005.

[125] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. SWIFT: Software implemented fault tolerance. In *Proceedings of the international symposium on Code generation and optimization*, pages 243–254. IEEE Computer Society, 2005.

[126] Rogue Wave Software. TotalView Debugger. http://www.roguewave.com/products/totalview.aspx.

[127] T. Ropars, A. Guermouche, B. Uçar, E. Meneses, L. Kalé, and F. Cappello. On the use of cluster-based partial message logging to improve fault tolerance for mpi hpc applications. *Euro-Par 2011 Parallel Processing*, pages 567–578, 2011.

[128] Thomas Ropars, Amina Guermouche, Bora Uçar, Esteban Meneses, Laxmikant V. Kalé, and Franck Cappello. On the use of cluster-based partial message logging to improve fault tolerance for MPI hpc applications. In *Euro-Par (1)*, pages 567–578, 2011.

[129] Philip C Roth, Dorian C Arnold, and Barton P Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, pages 1–16. ACM, 2003.

[130] Amber Roy-Chowdhury, Nikolas Bellas, and Prithviraj Banerjee. Algorithm-based error-detection schemes for iterative solution of partial differential equations. *IEEE Transactions on Computers*, 45(4):394–407, 1996.

[131] Swarup Sahoo, Man-Lap Li, Pradeep Ramchandran, Sarita V. Adve, Vikram Adve, and Yuanyuan Zhou. Using Likely Program Invariants to Detect Hardware Errors. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 70–79, 2008.

[132] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys*, 42:1–42, 2010.

[133] N.R. Saxena and E.J. McCluskey. Dependable adaptive computing systems - the roar project. In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, volume 3, pages 2172–2177. IEEE, 2002.

[134] Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, 2007.

[135] Bianca Schroeder and Garth A Gibson. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):337–350, 2010.

[136] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 193–204. ACM, 2009.

[137] Per Seltborg, A. Polanski, S. Petrochenkov, A. Lopatkin, Waclaw Gudowski, and V. Shvetsov. Radiation shielding of high-energy neutrons in SAD. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 550(1):313–328, 2005.

[138] Galen Shipman, David Dillow, Sarp Oral, Feiyi Wang, Douglas Fuller, Jason Hill, and Zhe Zhang. Lessons learned in deploying the worlds largest scale lustre file system. In *The 52nd Cray User Group Conference*, 2010.

[139] C. Slayman. Impact and mitigation of DRAM and SRAM soft errors. IEEE SCV Reliability Seminar http://www.ewh.ieee.org/r6/scv/rl/articles/Soft%20Error%20mitigation.pdf, May 2010.

[140] Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[141] Marc Snir and David A. Bader. A framework for measuring supercomputer productivity. *International Journal for High Performance Computing Applications*, (18):399–416, 2004.

[142] Daniel Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2002.

[143] 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.

[144] Vilas Sridharan and Dean Liberty. A Study of DRAM Failures in the Field. In *SC '12 Proceedings of the International Conference on High Peformance Computing, Networking, Storage and Analysis*, November 2012.

[145] Jon Stearley. Defining and measuring supercomputer reliability, availability, and serviceability (RAS). In *Proceedings of the Linux Clusters Institute Conference*, 2005.

[146] N.N. Taleb. *The black swan: The impact of the highly improbable*. Random House Trade Paperbacks, 2010.

[147] U. Trottenberg, C.W. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[148] Alexander Randall V. The Eckert tapes: Computer pioneer says ENIAC team couldn't afford to fail - and didn't. *Computerworld*, (40:8):18, 2006.

[149] J. Van Horn. Towards achieving relentless reliability gains in a server marketplace of teraflops, laptops, kilowatts, and. In *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, pages 8–pp. IEEE, 2005.

[150] M. von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor os kernels. 2012.

[151] N.J. Wang and S.J. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3), July-Sept 2006.

[152] J.J. Yang, M.X. Zhang, J.P. Strachan, F. Miao, M.D. Pickett, R.D. Kelley, G. Medeiros-Ribeiro, and R.S. Williams. High switching endurance in TaOx memristive devices. *Applied Physics Letters*, 97(23):232102–232102, 2010.

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

[154] Jing Yu, Maria Jesus Garzaran, and Marc Snir. Esoftcheck: Removal of non-vital checks for fault tolerance. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 35–46. IEEE Computer Society, 2009.

[155] S. Yu, Y. Yin Chen, X. Guan, H.S. Philip Wong, and J.A. Kittl. A Monte Carlo study of the low resistance state retention of HfOx based resistive switching memory. *Applied Physics Letters*, 100(4):043507–043507, 2012.

[156] Ming Zhang, Subhasish Mitra, T. M. Mak, Norbert Seifert, Nicholas J. Wan, Quan Shi, Keesup Sup Kim, Naresh R. Shanbhag, and Sanjay Jeram Patel. Sequential Element Design with Built-In Soft Error Resilience. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(13):1368–1378.

[157] G. Zheng, X. Ni, and L.V. Kalé. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6. IEEE, 2012.

[158] Jie Zhou, Mingxiang Wang, and Man Wong. Instability of p-channel poly-Si thin-film transistors under dynamic negative bias temperature stress. In *Physical and Failure Analysis of Integrated Circuits (IPFA), 2010 17th IEEE International Symposium on the*, pages 1–4. IEEE, 2010.

[159] Enrico Zio, Francesco Di Maio, and Marco Stasi. A data-driven approach for predicting failure scenarios in nuclear systems. *Annals of Nuclear Energy*, 37:482–491, 2010.