Inter-Agency Workshop on HPC Resilience at Extreme Scale

National Security Agency
Advanced Computing Systems
February 21 – 24, 2012

Coordinating Representatives
John Daly (DOD)
Bill Harrod (DOE/SC)
Thuc Hoang (DOE/NNSA)
Lucy Nowell (DOE/SC)

Workshop Report Committee
Bob Adolf (PNNL)
Shekhar Borkar (Intel)
Nathan DeBardeleben (LANL)
Mootaz Elnozahy (IBM)
Mike Heroux (Sandia)
David Rogers (Sandia)
Rob Ross (ANL)
Vivek Sarkar (Rice U.)
Martin Schulz (LLNL)
Marc Snir (DOE/ANL)
Paul Woodward (UMN)

Report Editor
John Daly (DOD)

Additional Workshop Participants
Rob Aulwes (LANL)
Marti Bancroft (MBC)
Greg Bronevetsky (LLNL)
Bill Carlson (IDA)
Al Geist (ORNL)
Mary Hall (U. Utah)
Jeff Hollingsworth (UMD)
Bob Lucas (USC/ISI)
Andrew Lumsdaine (Indiana U.)
Tina Macaluso (SAIC)
Dan Quinlan (LLNL)
Sonia Sachs (DOE/SC)
John Shalf (LBNL)
Tom Smith (DOD)
Jon Stearley (Sandia)
Bert Still (LLNL)
Jon Wu (LBNL)

Abstract

The following report summarizes the proceedings of a three-and-a-half day inter-agency workshop focused on the technical challenges of HPC resilience in the 2020 Exascale timeframe. The resilience problem is not specific to any particular program or agency; coordinated resilience solutions will be challenging because of the need for a truly integrated approach. The inter-agency workshop therefore focused on articulating practical, synergetic R&D goals by assembling a small but diverse group of experts representing system hardware, system software, application developers and users, algorithms and libraries, file systems, I/O and storage, visualization and data analytics for a collective deep dive on the problem of resilience. The workshop format was highly interactive, focused on problem solving teams of not more than ten persons each. Participants were tasked to collaboratively develop a plan and roadmap for implementing resilience at extreme scale, resulting in "proof of concept" strategies for resilience on future, general purpose HPC systems in the application domains of "predictive science" and "not predictive science". Those strategies were analyzed in the context of future Exascale requirements relative to power, performance, reliability, usability, dependability and time-to-solution. That analysis consisted of an assessment of current capabilities, gaps and dependencies culminating in a strawman R&D roadmap for an integrated resilience framework. These outcomes demonstrate both the need for and existence of practical resilience strategies that address the future needs of applications within the constraints of future Exascale technology.

Executive Summary

Resilience is about keeping the application workload running to a correct solution in a timely and efficient manner in spite of frequent hard (i.e., unrecoverable) and soft (i.e., recoverable) errors. At this inter-agency resilience workshop hosted by NSA Advanced Computing Systems, DOE/SC and DOE/NNSA, thirty representatives of DOD, DOE, the national labs, industry and academia came together to discuss the primary challenges facing resilience as we look down the road to general purpose HPC in the Exascale timeframe. Workshop participants identified the following topic areas as high priorities:

Fault Characterization: Reliability will get worse with deeply scaled process technologies creating new modes of failure. Based on anticipated technology trends, the HPC community needs to develop a useful taxonomy for describing the types of faults that future systems are expected to encounter, including their anticipated frequency and impact.

Detection: In the Exascale timeframe, error "recovery" will likely be manageable using known techniques for local checkpointing. Error "prediction" can reduce the frequency of permanent errors, not transients, but there are varying opinions on the current state of the research and whether or not the problem is solvable in the Exascale timeframe. The research focus should be error "detection" which requires the system and application to work together in a coordinated fashion. Industry is not going to solve this problem for the HPC community.

Fault-Tolerant Algorithms: Three classes of algorithms were identified: (A) those that are embarrassingly fault-tolerant, (B) those that are not fault-tolerant but are self-checking and (C) those that are neither fault-tolerant nor self-checking. Most algorithms currently in class C above could be moved to class B or even class A through a moderate R&D investment.

Fault-Tolerant Programming Models: Resilience would benefit strongly from a programming model that accommodates some notion of transactions in time (e.g., roll-back and recovery) and space (e.g., fault containment domains). An uncomplicated, directive-based interface using a handful of assertions (e.g., create persistent memory domains, allocate "reliable" and "unreliable" code regions, etc.) provides most of the necessary interfaces for implementing application fault-tolerance.

Fault-Tolerant System Services: System software must be built hierarchically on a small set of highly trusted services. Techniques for fault-avoidance are known and used in critical systems, but come at a higher development and execution cost. The software stack should be structured to utilize the majority of application and system execution cycles in software that may occasionally fail and rely on trusted services to recover.

Tools: Resilience lacks a mature, validated test infrastructure to verify the effectiveness of various resilience strategies for keeping the application running in the face of high rates of hard and soft errors. Fault injection tools are particularly needed to simulate all classes of faults. Models will be required to support the fault testing infrastructure at scale. In addition, tools themselves will need their own resilience strategy to operate correctly at scale.

In summary, the number of errors, particularly soft errors, occurring on HPC systems will continue to increase. A right-sized and well-conceived resilience strategy in the Exascale timeframe will be ultimately far more cost effective for HPC than continuing to rely on ad-hoc resilience solutions. That strategy must at a minimum provide for a resilience infrastructure that facilitates (1) system management of hard errors, by effectively "converting" them to soft errors whenever feasible, and (2) application management of soft errors, through interfaces that allow it simple controls over how and when to respond to errors. Such a framework, based on timely and coordinated error detection and recovery, can serve as the foundation of a deployable and sustainable HPC resilience strategy in the Exascale timeframe.

Contents

1 Introduction

1.1 Motivation for the Workshop

Resilience is about keeping the application workload running to a correct solution in a timely and efficient manner in spite of system failures. Future extreme scale supercomputers are likely to suffer more frequent failures than current systems: As devices scale, they are more susceptible to upsets due to radiation and to errors due to manufacturing variances. The probability of multiple bit upsets is growing, since an event is increasingly likely to impact multiple nearby cells. The use of near-threshold voltage in order to reduce power consumption also increases error rates. Thus, we can expect more frequent hardware failures, and a significant rate of undetected soft errors.

1.1.1 Hardware Considerations

Techniques do exist to develop more reliable hardware components – but such techniques have a cost, and this cost will increase. It is far from obvious that the commodity market will provide cheap, highly reliable components that can be used in future supercomputers. Components with lower reliability will be cheaper and will be "good enough" for most markets such as personal and mobile computing and large clouds. The market niches that require highly reliable systems – e.g., financial transactions or critical systems – will pay an increasing price premium for high reliability or will learn how to build highly reliable systems from unreliable components. This "low-end disruption" [8] has already started, with a large price differential between high volume commodity processors and "mainframe quality" servers.

1.1.2 Software Considerations

In addition, supercomputing software is becoming more complex. This software serves a narrow market and is hard to test at scale; hence, it is reasonable to assume it has a higher defect rate than similar commercial software, and is more failure-prone. Indeed, a large fraction of failures in current supercomputers is due to failures in the parallel file system and other complex software subsystems. Additionally, the application is becoming more dependent on multiple levels of the software stack including libraries and the runtime to support resilience. An error that kills the communications library kills the applications depending on it, whether or not the actual error was in fact "fatal" to the application.

1.1.3 The Role of System Scale

The HPC community appears to be learning how to cope with increasing failure rates and, quite possibly, with a high incidence of undetected soft errors. Our current systems are not ready for such a change. As an example, in a system where global checkpoint and restart take 20 minutes, and where a component fails once an hour, more than 60% of the system time would be spent checkpointing and restarting [9]. Faster checkpointing (e.g., using non-volatile memory) would help, but would come at a nontrivial cost; faster checkpointing would also increase the likelihood that errors occurring before checkpointing would be detected only after checkpointing. Finally, none of the currently used techniques handle undetected soft errors.

Resilience at Extreme Scale (RES) is the problem of designing extreme scale supercomputers that provide useful answers with acceptable time and power consumption, from hardware and software components that are failure-prone. This is a cross-cutting problem: Errors in hardware can corrupt application data, application code, system data or system code. Detection of and recovery from errors may happen at different layers of the system stack, with different costs and different implications to hardware and software designers; recovery is likely to require a coordinated action of many of these layers.

On the other hand, research on resilience is very compartmentalized: Research on resilient hardware assumes that error detection and/or correction is fully contained at the hardware level; research on resilient algorithms focus on application data corruptions but ignore any other type of errors; etc. A piecemeal approach will not produce a solution to the resilience problem at extreme scale. Instead, the HPC community needs to take a global view of the problem and on the possible ways of handling detection and correction in a coordinated manner, at different levels of the system.

1.2 Vision for the Workshop

The Inter-Agency Workshop on HPC Resilience at Extreme Scale, that took place at the Center for Exceptional Computing, Catonsville, MD, February 21-24, 2012 brought together a small group of experts from multiple disciplines to start developing this roadmap and a strategy for R&D that will lead to timely solutions to the RES challenge. The workshop was not intended to be the last word in resilience, nor was it intended to exhaust the resilience challenges in the Exascale timeframe. Rather, the vision for the workshop was to demonstrate both the necessity and sufficiency of a coordinated resilience strategy. We hope that the results reported here can serve as a "stake in the ground", a useful first attempt to define a practical resilience framework for the Exascale timeframe.

1.2.1 The Nature of the Challenge

The RES problem matters to mission agencies – in particular DOD and DOE. Its solution will come from an R&D program coordinated across multiple agencies. Such a program needs to be informed by a proposed roadmap toward RES, an identification of the main roadblocks, and a focused list of problems that need to be solved. Solutions can be incremental, encompassing an increasing number of applications and reducing, over time, the human effort needed to achieve RES.

Thus, attendees were chosen to span the RES space as broadly as possible while still keeping the working group sizes manageable (ten people or fewer) and the discussions focused. Participants were instructed to concentrate on requirements, not existing solutions. They were explicitly told that their current research interests were "off limits" for discussion. Participants stepped through the process of articulating a cogent RES strategy and asked to distinguish "what needs to be done" from "what could be done" or what constitutes an "interesting" research question.

1.2.2 Workshop Structure

The format of this report reflects the overall structure of the workshop. The technical challenges of resilience in the Exascale timeframe, described in Section 2, were addressed first via a series of overview talks. Next, domain experts were asked to summarize their requirements and capabilities with regards to resilience: what they need from resilience and what they themselves can provide. These discussions are summarized in Section 3.

Working from their knowledge of the salient architectural features of the system hardware and software, working groups developed an actual implementation that integrates target application domains into a notional system stack. Working groups were given tasking and specific assignments focused around developing a resilience framework for the Exascale timeframe, summarized in Section 4.They were encouraged to avoid getting "stuck" on issues not resolvable within the scope of the workshop, but to press on towards developing a credible resilience plan for the Exascale timeframe based on the knowledge of the experts at hand. This report describes their results.

2 Technical Challenges of HPC Resilience at Extreme Scale

The challenges of resilience in the Exascale timeframe are well documented by the HPC community through a variety of Exascale studies and plans such as [6, 11]. For the purpose of this workshop we summarize the key points as follows:

The scale challenge: At the system level, horizontal scaling of technology will require three orders of magnitude increase in the number or size of components to reach Exascale. The resulting aggregated failure rate at the system level will be two to three orders of magnitude higher than today's systems, assuming the component level reliability is kept at today's FIT rates. Some experts even doubt that maintaining today's FIT rates will be possible without an increase in power consumption, challenging the constraints on energy usage further for an Exascale system. It is therefore reasonable to expect that an Exascale system will have much shorter MTBF than existing Petascale systems.

The technology effect: It is expected that Exascale system will be made with a silicon node technology featured between 5 to 7nm. At this size, and this late stage of the CMOS technology, several technology related issues will have more effect on resilience than in the current generation, including thermal concentration, electrical noise, near threshold voltage operation, manufacturing quality and circuit complexity. An in-depth study of these issues is outside the scope of this document, so we consider only the net effect on system resilience. It should be expected that multi-bit upsets would be on the rise. Consequently, more complex circuits, more sophisticated error checking will be required, consuming power and/or reducing performance. Furthermore, some experts believe the unprecedented system size will result in some of these errors escape detection, even with the best efforts. Some postulated that a single silent error might affect the system per month of operation.

The energy challenge: Exascale systems must fit within a feasible power budget for operation and cooling. However, resilience provisioning at the fundamental level is a form of providing redundancy either in space or time. Redundancy costs energy and performance, and therefore Exascale systems will face an unprecedented tradeoff among power, performance and resilience. For example, near-threshold voltage (NTV) operation is likely needed to fit within the power budget. But NTV reduces resistance to soft errors, requiring compensating techniques that will cost energy and performance resources, possibly negating some of the energy savings. It is also possible that NTV may introduce susceptibility to temporary timing hazards which will be easy to detect but may require unwinding several instructions in various execution stages, again wasting energy due to detection and recovery actions.

The complexity challenge: RES in the Exascale timeframe will rely on the proper operation of a highly complex, concurrent, and interdependent set of system services. Such software is bug-prone and today is responsible for many of the interrupts experienced by the system. The need to handle scale, more frequent hardware failures and energy will result in more complex and more dynamic system services. Consequently, failure rates will increase.

Over the course of the workshop, participants met together in working groups to examine the current state of HPC relative to these challenges (Section 4.1). They identified gaps (Section 4.2) and dependencies (Section 4.3) associated with the challenges, and formulated a strategy (Section 4.4) and a roadmap (Section 4.5) to implement solutions.

2.1 System Challenges

Every strategy for RES will depend on the correct functioning of various system services: Hardware failures must be detected and correctly reported; checkpoints must be correctly stored and retrieved; processes must be correctly migrated; etc. However, system data and system code are as vulnerable to device upsets as application data and application code. While most of the memory contains application data, a significant fraction contains critical system data, such as page tables or routing tables; this data is frequently accessed and cached, and can be corrupted. The effect of such corruption – e.g., a corrupted page table or routing table entry can be far-reaching and very hard to diagnose.

2.1.1 Description of Hardware and Software

An Exascale system will be constructed of commodity components assembled with the goal of delivering 1000x greater double precision floating point performance than current Petascale DOE systems at a power footprint of less that 20 MWatts. Delivered memory and storage capacity, inter- and intra-node bandwidths and other system characteristics will ultimately depend on the availability of funding for Exascale R&D and the effectiveness of technology insertion into the commodity market. However, it is reasonably certain that a delivered Exascale system will achieve performance through scale out, and energy efficiency through application of deeply scaled components and carefully managed memory hierarchies.

For resilience, this implies increased number of components with no anticipated increase in per component reliability. In addition, it poses challenges of increased complexity, creating more data pathways that need to be protected against failure. Finally, the system susceptibility to soft errors is expected to rise dramatically as a consequence of the requirement to increase pJoules per operation by as much as 100x in Exascale timeframe [24].

System software is complex: it is distributed, nondeterministic, event driven and has timing constraints, due to various time-outs. Its behavior will change as system configuration and size changes.Therefore, it is practically impossible to test such software thoroughly; testing at scale is expensive; in practice such testing occurs only after the first capability system using the software is deployed. Most of the failures on current platforms are due to failures in various system services – in particular, parallel file systems – which is, ironically, the subsystem that supports checkpoint and restart. In some cases, parallel file systems have suffered from data corruption.

2.1.2 Why Not Just Build More Reliable Systems?

While it is desirable to have failure-free system hardware and software, this goal may not be achievable at reasonable cost as both hardened components and methodologies to design and test critical software tend to be extremely expensive. The challenge is to construct a system out of less than perfectly reliable hardware and software that nevertheless behaves as a reliable system from the perspective of the user.

Such a system is likely to be built hierarchically: e.g., the correct execution of application code may depend on various system services that schedule resources, allocate memory and provide recovery services; the correct function of these services may, in turn, depend on the correct functioning of a hardware and software monitoring infrastructure that detects malfunctions, prevent the corruption of critical state, and initiate recovery processes. A complete solution to RES will require new system services, to preemptively migrate components of a large computation, to negotiate with the application modalities for error handling, to compartmentalize in time and space components of an application, etc.

2.1.3 A Resilient Software Architecture

How might one structure system services so that (a) errors are detected; and (b) the probability that an error corrupts persistent state is negligible? One possible approach for achieving this goal is to use a *simplex architecture* approach [38]: Focus on a design where a small, simple kernel can be tested and implemented in a way that its fail-safe functioning is all but guaranteed; and where this small, trusted kernel ensures the detection of errors in system services and prevents the corruption of persistent state. Most of the software can be executed efficiently without being designed and tested to the exacting standards of a fail-safe kernel. Yet, that small kernel architected to more exacting standards and executed in a way that prevents hardware-caused errors is sufficient to ensure proper recovery from errors in the main execution stream.

An application may require different levels of reliability for different parts of their execution and different data sets. The system needs to accommodate these different reliability needs, e.g., by mapping threads and data sets to proper hardware resources, and providing an API to negotiate these needs with the application. The application may also require varying degrees of fault containment. The resource managers need to map "software containment domains" into suitable "hardware containment domains" to ensure containers that are assumed to have uncorrelated failure modes do not map to the same hardware resources.

Information about failures needs to be propagated from the "sensor" that detects the failure to the subsystem that handles the failure: e.g., form the hardware monitoring subsystem to the application run-time. Reliable "backplanes" are required to ensure that the information needed for error recovery is properly delivered in a timely manner. All of these services will need to support a programming model in which "exceptions" are not handled as global failures, but are instead handled by suitable, localized exception handlers. Such an approach allows the application to intelligently distinguish between those system failures which can be "tolerated" and those which cannot.

2.2 Application Challenges

Large-scale parallel codes can be divided into two classes: those which can easily be broken down into smaller, independent tasks, and those which cannot. While the computational needs for "embarrassingly parallel" problems can often be addressed by scaling out commodity parts with very little modification to the source, the latter class demands more sophisticated architectures, careful algorithm design, and an integrated approach to system design. Likewise, from a resilience standpoint, some applications can easily decompose into pieces which naturally lend themselves to a straightforward reliability strategy. These "embarrassingly resilient" applications will likely be well-served by riding the rising trends of utility (cloud) computing and large-scale Internet service infrastructure. For the remaining application domains, novel solutions for fault detection, notification, and recovery must be developed in order to overcome qualitatively new obstacles in the resilience landscape [7, 39, 46].

2.2.1 Current Approaches

Currently, detected faults are propagated through the system stack to applications through return codes (in the best case) or OS signals. These mechanisms are crude, often over-stating the severity of a fault at too low of a level. As the frequency and diversity of faults increases, the overhead involved in treating a recoverable error as unrecoverable will become cost-prohibitive. While catastrophic (loss of control) faults can and should be cause for termination an individual program instance, applications and system software must be given sufficient opportunity and information to take appropriate action in response to recoverable errors.

2.2.2 The Importance of Understanding What Is and Is Not Reliable

Application developers (unconsciously) assume certain fundamental behaviors of a computer, even in "fault-tolerant" codes. While a bit-flip in a data cache might be tested and caught with a software check, the same error in an instruction cache is infeasible to solve at the application level. As the underlying assumptions about reliability begin to move, clear boundaries must established on what can go fail and how badly, taking into account the fact that the costs of such decisions can precipitate increased costs across the entire system design. Moreover, in order for software developers to implement resilience techniques, they must be guaranteed a higher level of reliability in recovery mechanisms. Checkpointing to storage with an identical MTBF as system memory is nonsensical.

2.2.3 The Need for Portability

In order for any viable resilience solution to be adopted by the applications, it must be portable. From a productivity perspective, application developers cannot afford to develop and maintain different versions of a validated and verified code for multiple different system architectures. Most of the codes that will run at extreme scale were developed and debugged on smaller clusters or workstations. Thus, any "hooks" provide by the programming model into the resilience features of a particular platform should conform to a common API. In addition, the libraries upon which an application depends must support a common interface for communicating with both the system and the application about the types of errors it can and cannot handle. These interfaces must be able to accommodate different failure characteristics on different systems.

2.2.4 Productivity, Performance and Reliability Tradeoffs

From the perspective of the user, time-to-solution is paramount. A system supported resilience scheme that is transparent to the user may cost 3x in terms of performance, but it may be preferable to code rewrite that cost 10x in terms of time-to-solution. The cost of various resilience strategies need to be evaluated not only in term of their effect on application runtime, but in terms of application development, debugging, validation and maintenance costs. Without tools and programming model support, some resilience techniques that are attractive from the system perspective may be intractable from the application perspective. In general, a "light-weight" approach, i.e., effective and easy to implement, is preferable to a "full-featured" alternative.

2.3 Resilience Challenges

The optimistic assumption of some workshops looking at HPC in the Exascale timeframe has been that nothing need be done about error rates because industry will presumably have to solve the reliability problem out of necessity. This line of reasoning often fails to take into account that the problems articulated here are a result of the scale of the system and not just the technology. Commercial components will indeed be good enough for indented hand-held and desktop applications. Also, server software has now matured to adopt execution models that compensate for potential problems at the hardware level. There is little incentive for the industry to over-engineer systems to provide manageable interrupt rates and end-to-end data integrity for HPC at extreme scale.

The HPC research community will be responsible to develop the techniques to provide resilience beyond what is available from the commodity market.

Current approaches to fault-tolerance are very limited. Applications use global checkpointing and assume that fault detection is handled transparently to the application; various subsystems, such as hardware monitoring, parallel file system, or resource manager each have their own ad-hoc mechanisms for detecting and handling failures. Detected faults are propagated through the system stack to applications through return codes (in the best case) or OS signals. These mechanisms are crude, often over-stating the severity of a fault at too low of a level. As the frequency and diversity of faults increases, the overhead involved in treating a recoverable error as unrecoverable will become cost-prohibitive. While catastrophic (loss of control) faults can and should be cause for the termination of an individual program instance, applications and system software must be given sufficient opportunity and information to take appropriate action in response to recoverable errors. The workshop attendees believe that new approaches will be needed in the Exascale domain.

2.3.1 The Need for New Approaches

The scale challenge spells the demise of global checkpointing where a single failure requires the entire system to roll back to initiate recovery [9]. This is not a tenable proposition for systems that are expected to have millions if not billions of threads with the associated increase in failure rate as discussed before. We stress here that this is not a "checkpoint performance" problem that can be solved by cleverer checkpointing implementations or faster storage solutions. This is a problem of the expected high frequency of failures and therefore it has to be solved fundamentally at the execution model level. That is, this is about failure containment and isolation.

The current popular execution model based on message passing, as manifested by MPI, lacks in failure containment and isolation. It needs to be enhanced or replaced. In particular, it may become necessary to involve the application in fault detection. Stopgap measures such as independent checkpointing possibly coupled with some form of message logging, can and should be deployed and tested in the interim. Such methods when combined with more sophisticated checkpointing at the node level and faster stable storage (e.g., SSD) can extend the usefulness of checkpointing further but we should not be sanguine about the effectiveness of these methods in the long run they only delay the inevitable switch to a more inherently resilient execution model. We present some promising directions in this regard in Section 4.4.

2.3.2 The Impact of Soft Errors

The likely increase in SER due to technology issues *and* system scale will require some research to assess the potential damage and solutions. Previous research shows that about 40 to 50% of errors that escape hardware detection end up causing no damage, either because they hit an unused functional unit or a dead variable. For the remaining failures, research is needed to find the best tradeoff in hardening the system to maintain or improve upon the current SER rate in future systems. There are also potential system level and application level techniques that can be deployed to help in the problem, but the current state of the art requires some fundamental advances to enable such techniques to be practical [32].

It is also important to stress here that some optimistic assumptions have been stated at several Exascale meetings that nothing need be done because the industry will have to solve the problem out of necessity. This line of thought ignores that many of the problem articulated here are due to the scale of the system and not just the technology. Components designed for "COTS" will indeed be good enough for indented hand-held and desktop applications. Also, server applications have now matured at the software level to adopt execution models that compensate for potential problems at the hardware level. There will be no incentive for the industry to over engineer systems just to meet Exascale requirements that will be needed in one or two large systems. The Exascale community must research additional techniques that will provide added resilience beyond whatever improvement the industry will introduce.

2.3.3 New Opportunities in Hardware

New hardware advances are also likely to impact resilience and provide opportunities. It is expected that some form of SSD-based memory will be available at competitive cost to provide a durable byte-addressable memory. It is conceivable that a good portion of the system memory will be implemented with this technology due to its energy and size advantages. However, beyond serving as a fast storage device for paging or file system metadata we currently do not have technology to exploit this technology at the operating system, system architecture, or application levels. There will be a need to exploit this technology beyond just providing a faster stable storage device. Research is needed to exploit these technologies beyond the obvious usage. For instance, if every memory write is durable, what would be the impact of this on system resilience overall?

2.3.4 Global View of Resilience

Resilience is a cross-cutting problem that affects every component of the system. A bit-flip in a cache can corrupt application data; it can, to the same extent corrupt user code, corrupt system data, such as page tables or routing tables, and it can corrupt system code. The effect of such corruption – e.g., a corrupted page table or routing table entry can be far-reaching and very hard to diagnose.

Software developers (unconsciously) assume certain fundamental behaviors of a computer, even in "fault-tolerant" codes. While a bit-flip in a data cache might be tested and caught with a software check in the application code, the same error in an instruction cache cannot be handled at the application level and application programmers assume such errors do not occur. Users periodically checkpoint the application states and often assume checkpoints are reliable, but the storage may be no more reliable than the data being checkpointed.

As underlying assumptions about reliability begin to change, it is important to categorize error types and understand the frequency and impact of each type of error; it is important to understand how the different system components collaborate and depend on each other for fault detection and correction. Clear, explicit boundaries and divisions of responsibilities across layers will be essential to design fault-tolerant software. As a result, the design choices may have a significant impact on the cost and performance of the selected approach.

2.3.5 Evaluating the Tradeoffs

Finally, the tradeoff among energy, resilience and performance is not very well understood currently, let alone for three technology generations in the future. There is currently no way to identify the increase in energy consumption due to resilience, as there are no reliable models or system-level measurements to serve this purpose. Such methods need to be developed, and the impact on applications must be understood. It will be necessary to experiment with different execution models to see how they handle the tradeoff of energy-resilience-performance.

3 Perspectives on HPC Resilience from Across the System Stack

3.1 System Hardware

Exascale machine in 2018-20 timeframe will use 7 nm process technology, it will have three orders of magnitude higher parallelism, operate near- threshold voltage (NTV) operation which will increase concurrency burden even further, and thus the machine will be even more fragile. Expect reliability to get worse with deeply scaled technologies exhibiting unknown aging effects, increased hardware, increased variability, increased soft error susceptibility, and decreased noise immunity due to NTV operation [36, 37]. Traditional redundancy based reliability approaches are not practical because they require 2-3x more energy, which is not practical. Therefore, resilience is the best affordable approach. We define resilience as a technique to asymptotically provide reliability of an N-modular redundancy scheme, with the goal of only 10% energy and hardware cost.

3.1.1 Categorization of Faults

Faults can be broadly categorized as permanent, gradual (spatial and temporal), intermittent, and aging as shown in the table. Permanent faults are screened during production, or in the field, and are easy to detect and correct. Gradual faults will be new, manifested by NTV operation, such as variations, effects of temperature and other ambient conditions. In the past we have employed guard-bands, such as lowered frequency of operation to circumvent these faults, but in the Exascale timeframe such guard-bands will be prohibitive.

Faults	**Type**	**Example**
Permanent faults	Stuck at 0 - 1	Open, shorts, power supply or fan shutdown
Gradual faults	Spatial: Variations Temporal: Temperature effects	Fast and slow cores Change in frequency with temperature
Intermittent faults	Soft errors Voltage droops	Data corruption, loss of control, not reproducible
Aging faults	Degradation (slow gradual temporal)	Loss of frequency over time, erratic bits in memory

Table 1: Broad categorization of faults most relevant to an Exascale class machine in the 2018-2020 timeframe

Intermittent faults caused by soft-errors and noise are the most to fear about because they could corrupt data, exhibit loss of control, may go undetected and are not reproducible. Soft-error rate of state-elements, such as memory and latches, is trending down with technology scaling; however, the number of state elements increases faster, resulting in overall increase in soft-error rate of the system [32, 36].

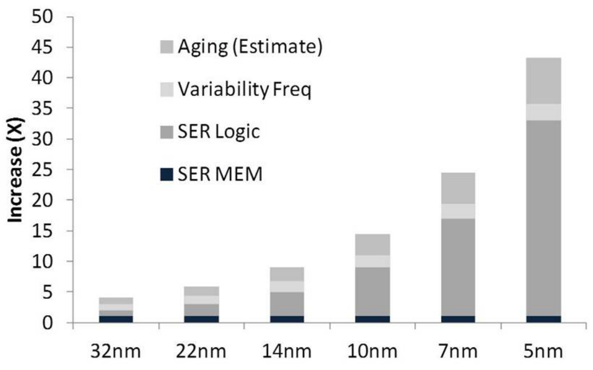


Figure 1: Estimates of the relative increase in error rates as a function of process technology

Permanent faults in the field typically do not increase with the level of integration. That is, loosely speaking, the rate is constant per integrated component. These faults are more common with non-integrated hardware components such as power supplies and fans. Also, they are relatively easy to detect and correct using today's well understood methods. Coping with gradual and aging faults, on the other hand, will be increasingly difficult and needs rethinking of the whole system design. And the impact of intermittent faults for sustained and reliable operation of the Exascale machine needs the most investigation.

3.1.2 Expected Impact of Faults

Figure 1 illustrates the level of difficulty we will face with each type of fault, and how it stacks up compared to 45 nm generation. Soft-error related faults in the logic latches are the most fearful. These single event upsets in memory are well understood and are corrected by ECC; however, multi-bit upsets will be on the rise. Variations and unknown aging effects will also be on the rise. Adding all of these effects suggest that resilience of the Exascale system in 2018-20 timeframe could be factor of 20 to 40 more challenging.

3.2 System Software

As indicated in Section 1, Resilience at Extreme Scale (RES) is a cross-cutting problem that will need cross-cutting solutions that span software and hardware components of the system. The responsibility for error detection and error correction in resilience solutions must be shared jointly between hardware and software. Current approaches focus on techniques that are most conveniently implemented at a specific level of the system. For example, hardware approaches to error detection include techniques such as parity checks and watchdog timers, while software approaches to error detection include techniques such as anomaly detection and redundant executions. Likewise, hardware approaches to error correction include ECC in DRAM and software approaches to error correction include a variety of rollback/replay techniques ranging form checkpointing to transactional paradigms. Instead, a truly integrated solution to resilience will require close coordination of hardware and software that focuses on increasing the effectiveness of error detection and correction across the board.

Keeping this co-design goal in mind, we now focus on a software perspective of resilience. Even within software, there are multiple levels at which resilience support can be introduced, and an integrated solution will need to harness the capabilities of all the software levels. Thus, some of the most important challenges include developing an integrated solution for resilience support across all levels of software stack, as well as quantifying resilience requirements (error significance) for applications, and resource optimization with respect to those constraints. Table 2 lists examples of resilience support at different levels of software.

Most importantly, these techniques have to be integrated in a meaningful manner: It does not help to have a fault-tolerant algorithm, if the services it relies on are error-prone. The overall resilience of a system is determined by the resilience of the weakest link in the system.

3.2.1 Idempotence and Data-Flow Tasks

Let us examine some of the examples listed in Table 2. A powerful concept in parallel programming models is that of *idempotent* tasks. A task T is idempotent if multiple executions of T result in the same answer as a single execution. Map tasks in a Map-Reduce model are a simple example of idempotent tasks. For more general parallel programs, a sufficient condition for establishing idempotence is that the task's output shared variables must be disjoint from its input shared variables. In many cases, this condition can be checked by a combination of compile-time and runtime techniques. If a task is declared to be idempotent, an error message can be issued statically or dynamically if the idempotence property is violated.

Another programming model concept that can aid in resilience is that of *data-flow* and *data-driven* execution. In a pure data-flow model, each task's outputs are pure functions of the task's

Software level	**Examples of resilience support (includes ideas still in research stage)**
Application frameworks	Resilient algorithms [7,22,29]
Libraries	Resilient data structures (replicated arrays [34], tuple spaces [3]), checkpointing libraries [33]
Programming models	Idempotent tasks [31], transactional semantics [25], data-flow [5,13], actors, fuzzy barriers [18], phasers [40]
Compilers	Checkpointing optimizations, rollback support, detection of soft errors
Runtime systems	Process-level resilience [43], task-level resilience [30], task/data replication and migration, portable resilience with heterogeneous processors, checkpoint-on-demand
Communication and data consistency systems	Message logging, containment domains [42], hierarchical places, transactions/sandboxes
Middleware, virtualization, operating systems	Publish-subscribe frameworks [19], resilience domains, durable storage, process isolation and migration

Table 2: Examples of resilience support at different levels of software

inputs. Thus, as objects become part of the *execution frontier* in a data-flow program, they can be checkpointed asynchronously in parallel on one processor without requiring any coordination with checkpoints being obtained asynchronously on other processors. Examples of current data-flow models can be found in the Intel Concurrent Collections (CnC) language, and in a large variety of libraries and frameworks that support "DAG parallelism".

While resilience support can be greatly simplified if programming constructs are constrained for analyzability (e.g., idempotent tasks, data-flow tasks), the ultimate success of programming model extensions lies in the adoption of such extensions by the HPC community. Fortunately, we have a unique opportunity to motivate the use of resilience-friendly programming models right now by observing that *what's good for simplifying parallelism is good for simplifying resilience.* Bear in mind that none of this comes without a tradeoff, as improvements in productivity and resilience must be balanced with the impact on performance, resource utilization, and energy efficiency in overall system design.

3.2.2 The Runtime and Compiler

We now briefly discuss a few examples of resilience support at the runtime and compiler levels. A recent breakthrough in the use of heterogeneous processors is to make them more amenable to dynamic task scheduling. In such as a scenario, it becomes possible to pre-compile multiple versions of a task and let a a runtime system dynamically decide which kind of physical processor (e.g., CPU, GPU, FPGA) a given task should run on. Such an approach should be beneficial for resilience as well, because it could easily be extended to support replay of tasks across multiple (heterogeneous) cores.

Likewise, runtime scheduling on hierarchical structures such as Containment Domains and the Hierarchical Place Tree (HPT) can also potentially lead to new opportunities for resilience. The compiler level also offers a number of techniques to assist with resilience, and to reduce the overall overhead of resilience via code transformations. For example, if memory is assumed to be reliable, then a compiler can modify code so that all operations are duplicated; the number of memory accesses is not increased (except as caused by increased register pressure), so that the total overhead can be significantly lower than a factor of two.

3.2.3 Areas of Greatest Concern

Finally, the areas that need the greatest attention are as follows:

1. Identification of execution model primitives to integrate software solutions for concurrency, energy efficiency, and resilience
2. Tight integration of inter-node and intra-node resilience solutions
3. Identification of realistic testbeds for evaluating software solutions for resilience
4. Collaborations among application developers and system software developers to develop usable approaches for software resilience

3.3 Visualization and Data Analytics

The ultimate purpose for Visualization and Data Analysis (VDA) is investigating, understanding and discovering science within the simulation codes. Visualization and data analysis is an integral part of scientific simulations and experiments. It is also a distinct separate service for scientific discovery, presentation and documentation purposes.

3.3.1 Current Practices

Current visualization and data analysis is largely done as a post-processing step, in which interactive visualization tools read in data saved to disk, and an analyst sitting at a desk interacts with that data in real time. This method uses the disk as a communication mechanism between the application and the VDA application, so the scientific code and interactive analysis are functionally decoupled. There are individual efforts aimed at changing this workflow, but it remains the main way that people interact with their data.

3.3.2 Significant Changes at the Exascale

At the Exascale, this workflow will be completely broken due to the mismatch between the rate at which we can create large data, and the rate at which that data can be moved to persistent storage. In fact, this data movement will be so costly in terms of energy that it will be cost prohibitive to move results from memory to persistent storage. Because of this, Exascale computing will have integrated VDA as a method of determining what data are of interest and therefore worth committing to persistent storage.

As a consequence, post-processing will fundamentally change due to the complex nature of the artifacts we are putting to persistent storage. Thus, the following "use cases" will be necessary at Exascale:

In-Situ VDA: A simulation is compiled with a VDA capability library, integrating the capability directly into the code. After a time step is computed, the application executes VDA operations on data resident in memory. Results may be saved to persistent storage.

In-situ VDA Service: A simulation code sends data and VDA requests to a VDA service, running independently from the simulation. VDA operations are carried out by the service, and occur independent of the code.

In-situ VDA Hybrid Service: A simulation code is compiled with a VDA library which performs some operations in memory, and ships other operations out to be handled by a VDA service. Examples include extraction of geometry, then shipping that geometry to a service to be rendered.

Post-processing batch mode VDA: A VDA executable is run in batch mode on data that have been saved to persistent storage. The results are then written to persistent storage. An example of this is a post-processing analysis performed over an ensemble of 1000 runs of a single simulation.

Interactive post-processing VDA: VDA application is run as post-processing step, operating on data that has been saved to some persistent storage. Examples include interactive visualization and exploration of data.

In each of these use cases, VDA has the same resilience requirements as an application, and will handle resilience according to the system design. In particular, if resilience is "handled" by other layers of the software stack, VDA capabilities will take advantage of those. If resilience is left to the applications, then VDA capabilities will handle resilience internally. We expect no VDA-specific requirements other than those already needed for applications.

3.3.3 Assumptions and Requirements

In each use case, we assume that data integrity and correctness are clearly defined by the system, and the context for that data - its provenance - is consistently propagated through the entire software stack and simulation workflow. We expect that the default behavior of the system is that data are correct, but context data must be propagated so that this is guaranteed across the system.

One exception to this is interactive post-processing. This is a very different use case for resilience, as *it requires that failures be resolved as the data are interacted with.* This may be a novel use case, if applications do not require on-the-fly recovery in the face of failures. In particular, if the system solution for failures is simply "detect and restart", interactive analysis and visualization will need to develop an independent solution. Since it is not clear at present if on-the-fly recovery is a requirement for applications, this specific use case should be considered a VDA-driven resilience requirement.

An additional requirement for VDA is that the visualization itself - the only truly independent product of VDA, in that it is not analogous to any other process - must not introduce visual errors as a result of resilience issues. Again, we expect that the system wide solution for applications running at Exascale will be applied by the VDA algorithms, with the additional constraint that no visual artifacts be introduced.

3.3.4 Need for a Coordinated System Approach

As a capability library, a service, and an application, Visualization and Data Analysis will be subject to the same resilience constraints as codes and code libraries running at Exascale. It is critical that VDA take advantage of the system-wide definitions and APIs that enable science applications to run usefully at Exascale.

Making sense of data at Exascale requires a system-wide approach to data integrity and correctness, which all applications must participate in. Again, VDA will participate in this as a peer, and must obey the same constraints, interfaces and optimizations that codes face.

VDA algorithms, like all other algorithms, must not introduce unquantifiable changes in the data, and the visualizations must be quantifiably correct. Otherwise, investigation and understanding of the data will not be possible.

3.4 Data Storage and I/O

HPC application teams rely on data storage and access to accomplish two goals. First, the storage system is often used as a holding area for "checkpoints" of application state that can be used to restart computation in the event of an application interruption. Second, the storage system is used to hold data that will be further analyzed or shared at a later date.

3.4.1 Current Trends

Traditionally, HPC storage has taken the form of a parallel file system managing disk-based persistent storage that is positioned external to the compute fabric, and today most deployments are still of this nature. However, three trends indicate that change is coming.

1. Current parallel file systems are seen as both unreliable and a performance bottleneck for many applications [4,28]. Parallel file systems are a major cause of application interrupts on current-generation systems, and the expected growth in node counts is likely to exacerbate this problem. While parallel file systems can deliver a great majority of the underlying hardware bandwidth for carefully-written synthetic benchmarks, rarely do applications approach hardware speeds.

2. Disk drive bandwidth improves at a very slow rate, forcing disk-based storage systems to employ increasingly large numbers of devices to achieve required I/O rates. Deployments using millions of drives are seen as untenable; thus, it is expected that disk drives will continue to be used to meet capacity requirements while other technologies will be needed to meet bandwidth requirements [17].

3. In-system storage is being successfully utilized as an augmentation and alternative to traditional, external storage deployments. As an augmentation, in-system storage is being used as a "burst buffer" to hold data in transit to external storage, freeing applications to return to computation while data is asynchronously written [27]. As an alternative, applications and middleware are using in-system storage (e.g., DRAM, SSD) as a container for checkpoint data [33], enabling protection from certain classes of failures without the need to interact with external storage.

To best address the data storage needs of future HPC systems and applications, additional effort is needed to understand and adapt to these three trends.

3.4.2 Software Concerns

A primary concern with respect to HPC storage is the reliability of storage software. Existing storage software (e.g., parallel file systems) are a significant source of failures, at least in part because the health of these services is tied to the health of clients. This tie must be broken. Additionally, the Internet services community has demonstrated the effectiveness of highly-distributed storage systems built on commodity hardware [12,16,21].

A better understanding is needed as to the degree that HPC storage software can meet HPC application requirements while using this more economical hardware. Also, in-system storage brings a new resource that must be managed, and it is unclear whether this should be managed as a separate tier of the "global" storage system, or whether this resource should be managed by separate software as an application resource. Finally, end-to-end data integrity is not ensured by current systems, and methods must be developed for allowing applications and storage software to work together to meet this requirement.

3.4.3 Hardware Concerns

In the next 10 years, it appears that a mix of spinning disk and solid-state storage will be the most cost-effective means to meet HPC performance and capacity requirements. Even though this approach limits the number of disk drives that will need to be deployed, problems such as latent sector errors in spinning media must be addressed in a manner that preserves performance for applications. Furthermore, while it does not appear that the write endurance of solid-state storage used as a "burst buffer" would be an issue, if this in-system storage is used for other purposes, it is less clear that current technologies are appropriate. A better understanding of the I/O requirements of applications using in-system storage is needed and must be mapped to the capabilities of current and future solid-state storage technologies.

3.4.4 The Role of Abstractions

The abstraction presented by storage has implications on resilience as well. Providing the POSIX abstraction used in HPC storage requires coordination that is in part responsible for the fragility of current HPC storage software. The Internet services community has found alternatives that allow them to accomplish computing tasks while reducing the impact of faults. New abstractions and associated semantics are needed that facilitate resilient implementations (and simultaneously provide better usability). Additionally, given the need to optimize resource use to achieve high performance in HPC systems, applications should be presented a more rich model of storage that allows teams to trade-off between performance, resilience, and consistency of data.

3.5 Tools

Resilience techniques will have a profound impact on the entire software stack. They will increase the system's complexity and users will expect tools that either hide this extra complexity or explicitly expose its effects.

3.5.1 Enhancing Existing Tools

Foremost, users will expect that their familiar performance analysis, debugging, or code correctness tools still continue to work despite lower reliability and independent of any fault tolerance or resilience implemented within the runtime system, the operating system, or the underlying architecture. This will require significant enhancements on current tools. They need to be able to coordinate with fault-tolerance mechanisms and they need to be able to hide effects caused by faults.

For example, debuggers need to survive checkpoint/restart cycles with their complete state intact so that users can debug problems that only occur after significant runtimes; performance analysis tools need to eliminate performance information that is affected by fault mitigation techniques; tools need to be able to ignore source code changes introduces by source-to-source translation approaches targeted at code duplication; or communication correctness tools need to be able to perform their work despite message or link failures in the system. The changes required for this functionality should be transparent to the user and hidden inside the tool implementation.

3.5.2 Tools for Fault Monitoring, Management, and Logging

At the same time, tools need to be available that expose failures as they happen, allow users to track them, and help understand the impact they have on the application and system software stack. This should include the ability to trace failures along with corresponding mitigating actions as well as to map the information back to source code, e.g., to allow users to identify code regions that are the cause for higher fault rates.

Combined with the requirement of transparency discussed above, this will have to lead to tools that allow users to interactively change the level of transparency and thereby enable them to adjust to the particular problem they need to investigate. As one of the main challenges, such tools need to be kept intuitive and provide a clean transition between the different levels of transparency, allowing the user to cleanly map data between the different view points. In addition to these application level tools, system administrators will expect an additional set of tools that allows them to monitor and log overall system state as well as system health.

3.5.3 The Need for Interface to Introspect Resilience Techniques

One of the most important aspects in fulfilling the requirements laid out above is the need for tools to interact with the remaining software stack, in particular the components that implement fault-tolerance techniques. Tools need to be able to capture any fault and any corrective action taken, such that they can react to the same fault appropriately. This will require a set of interfaces, co-designed across the entire software/hardware stack.

3.5.4 Fault Tolerance in Tools

In addition to the new functionalities discussed above, tools themselves also need to be resilient to failure. In checkpoint/restart schemes, they need to coordinate with the checkpointing system, be it in the application or the system, to include tool data in the checkpoint and they need to be able to retrieve it on restart. In scenarios in which the applications is hardened against failures, tools needs to apply the same techniques in order to maintain the same level of resilience, in particular in scenarios in which they directly add instrumentation to the application.

It is important to note that these challenges are not unique to the tools ecosystem. Other layers of the software stack, including I/O, visualization and in-situ analysis, will face similar obstacles in fault-tolerant software design. Therefore, it is highly desirable to integrate these services in a common framework and to provide a coordinated approach of resilience.

3.5.5 System Support for Tools

The various tools will need to gather and correlate information streams coming from many subsystems: e.g., hardware monitoring infrastructure, system management infrastructure, file-system, run-time, application, etc. Some of this information is needed in real-time, in order to initiate proper fault-avoidance or fault-recovery operations, some is needed for proper resource management at the level of the OS and runtime, and some needs to be stored off-line for long-term analysis e.g., to study failure characteristics of individual components, and easily accessible for post mortem analysis.

For information collection and control, tools rely on external resources, such as tree-based overlay networks running on a separate set of tool nodes. One example for this kind of tool is the Stack Trace Analysis Tool (STAT) [1, 26], which is based itself on MRNet [35]. These types of dependencies will create additional vulnerabilities outside the application. Tools will need to be resilient to problems on these extra resources, i.e., neither applications nor users should be impacted by failures that happen outside the actual application. Furthermore, communication system such as MRNet assumes a simple flow of information from the components of an application to a user console, and a simple reverse flow of control. This approach may not be scalable. In addition, many subsystems may collect the same information redundantly as a consequence of such an integration.

To avoid these issues, it will be important for the system to provide a robust and scalable "publish and subscribe" infrastructure that gathers information from the various subsystems, distribute it to the various control agents and stores it, as required. This infrastructure must be fault-tolerant and not interfere with the compute performance of the system.

3.5.6 Testing Through Fault Injection Tools

While the tools and tool capabilities described above cover the necessary aspects of running tools on a system that may exhibit faults, we also require tools that help design and test algorithms, applications, runtime systems, and even tools for such an environment. Even with the higher error rates expected in future architectures, they will still be rare enough to make testing and code hardening difficult. To overcome this challenges, we will require tools that enable developers to inject errors, faults, or other anomalies in a controlled and repeatable fashion.

Such injection tools can work at multiple levels, from adding code to binaries to modifying source code, from adding individual instructions to altering communication traffic. A wide variety of techniques for this are known [20, 23, 41], but are mostly restricted to certain error classes and used mainly for specific research studies. We need to extend and generalize such tools and make them available to developers. Additionally, and more importantly, we need to change development habits and make testing for fault-tolerance a fundamental part of any software design cycle.

3.6 Algorithms

Presently it is difficult to implement fault resilient algorithms since current programming models assume reliable computations. As a result, the algorithms-based resilience discussions at this workshop focused on extensions to existing programming models that would allow algorithm designers and programmers to reason about, design and implement resilient algorithms.

3.6.1 Background

Fault resilient algorithms have been the focus of a number of papers. These effort are primarily in the area of algorithm-based fault-tolerance (ABTF) where meta-data about the problem, or knowledge about basic algorithm properties is used to monitor computations, detect faults and, if possible, recover state [10, 22, 29]. Although academically interesting, these algorithms have not played a significant role in high-end computing so far. This is because, although resilience has been a concern for some years, systems have, to-date, been more reliable than predicted. Also, even for application developers who are concerned about resilience, commonly available programming and runtime systems do not readily support the required recovery mechanisms for these algorithms. Furthermore, recovery of problem state from these algorithms alone–state that is usually derived from some primary state–is not useful unless the primary state itself is recovered. For example, if a linear solver can recover from a fault but the nonlinear state that was used to generate the linear system is lost then the linear solver state is irrelevant. The principle ideas discussed are presented in the sections that follow.

3.6.2 Persistent Local Data

Most scalable science and engineering applications use an SPMD programming model enabled by MPI. They also have some notion of locality such that data is decomposed so each MPI process is associated with a subset of the global data, and in order to make progress to the solution each processor needs to communicate regularly with only a subset of its neighboring processors.

Presently these applications tend to write problem state to a global checkpoint/restart file. This file requires coordinated access across all processors. This approach represents a significant bottleneck to scalability on future systems because the time it takes to create and restart from this global file rivals the required uptime of future systems.

In this workshop we discussed an approach that supports localized checkpoint/restart. The fundamental idea is that the programming environment and runtime system would allow application developers to store specific problem data *persistently* so that, if an MPI process failed, a new process would be started and the persistently stored data would be visible to the application in the new process through a specific function the application had previously registered with the runtime system.

By having an API to store data persistently, and a recovery mechanism that allows a recovered process of the application to access its own persistent data (and perhaps its neighbors' data), application developers can design and implement checkpoint and recovery algorithms that permit localized checkpointing (for example, the persistent data of one processor can be replicated on a neighboring processor, or on local NVRAM).

One simple example that could use this approach is an explicit algorithm for transient PDE simulations using finite elements, differences or volumes. For this application, each MPI process is assigned a patch of the domain and has additional storage for "ghost" or "halo" data. These halo data are copies of values from time step n on neighboring processors acquired during a halo exchange step, completed just prior to computing values for time step $n+1$. Prior to computing time step $n+1$, state from time step n is stored persistently. If an MPI process is lost during a particular time step it is possible to re-compute it time step $n+1$ values by retrieving time step n data from persistent storage.

Of course we would not store data from each time step persistently, but do so often enough so that the cost was worth the benefit. One final note worth mentioning is that, by providing access to persistent storage of neighboring processors, a recovering process can reconstruct multiple time steps simply by accessing a larger halo of values. For example, if time step $n-1$ data were persistently stored, the restarted process independently compute it time step $n+1$ by getting a halo of twice the size from its neighbors' persistently stored time step $n-1$.

Persistent local data storage and recovery, if provided through a simple API, enables a broad set of checkpoint and restart algorithms that should result in much more resilient application execution. Furthermore, the data and programming needed to support this model are not much different from what is already done to support global checkpoint and restart.

3.6.3 Algorithm-based Self-consistency Checking

When computing on a reliable hardware platform we take for granted that implicitly define algorithmic properties are preserved. When reliability becomes a concerns, these properties can be explicitly tested to detect, and maybe even correct, errors. Many problems have easy-to-check consistency tests. For example, the computed answer $\tilde{x}$ to the linear system $Ax = b$ provided by an iterative linear solver can be tested by explicitly computing the norm of $r = b - Ax$.

More algorithm-specific tests could include confirming orthogonality properties of basis vectors, or checking that known conservation properties are preserved. For example, interior cells of a conservation-based formulation of a PDE have a zero sum property.

More generally, we can explore the use of preconditions, postconditions and invariants to drive the formulation of self-consistency checks. Although such checks are already present in many well-written codes, as a form of sanity test for user-input errors or algorithmic breakdown in the presence of floating point roundoff error, they can serve a very valuable role in the detection and even correction of data and execution faults.

3.6.4 Selective Reliability

As software errors increase, the chance that faulty results will be undetected by the hardware or system software increase, to the point where an application may have to deal with unreliable results. In order to address this concern we need to develop new algorithms that can tolerate soft errors.

Robust fault soft error resilient algorithms need some ability to compute reliably with reliable data. Without these features, there is no way to assure that any computational result is reliable. In this workshop we discussed the concept of *selective reliability*, such that the programmer can declare specific data and computations to be more reliable than the "bulk" reliability of the underlying systems. This higher reliability can be implemented in a variety of ways that will likely vary depending on the system. Given the ability to have highly reliable data and computation, new algorithms can be developed that keep much of the data and computation in bulk reliability mode, but have some in highly reliable mode.

One approach that can benefit from this approach is a sandbox model, where an algorithm starts in an outer level highly-reliable mode and dispatches a computation in an inner level bulk reliable mode. Upon completion (if it completes!) the sandbox computation is analyzed by the outer level to determine if the result is valid. If it is valid, its results are assimilated. If not, the results may still be useful, or may be discarded.

3.7 Application Developers and Users

Application resilience has been practiced for decades by designing the application to generate restart dumps on persistent storage at user-selected intervals, and restarting from these files in the event of either a system crash or the expiration of the user's time slot on the system. For handling the event of time slot expiration, this mechanism is likely to be practiced into the indefinite future. However, as a mechanism for handling hardware or system software generated interrupts, this strategy, known as checkpoint/restart, is a blunt weapon.

3.7.1 Limitations of Traditional Checkpoint Restart

No resilience strategy comes free of cost. Essentially all strategies for resilience demand some form and degree of redundancy. The restart file is a redundant copy of the state of the computation saved to persistent storage. It is usually considerably smaller than the entire state of the memory used in the computation, but it is still relatively large. It is much larger than the normal data output from the computation, which is likely to consist, at most, of visualization files, requiring only eight bits per variable, and global or sub-domain averages of variables, spectra, or other highly distilled information.

For Exascale systems, and for large runs that utilize the entire system, we anticipate a frequency of non-user-generated interrupts that is too large to make global checkpoint/restart to shared persistent storage a practical resilience strategy. If only a single node or a single switch has failed, it is unreasonable to require that the entire computation, which might involve 10,000 or 100,000 nodes and thousands of switches, stop and return to a global saved state from persistent storage. Not only would such a global reset be wasteful, but it might not be possible to write checkpoint files fast enough for the overall computation to make progress using this strategy.

Note that this analysis does not intend to suggest that the remedy is a simple as replacing global checkpointing with local checkpointing. In fact, local checkpointing time at large scales is not dominated by disk write times but by the complexity of reconstructing a globally consistent state from millions or even billions of communicating processes. In fact, this problem may be harder to solve than the global checkpoint problem. On the other hand, a strategy that involve writing the global checkpoint file in an embarrassingly parallel fashion to local or neighboring storage may be in some sense optimal since it at least scales with the number of compute elements.

3.7.2 Alternatives to Global Checkpoint Restart

In most applications, we can define a domain of dependence for the present computational state at a given node. Going backward in time, as enumerated in the computation through "time steps," or in iterations of some global solver, we can identify those nodes upon which the present state at a node of interest depends. For hyperbolic systems of differential equations, this concept of the domain of dependence comes from finite signal propagation speeds, which limit the size of the domain at any previous time that could possibly affect our present state.

Computationally, the domain of dependence is usually larger than this physical domain, since arithmetical signals can propagate faster than the physical signals that they simulate. Nevertheless, it is usually the case that if for any reason the computational state of a particular node is lost or contaminated, we need only to regenerate from a set of nodes at a saved earlier state that is very much smaller than the entire system participating in the computation. From this perspective, we can easily see the wastefulness of forcing the entire computation to return to an earlier state, although doing that is by far the easiest strategy to implement.

One might object here that global solvers, by their very definition, upon every time step tie the state of a single node to the state of every other node in the system. However, it is a general feature of such solvers that the effect of a node upon a distant node is small, and that therefore this effect can be encapsulated in only a small amount of data that needs to be transmitted. This feature results from the solvers approximating the long-time limit of physically causal systems. (For example, Poisson's equation approximates the action of gravity in the action-at-a-distance limit in which the speed of light is infinite. It also appears in incompressible flow solvers, where the speed of sound is assumed to be infinite. The effects of signals are diluted with distance by the simple fact of geometry.) Because only a small amount of data is needed from a distant node in such a global solver, effective strategies for regenerating this data to recover from a node failure without having to repeat the entire calculation at the distant node can most likely be devised. One possibility would be to save this compressed state using a shorter time interval than is used for the entire restart dump. A second might be to use message logging with asynchronous checkpointing, leveraging the notion of physical causality to simplify the problem of reconstructing a globally consistent state.

3.7.3 Recovering State After an Interrupt

Recovering the state of a node, either after that node is reset or on a new node assigned to replace its role in the computation, only seems to make sense if the application has implemented some strategy for dynamic load balancing. Otherwise, even if a new node is provided, the recovering nodes will lag behind the rest of the system, and will eventually slow down all the other nodes; the performance will be not better than with a global, synchronized restart. It is not surprising that when we allow for a dynamically changing set of computational nodes, as various nodes encounter hardware or system faults and are replaced, that we must also devise a strategy for reapportioning work, so that the recovering nodes can be assisted in getting back to the time or iteration level of the rest of the computation. Load balancing will be useful for many other reasons, such as coping with dynamic changes in the execution pattern, or coping with changes in processor speeds due to power management. Once we assume that such a dynamic load balancing strategy has been implemented, it may no longer be necessary to assign replacement nodes, but instead simply reapportion the work among a smaller set of nodes.

3.7.4 Implications of Resilience

From the above discussion, we see that resilience places two new burdens upon application code developers. First, they need to identify failed nodes, their computational domains of dependence, and code to execute in order to recover using data from the domains of dependence. Second, they need to implement dynamic load balancing strategies. These are significant programming burdens, and the second is far greater in most cases than the first. These new demands are, however, fairly general, and it is possible that general tools, libraries, or frameworks - or possibly new programming language features - could be devised to assist programmers in addressing them.

These efforts demanded from application developers are, however, not enough. There is no point in making these code modifications if the application will be killed by the system upon the first node failure, regardless of the application's ability and willingness to recover unaided. At present the default response of system software to a detected error is to terminate the application, at least in part because alternative containment mechanisms to prevent error propagation do not currently exist. Present job scheduling software will force the application to relinquish its node set in such an event. This node set is likely to hold the live redundant state information necessary for recovery. Thus the checkpoint/restart resilience strategy is hard-wired into the present software stack at multiple levels. There is no point in changing one of these levels unless all are changed in a coordinated fashion.

The application code sits at the top of the software stack, and thus is accustomed to receiving alerts, interrupts, or error messages from below, but it is not generally accustomed to sending information about system health back down to the lower layers. Nevertheless, a code receives system health information unintentionally during the normal course of its operation, and it could be instrumented to transmit that information to other system agents. Much of this health data is a fundamental part of a dynamical load balancing strategy. Therefore the application will have to monitor system health in order to be resilient in any event. Consequently, it might as well relate that health information to the agents who might act upon it.

3.7.5 Coordinated Resilience: An Example

One can see this more clearly from specific examples. In implementing dynamical load balancing, one might want to monitor the amount of work involved in individual tasks that are dynamically assigned to nodes. This could be a tally, made by the code as it runs, of the flops performed as well as a tally of the amount of data read from and written to remote nodes and the amount of data transferred during the computation to and from the node's CPUs. Upon each time that this task is performed, one could make these tallies and record them along with the particular node that did this task and the time it began and finished the task. From this data, which is very helpful in deciding how to assign the task the next time it must be performed, we can easily compute the aggregate performance of this node.

We would expect this performance to be fairly consistent, but we might find that a particular node runs very much (say, a factor of two) slower than other, supposedly identical nodes on this sort of task. One could inform the system that this node appears to be running slowly. A particular cause of this sort of system health signal in the past is node overheating, although other factors could be involved as well. Another node health signal we might see in an application is that a particular node takes ”forever” to write data to persistent storage, while others take only a normal amount of time. We could exclude the node, but we could also inform agents who might be able to address this situation. A very easily detected health signal is that we find that a node has never completed a task it was assigned. We can then assign the task to a different node, and if it completes the task in a normal time interval, we may conclude that the non-responding node is dead.

The system is most likely aware of this node’s demise from other software layers, but we might still inform an appropriate agent. Signals concerning interconnect health can also be apparent to a running application. Such problems can be undetected data transmission errors. An obvious data transmission error, which has been known to happen, is a string of zeroes in the intended message. One might think that checksumming by the application would be needed to detect these errors, but dividing by one of the zeroes in a bad message string will certainly cause a detectable error. To determine that this is a system and not a code error would require that a resilient code implementation recover from a saved state by re-executing the same code and discover that the error does not recur. Such a process should be automated in coordination with both the system and application.

3.7.6 Role of the Application in a Total Resilience Strategy

A resilient application can provide assistance to system software layers in still other ways. For example, it could inform the system that a given file it is writing to local persistent storage is a restart dump file. Such files have very special properties which the system can exploit. As an example, they can be formulated so that the system knows how to prioritize staging of them to shared persistent storage. Since these files are necessary for application recovery the system may also provide additional end-to-end data integrity checks to guarantee that they are protected from errors.

Another way in which the system can be aided by a resilient application that incorporates work load redistribution capabilities is for the resilient application to offer, under appropriate circumstances, to either add more nodes to its running job or to give up some of its nodes to a higher priority job. The potential benefit here is more efficient resource utilization than is presently found on most large systems.

4 Developing an HPC Resilience Framework for Extreme Scale

4.1 Current State

When a fault occurs, it may cause an error which leads to a failure. Whenever possible, the system tries to detect the error in the hardware and respond using the entire software stack. Participants recognized however that the number of error producing faults will increase in the Exascale timeframe for most classes of errors. In some cases, such as silent data corruption (SDC), the application itself may be best suited to perform error detection [32]. A resilience framework for the Exascale timeframe must account for the following observations:

1. Users currently presume faults that result in errors occur (relatively) infrequently, and the consequent failure will be detected in the hardware checks such as whenever possible.

2. Since the failures are presumed to be infrequent, diagnosing and taking corrective action does not currently impact performance and energy (much). This assumption needs to be validated.

3. There is an implicit assumption that transient errors occur much more frequently than permanent errors.

4. Only one fault occurs at any point in time in the confined area, the area which can be serviced independent of other faults in the system. Will this assumption continue to be valid?

5. The time to service an error or diagnose a fault needs to be small. That is, mean time to a fault should be much larger than the time it takes to service a fault in order to enable convergence of resource availability to a steady state.

6. Diagnosis of the error to isolate it (location of the error), confinement (so it does not propagate), reconfiguration of the hardware if necessary, recovery and system adaptation are all done in the software stack.

7. All levels in the stack, from applications down to circuits, will need to participate at Exascale.

System level fault-tolerance is currently achieved through two primary means: (1) reactively, responding to the failure to detect an error and recover, and (2) pro-actively, by continually evaluating the system for potential failures, such as aging of components and decommissioning the hardware to prevent future failures. It should be noted however that the general consensus of workshop participants was pro-active fault-tolerance is less well understood than reactive techniques. Though it can play an important role in a resilience strategy, it should be viewed as complementary to reactive fault-tolerance and not as a replacement to it because the accuracy of pro-active techniques remains limited.

To cope with the anticipated error rates and classes in the Exascale timeframe, the application will need to work with the system hardware and software in a coordinated fashion in order to provide effective resilience. Research is required in architecting the resilience manager, a part of the system software, for diagnosis and recovery, in reactive mode as well as proactive testing for future failures. State storage (i.e., check-pointing) and recovery is also an important research topic to implement such a resilience scheme. Most importantly, the assumption that errors undetected by hardware are too rare to worry about may not be valid anymore: New software error detection techniques will be needed.

4.1.1 Hardware Features

Faults can be broadly categorized as permanent (e.g., power supply failure), gradual (e.g., device aging), or intermittent (e.g., noise or soft error). Working groups proposed simple sensors to detect failures in non-integrated hardware, power supplies, fans, and such. The logic in the integrated components (e.g., processor or memory) needs hardware features to detect intermittent errors. The memory (DRAM) is covered by ECC with silent correction, yet errors have reported for tracking. State machine hang-ups caused by control errors also need hardware detection. Confined regions of hardware (such as cores and memory) can be decommissioned by a local or global control system. Remote access by other nodes to local state on a failed node can significantly facilitate recovery.

4.1.2 Reactive Response

A reporting architecture will be necessary not only to identify the existence of the error, but also to point out the location of the error for diagnosis. It is not clear that hardware will detect a sufficiently high fraction of soft errors in the latches of the cores' circuits. When an error is encountered the reporting scheme must interrupt the parent processor as well as the neighbors to notify of the error. If the parent processor core is active then it negotiates with the neighbors about responsibility for diagnosis, otherwise the neighboring processor starts the diagnosis process. The first time occurrence of the error is interpreted as an intermittent error, the instance is logged and the thread is retried or state recovered from the earlier check- point. Subsequent errors of the same type may indicate faulty hardware in the confined area, in which case it is decommissioned and the system is reconfigured.

4.1.3 Proactive Screening

Every processor engages in regular proactive screening when the resources are available, that is when the energy is available, the processor is idle and not in the performance critical path. The screening could be even at predetermined intervals such as hours or days. The screening runs various tests to determine aging of the cores if any, erratic bits in DRAM, any such hardware marginalities, and also reviews history of previous intermittent faults to determine health of the individual components. It may reconfigure the system, such as changing the frequency of the core, reconfiguring DRAM, or may even decommission the region of the hardware and reconfigure the system. Such techniques may be useful for detecting hard errors, but may be less useful for identifying errors that are statistical or transient in nature.

4.1.4 State Storage (i.e., Check-Pointing)

When a error is detected and serviced it may be necessary to restore the entire system, or part of the system, to a known good state for recovery. The state storage should be frequent enough so as not to impact the application performance due to recovery, yet not too frequent to spend too much time and energy. Hence, a hierarchical and incremental approach for state storage and recovery will likely be required. Fault behavior and the frequency of occurrence need to be much better understood to determine efficient check-pointing scheme. It is important that state storage be substantially more reliable than memory in order for it to be effective for fault-tolerance.

4.1.5 Architectural Considerations

The working groups were asked to make detailed assessments of the most likely architecture targets in the Exascale timeframe, and evaluate the implications for resilience in terms of metrics that the working groups themselves defined. The results were as instructive for what they had in common as for the ways they differed. In particular, there was general agreement among the working groups that the right metric for “resiliability”, or the resilience characteristics of a system and its application workload, was expressible as some form of *expected time-cost-energy to a correct solution.* The definition of ”correct” however generated a great deal of discussion and no consensus definition was achieved. Interestingly most of the definitions included some notion of probability or expectation, indicating the statistical nature of the metric. The general sense was that a 10% loss of “resiliability” due to errors was “acceptable”, but 25% or more would not be acceptable.

In terms of system architecture, most of the working groups assumed an Exascale architecture in the 2020 timeframe would consist of on the order 400k chips, 400M ALUs and 300 MW with little or no additional Exascale investment and on the order of 350k chips, 1.2B ALUs and 50 MW with aggressive investment. It was the general consensus of the workshop participants that the mainstream compute industry would not likely address the resilience concerns of HPC. Estimates of MTBF for such systems ranged from 30 seconds to 3 hours depending on failure in time (FIT) rate estimates made independently by each working group. This suggests that more work is needed to improve estimates of likely MTBF, though for back of the envelope estimates these showed remarkable agreement.

Finally, the most compelling observation made by the working groups is that *the basic resilience conclusion do not appear to depend heavily on the system path forward.* This conclusion is a consequence of the belief that failure rate is more strongly correlated to the component count than the number of ALUs. All of the assumed architectures have similar component counts, even though the number of ALUs varies significantly. The observation is particularly important because it suggests that development of a resilience strategy need not be gated on the definition of a final architecture. Resilience research and the implementation of a resilience framework can proceed in tandem with system design and need not create additional programmatic risk or cost if the system design changes.

4.1.6 Proxy Applications

Applications can be characterized in terms of both their susceptibility to faults and their ability to detect those faults. Participants characterized the following three classes of applications in terms of their fault-tolerance behavior:

Non-checking, flexible algorithms are those for which a self-consistency check may be expensive or hard to compute, but the algorithms it implicitly tolerant of errors. For example, a GUPS or needle-in-a-haystack type problem may be able to tolerate some number of false positives or false negatives, and elliptic PDEs may converge iteratively even under failure.

Non-checking, frangible algorithms are those which lack an obvious self-consistency check, but their results are highly sensitive to even small errors. Initial value problems may be an example of such algorithms, but this category was not explored in great detail.

Self-checking, flexible or frangible algorithms are those for which an internal consistency check is easily and cheaply implemented. Linear algebra (e.g., Linpack solving $Ax = b$) is an example of such algorithms since it is computationally inexpensive to check the solution compared to the time to solve.

Attendees agreed that the tradeoffs between application performance and fault-tolerance of these three classes of applications remains an important open issue in the Exascale timeframe.

Self-checking algorithms need system hardware and software support run through errors. Those applications need to have the information and resources to decide for themselves whether or not to continue running. For non-checking algorithms that are frangible, it may be that N-modular redundancy (NMR) solutions will be required to provide detection of errors. Two examples of specific algorithms representative of these general classes were targeted by the working groups in working out the strawman resilience strategy in Section 4.4.

4.2 Gaps

Improving the end-to-end integrity of applications and architectures in a system constructed of unreliable components is a complex endeavor and the current state of the art still exhibits significant gaps, both in terms of the fundamental understanding of the problem itself and the techniques needed to overcome them. Workshop participants articulated this challenge at a high level as a problem of "separation of concerns" [14]. From that perspective, the role of system stack beneath the application is to address hard (i.e., unrecoverable) errors in such a way that they appear to be soft (i.e., recoverable) errors from the application's perspective. The role of the application is then the handling of soft errors.

This simple yet powerful concept means that *the role of the system hardware an software is to insulate the application from unrecoverable errors, such as component failures, while allowing the application itself to determine how to manage recoverable errors, such as corrupted data.* This way, if the system loses a component that the application does not need to perform its work then the system gives the application a means by which it can keep running. For applications that do not wish to participate in their own fault-tolerance, this approach satisfies the "do no harm" dictum, since those applications will run no less reliably as a consequence of their being oblivious to the concerns of fault-tolerance. For applications that are self-checking (ref. Section 4.1), this approach provides a straightforward mechanism by which the developer can intelligently tailor the application's response to various types of system errors. However, certain gaps remain in our ability to implement such a strategy in the Exascale timeframe. The following section lists a set of those gaps, grouped by high-level concepts.

4.2.1 The Need to Better Understand the Problem to be Solved

Foremost, workshop participants identified gaps in understanding the depth and magnitude of the resilience problem, how far current and future systems will be impacted, and what can and will be prevented through new hardware and software techniques. This gap ranges from basic questions on error types and rates to metrics to evaluate their impact as well as the associated costs of protecting against them.

Error types and rates: The HPC community needs a clear classification of which kind of errors exist on current or will exist on future systems. Today's approaches loosely distinguish only a few classes of faults, such as permanent, transient, gradual and silent errors, but this classification needs to be extended and refined. This is a hard to solve problem that is currently not well understood.

Magnitude of the Problem: Additional studies are needed that help document both the magnitude of the problem (in terms of current and expected error rates) as well as the impact of running on unreliable hardware and the impact on software productivity and robustness.

Error Propagation: While some recent work is starting to address questions on how errors and faults propagate in a system, this is generally an open field. Understanding propagation, however, is critical as a step towards whole system evaluations and realistic fault prediction and analysis techniques.

Metrics: There are currently only a limited number of metrics that allow us to characterize the fault behavior and characteristics of both applications and architectures. This, however, will be fundamental to evaluate the reliability of future system and to judge any improvement or success of proposed new techniques.

Cost of Resilience and Fault Tolerance Techniques: Fault tolerance will not be free and we will pay in terms of performance and energy both in fault and fault free cases. The HPC community needs the ability to understand these costs and their tradeoffs in order to make informed tradeoff and design decisions.

Validated Fault Injection: There is currently no community accepted, validated fault injection tool or fault model for the HPC community. Without such tools and models the community has no means for testing fault-tolerance of resilience detection and recovery techniques at extreme scales where testbeds do not exist.

4.2.2 Support for Event Analysis and Notification

One of the fundamental needs for any fault-tolerance technique is the ability to be notified in case of a failure. However, there is no central location where all errors or faults are detected. Instead, every components will need to participate, including but not limited to the underlying hardware from compute cores to the I/O system, runtime and operating systems, libraries and applications as well as through the help of external tools.

Such detection must not be restricted to catching of external events, but can and should also include self-checking software components and algorithms that can detect external influence indirectly. The latter will be essential for detection techniques in applications and numerical libraries.

Once detected, any error or fault must be made available to all agents that need to participate in the recovery process. This requires a system wide, resilient yet light weight publish-and-subscribe infrastructure. It must allow for every component to both deliver events to all other components as well as receive events.

A fault avoidance or fault recovery action may depend on many events, generated at different times and different locations. For example, the imminent failure of a memory may be predicted from the stream of warnings about previously corrected errors; in order to identify a low-performing network link, one may need to correlate information on end-to-end messaging performance, with information on the current routing scheme used; and so on. In addition to the ability to collect and distribute information, one will also need a significant ability for real-time "data fusion" and off-line data mining.

4.2.3 Coordinated Holistic Approach

Already the notification discussion above shows that resilience cannot be achieved in a single location in the overall system, nor is it possible to implement techniques in multiple layers independently. A successful resilience strategy will need a vertically coordinated and integrated approach that gives a holistic view on faults and ways to mitigate them.

Interfaces: A key part will be the development of the necessary interfaces that allow applications to expose their information, state, and detected fault to the other software layers in a standardized and coordinated fashion. This requires the creation of common abstractions and is closely related to the gaps on fault types and metrics above.

Containment and Isolation: In order to provide clean semantics and to reason about errors and their propagation, we will need techniques that provide error containment and isolation between system components and layers.

Coordination: Fault tolerance techniques will be implemented in all layers of the application stack, potentially by different developers and/or vendors. However, in order to be effective and efficient these mechanisms must be coordinated to avoid unnecessary replication of state, inconsistent snapshots of system state, or missing information. This not only refers to coordination between the application, the runtime, and the underlying architecture, but must also include other system components, such as in situ and online data analysis, runtime tools, or the I/O subsystem. *Fault tolerance approaches developed disjointly in an adhoc manner could have significant overhead and yield spotty or ineffective resilience when applied at extreme scales.*

4.2.4 Viable Prediction

Titled in the workshop as “The Holy Grail of Fault Tolerance and Resilience”, an accurate prediction of faults will a substantial aid for system-wide fault avoidance. If prediction is successful and correct, systems will be able to preemptively vacate failing resources, avoiding costly correction or restart techniques. Further, correct predictions of failure rates will help with fine tuning techniques, e.g., by setting optimal checkpoint intervals, and will help concentrate efforts on the most critical and vulnerable resources.

Note that fault prediction is useful, even if it has low precision and recall. Suppose that one can predict in a timely manner half of the node failures, but half of the predictions are wrong. Then one will have avoided half of the restarts, at the expense of twice as many process migrations. This is likely to be a worthwhile trade-off. Current research shows that better precision and recall are already feasible [15]. On the other hand, fault prediction will not be perfect, and will not obviate the need for fault detection and recovery techniques.

4.2.5 Adoption and the Road to It

Implementing fault-tolerance techniques and a resilience strategy will not be effective without changes to applications and system software. Some of these changes may be significant and in some cases will require a rethinking of fundamental software design. This could impact on the sustainability of existing codes, even though the proposed resilience strategy in Section 4.4 was formulated with the intention of supporting the execution of unmodified codes. Such codes would continue to run in the new resilience framework, but may derive little or no resilience benefits without some amount of rewriting.

Application and library developers desiring to fully avail themselves of new resilience features will likely face increased code complexity, and depending on implementation choices may find themselves facing a steep learning curve. In order to overcome hurdles in the adoption of these new techniques, it will necessary to both focus on education and training of developers as well as on the tools and methodologies that enable developers to more easily transition existing applications to a fault-tolerance aware model. And with changes in the programming model looming in the Exascale timeframe, because of parallelism and data locality challenges, *now may be the opportune time to embrace the disruptive changes required to implement a coordinated HPC resilience strategy.*

4.3 Dependencies

In order to build a software stack that enables all the required interdependencies between the separate layers that are necessary for resilience, it is useful to begin with the concept of a fault model, or fault API. The idea is to establish a standard description of faults, so that the software layers can respond appropriately to them. This fault model must include features that allow each layer to be developed separately and in parallel by different teams, without needing to know the implementation details in any of the other layers.

Developing this model may require some experimentation, but the computer science community already is familiar with a very serviceable approach - that of exception handling. This seems a very reasonable starting point. Workshop participants do not intend to minimize the effectiveness of automatic correction of faults, transparent to an application, as one finds with ECC memory or error correcting data transmission protocols. On the contrary, we expect the vast majority of faults to be handled this way. However, these techniques cannot cover every case. Exceptions are then the faults that these techniques do not correct.

4.3.1 Fault Model

A fault model built on exception handling has three main components: detection, notification, and recovery. Fault detection can occur at multiple layers of the software stack, including even the top, application layer. One may expect the granularity of fault detection to increase as we ascend from layer to layer, with the application responding mainly to relatively large faults, such as the loss of a node or a small, undetected error that has propagated extensively and ultimately led to some physically impossible behavior that the application can detect.

Fault Types	**Examples**	**Effects**	**Today**	**Exascale**
Permanent	Fans, Power supply, Chip	Hardware fails	EXISTS	WILL EXIST
Gradual	Spatial and temporal (temperature, voltage, external sources, etc)	Slow down, speed up, data corruption, loss of control	DESIGNED OUT	WILL EXIST
Intermittent	Soft errors (cosmic rays, packaging), Noise (voltage droops)	Data corruption, silent, loss of control	EXISTS	WILL EXIST

Table 3: Fault types: today vs. Exascale

A well designed fault model will enable post-mortem root cause analysis, so that equipment can be replaced or software bugs fixed. For the application level, the idea is to provide fault notifications with sufficient information to allow a willing application to take appropriate action. For example, an API in which the application could specify the following set of capabilities by task, work ID, and work phase provides the basis of an uncomplicated, flexible and powerful interface:

Persistent State Storage: provides the application an ability to identify and allocate storage blocks that will persist through node failures.

Ability to Recover This State: provides a mechanism for automatically recovering program state among neighboring nodes.

Notification: provides information on what is happening at the system level (e.g., lost data, lost resources, degraded operation, etc.) in response to the error.

Log Messages: provides an interface for the application to start, stop and update its assessment of the state of a computation.

Applications that are not designed to take such action will be killed, presumably to fall back upon a restart dump that they will have written earlier. Application developers will be able to decide whether or not faults are frequent enough and costs of roll-backs and restarts are high enough that they should provide the appropriate exception handlers to deal with these faults immediately as they occur. Thus, even applications that leverage none of the failure handling infrastructure supplied by the resilience framework can still run normally.

4.3.2 Fault Notification

At the application level, it will be possible to respond to or ignore fault notifications, but at other layers of the software stack this will not be an option if the entire system is to function properly. For example, if the response of the messaging layer is to kill the job upon the loss of a node then it is pointless for an application to be resilient to such node loss. Here we have no chicken and egg problem - it is absolutely clear where the work of resilience must begin.

We can thus imagine a process for enabling resilience that begins at the lower layers of the software stack and builds upwards. Only considerably into this process would it make sense to build truly resilient applications. Nevertheless, a small number of representative applications might volunteer to be resilience pioneers and to develop the necessary exception handlers in parallel with the changes to the lower levels of software. Such early resilient applications would most likely be necessary for testing the effectiveness of the overall changes to the software stack.

4.3.3 Containment Domains

In this picture, we consider a chain of resilience dependencies that begins at the bottom and extends up to the application layer of the software stack. In order to recover from a fault, one needs to identify the limits in hardware or in data that contain the fault (a “containment domain”). At present, this containment domain is the entire job. For resilience to be effective, this domain must become smaller. We can also think of such containment domains in terms of the layers of the software stack. Thus we could design the fault model in such a way that particular kinds of faults are contained within certain layers or groups of layers. This would enable each layer to respond to only a limited subset of all faults identified in the fault model.

For example, one might demand that data transmission faults on the system interconnect be handled by the messaging layer, so that at the application layer these faults would not need to be either detected or handled. An aggressive application design might nevertheless notice slow data delivery to a particular node and then exclude that node from the working set, but this instrumentation of the application would be optional, and not required for resilient operation. The total loss of all connection to a given node could be signaled to the application as a loss of the node itself, since the action to be taken by the application would be the same regardless of why the node was lost. In this fashion, the amount of new work required at each level of the software stack could be kept manageable.

4.3.4 Fault Characterization

As a start in identifying fault types and which layers of the software stack must deal with them, we may sort faults according to Tables 4a–c. The first two tables serve to expose those hardware faults that we can clearly anticipate. The third table captures what we believe we know today about

Component	**Prob. of Fail**	**Impact**	**Scope**
Fans	High	Low *	Node down
Power Supply	High	Low *	Node down
CPU / SRAM	Low	Low *	Node down
DRAM	Mid	Low	Reconfiguration (ex: page mapped out)
Solder Joints	High	Low *	Node down
Sockets	High	Low *	Node down
Disks	Mid to High	Low	Reconfiguration (ex: rebuild)
NAND / PCM	Low	Low	Reconfiguration (ex: rebuild / page mapped out)
Network	Low to Mid	Low	Reconfiguration (ex: dynamic route)

(a) Permanent and gradual hardware faults at Exascale

Component	**Prob. of Fail**	**Impact**	**Scope**
Fans	-	-	-
Power Supply	-	-	-
CPU / SRAM	Unknown	High	Soft error / noise / data corruption
DRAM	Unknown	High	Soft error / data corruption
Solder Joints	-	-	-
Sockets	-	-	-
Disks	Mid to High	Low	Loss of access
NAND / PCM	Unknown	High	Soft error / data corruption
Network	Low	Low	Noise / data corruption

(b) Intermittent hardware faults at Exascale

Component	**Prob. of Fail**	**Impact**	**Scope**
Node OS	Unknown	Low	Node
Storage	Unknown	High	System
Runtime Intra-node	Unknown	Low	Node
Runtime Inter-node	Unknown	High	System
Scheduler	Unknown	High	System
Programming System (ex: compiler)	Unknown	Low	Application
Application	Unknown	Low	Application
Tools	Unknown	Low	Variable
Vis and Data Analysis	Unknown	Variable	Variable

(c) Software-related faults at Exascale

Table 4: A partial fault characterization for Exascale

software-related faults, and in particular it is clear that we do not yet understand the probability of failure of software components. The hard work of determining which layers of the software stack should deal with hardware failures, and in which ways, is left to a resilience strategy that will result from research. Nevertheless, exception handling and fault containment are general principles that we expect such a strategy to apply to this problem.

4.4 Strategy

Any strategy for resilience will ultimately involve tradeoffs. Figure 2 is helpful in visualizing those tradeoffs in terms of existing fault-tolerance techniques. In particular, more general techniques such as triple modular redundancy (TMR) can be very easy to implement, from the application perspective, but are expensive from the perspective of system resource utilization. On the flip side, fault-tolerant Linpack (FT-HPL) and various techniques in algorithm-based fault-tolerance (ABFT) can be extremely efficient in terms of system utilization, greatly reducing the performance and energy impacts of the fault-tolerance scheme, but are often application specific. Thus, the resilience strategy exercise consists in trying to develop a balanced approach that accommodates the needs of both the application and the system.

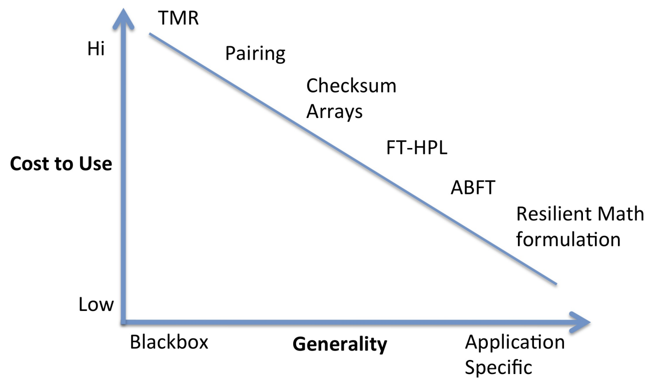


Figure 2: More general fault-tolerance techniques require greater overhead but are also more flexible

4.4.1 Focus on Real Application Requirements

The charge to the working groups was to let the needs of the application drive the requirements for the resilience strategy. The goal of this approach is to develop a *complete* resilience strategy for the "predictive science" and "non- predictive science" domains by first focusing on the particular requirements of two "proxy applications" chosen to reflect the subject matter expertise of the workshop participants. For these purposes, the strategy working groups focused on GMRES [45] as one proxy application and PPM [44] as a second. Both of these are mature computer kernels representative of important resilience challenges facing the HPC community in the Exascale timeframe. The working groups evaluated their resilience strategies in terms of these two proxies, which attendees believe will be generalizable across the target application domains.

4.4.2 Transactional Semantics

The strawman strategy that follows was put forward by the workshop participants as a starting point for attacking the resilience problem. The purpose is not to confine the research space or identify *the* solution. Rather, we aim to present a different thought process on how to structure the application and the system according to a different execution model that promotes failure containment and isolation. This exercise should serve as an example of how to attack the problem with a holistic approach that includes all system layers.

The proposed strategy is based upon an abstract entity called the *Recovery Unit* (RU) (see Figure 3,, which is the unit of failure and recovery in the system. This is a construct similar to the

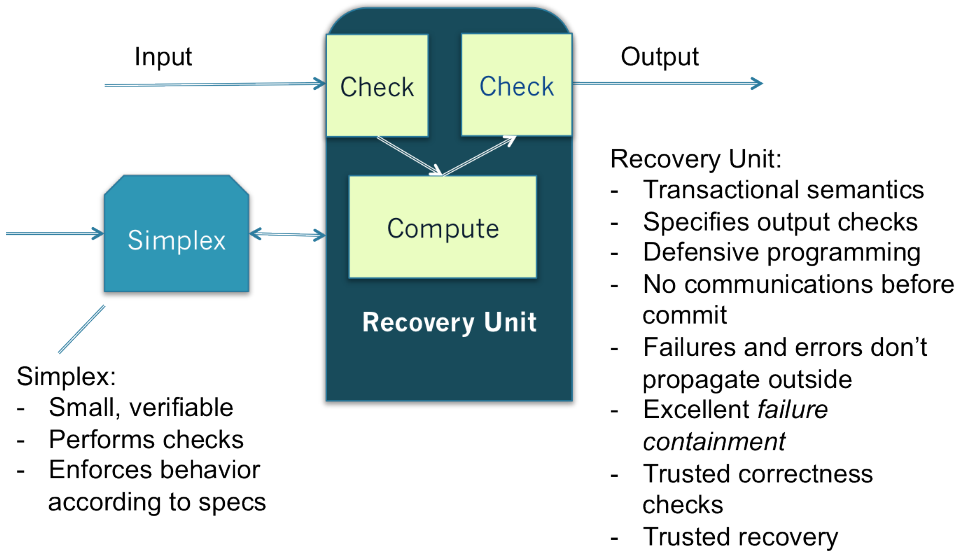

Input
Check
Check
Output
Simplex
Compute
Recovery Unit
Recovery Unit:
- Transactional semantics
- Specifies output checks
- Defensive programming
- No communications before commit
- Failures and errors don't propagate outside
- Excellent *failure containment*
- Trusted correctness checks
- Trusted recovery
Simplex:
- Small, verifiable
- Performs checks
- Enforces behavior according to specs

Figure 3: A diagram of a Recovery Unit (RU) that serves as the backbone of the proposed resilience strategy for an extreme scale HPC system in the Exascale timeframe.

containment domains proposed by Mattan Erez at. al [42]. The RU has transactional semantics in the sense that it executes a program in the form of a sequence of atomic actions, each receiving input and producing output according to program control and semantics. (Logically, internal state can be viewed as an output of the action that is an input to the next action.) Its implementation should thus promote defensive programming on the part of the programmer.

Each action within the RU is checked at input and output stages to ensure that all errors are detected before it can affect the state of other RUs. Thus, failures and errors will not propagate outside the boundaries of the RU. The error checking presents a tradeoff to the programmer, where more effort spent can lead to easier time during debugging and more resilient operation. Such an effort-reward system is different from today's practice where resilience is essentially an after-thought. The approach also can naturally extended to include security provisioning.

RU's can be defined statically, as part of the system structure, or dynamically, as part of the execution evolution; they can be single level, or hierarchical. The system will consist of a number of RUs (likely a large number), with provisions in place to ensure that:

1. If an RU fails, it is simply replaced by redirecting the input toward another functioning RU. No global rollback or aborts are necessary.

2. Once the output of the RU is produced, it will persist future failures. This provides a mechanism to advance the computation through failures in a scalable manner. To achieve this goal, it may be necessary to store the output in stable, persistent storage, until it is consumed.

4.4.3 Error Containment

The system is resilient if errors are not allowed to propagate from RU to RU and state is properly preserved. To achieve this goal The RU includes a "simplex" subsystem that takes as input some specification of the expected behavior of the application. The input would preferably be composed of a formal specification of the system behavior. Less than ideally, the input could consist of a number of assertions that are used to verify the system execution. The simplex is responsible for checking the input and output into and out of the RU, respectively, to ensure conformance with expected behavior. (Input checking can be avoided if communication is reliable.)

For example, the simplex may verify that an index to an array as an input parameter is within the bounds of legal values. Similarly, the programmer may specify that the expected value of an output variable lies in a certain range, and the simplex will verify that indeed the output is within the stated range. The simplex is likely to be implemented in a manner where its correctness is verifiable (e.g. a small operating system kernel working on an adjacent simple processor that can be cheaply hardened). The checkpoint and restart mechanisms need to be similarly hardened.

The interface to the programmer is not very different from the proven transactional model. The system should communicate failures upward and extensive use of exception handling must be provided to enable the program within the RU to rectify the situation instead of merely "killing" the program. Of course, the program within the RU will be required to perform best effort resolution of the problem before it declares to the system that a failure has occurred, at which time the RU is taken off line and substituted with a spare.

4.4.4 Efficiency and Overhead

The system is efficient if the overheads of reliably testing and storing outputs is low. A naive implementation can have unacceptable overheads. Various optimizations will be needed to achieve acceptable performance. For example, application state is often redundant. This allows one to reduce the amount of persistent storage needed, possibly at the expense of a more expensive process to recreate lost values. Also, one can use hierarchical designs, where frequent errors are avoided with fine grain RUs while infrequent errors may require the re-execution of a coarse grain RU (possibly, an entire application).

The efficiency of the system will also depend on the frequency of errors and the number of RUs affected by an error. The later can be reduced by matching the boundaries of logical RUs, with the physical boundaries of errors. A hardware error can affect a single processor chip, or a single node containing several processor chips that share memory; Accordingly, RUs should be mapped, to single chips or single nodes. On the other hand, an error in the communication network can have a global effect. This can be mitigated by ensuring that unrecoverable communication errors in the network are very unlikely, or by compartmentalizing communication.

In particular, new persistent memory technologies should provide opportunities to write information to a stable storage device at an unprecedentedly low performance overhead. The approach also can leverage existing technologies within the confines of the RU, such as local checkpointing for long transactions, virtualization, etc.

4.4.5 Flexible Tradeoffs

The RU has to allow the user to specify a flexible tradeoff among performance, energy consumption and resilience. Flexibility in providing these measures is therefore a *necessary* tenet for building the system. The system should allow the user thus to have "knobs" to identify the appropriate measure most appropriate for a certain applications, with the understanding that not all applications require the same level of resilience, or are of the same priority. The concept can also be taken at various components of the application, affording increased resilience and the associated increase in energy consumption to the application component most critical to correctness, and perhaps less so for other components deemed as more inherently resilient or whose compromise may not affect the quality of the result.

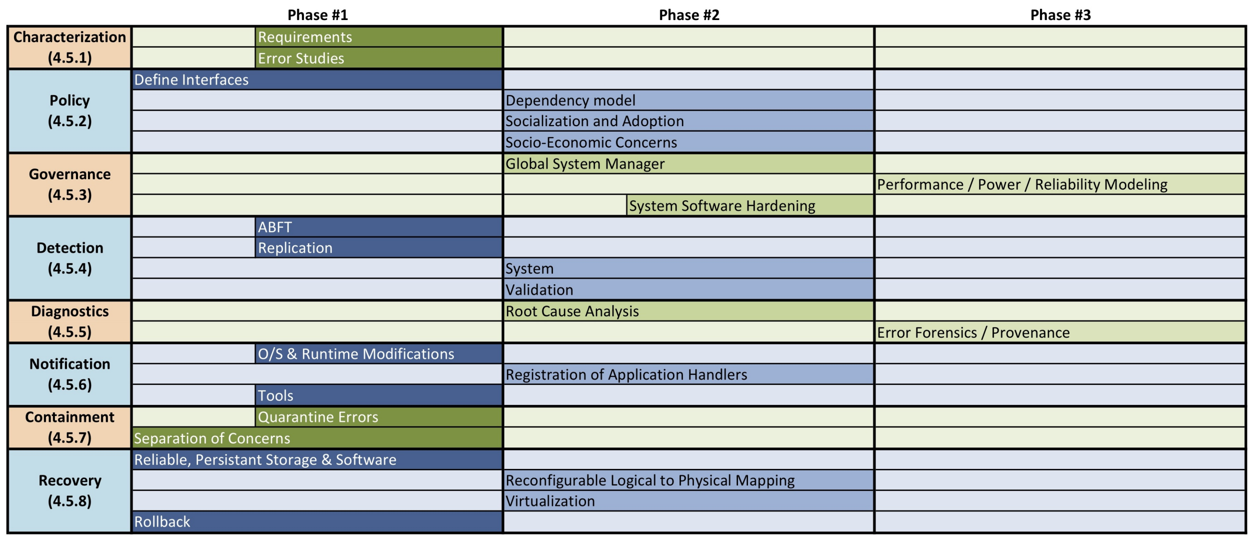
A transition to a new execution model is certainly not a simple undertaking. There is research that must be performed not just to validate the strategy but to also automate the transformation from the existing execution models such as MPI, Map-Reduce and PGAS, into the new model. Such automation is necessary to preserve the previous investments in software and to move this software to a new era where it can be made more amenable to sophisticated treatment in the performance-power-resilience tradeoff.

There are additional advantages to the approach we present here as an example. It enables the programmer to specify resilience and reason about the properties of the application. This has the nice byproduct of simplifying the process of application debugging, as well as improving maintainability and robustness of code. Further, it provides a stable, well compartmentalized system software stack to facilitate reasoning about and diagnosing bugs arising during the system stabilization period subsequent to initial operating capability (IOC).

4.5 Roadmap

The final workshop activity was roadmap development, to identify R&D activities necessary to realize essential components of a resilience infrastructure appropriate to extreme scale system in the Exascale timeframe. Each topic lists specific tasks. After each topic is a number (1, 2 or 3) showing approximate phasing and dependencies (i.e., Tasks in phase #1 should be completed before moving ahead with tasks in phase #2), followed by a priority (Med or High – low items are not listed). Sometimes a letter is included with the number (e.g., 1b) to identify subordering of tasks that may be conducted concurrently.

Topics are further divided into sublists of finer grained tasks. Participants tagged tasks by their approximate cost ($, $$, $$$ or ? when uncertain) and were asked to differentiate between "little-r" and "big-R" research tasks (r/R) as a coarse measure of technical risk. This distinction corresponds to a relatively higher or lower technology readiness level (TRL) [2] and should be helpful in tailoring a "right-sized" resilience strategy for a program developing its technology portfolio for the Exascale timeframe. Finally, all of the roadmap tasks from this section are summmarized in timeline format in Figure 4.



	Phase #1	Phase #2	Phase #3
Characterization (4.5.1)	Requirements		
	Error Studies		
Policy (4.5.2)	Define Interfaces		
		Dependency model	
		Socialization and Adoption	
		Socio-Economic Concerns	
Governance (4.5.3)		Global System Manager	
			Performance / Power / Reliability Modeling
		System Software Hardening	
Detection (4.5.4)	ABFT		
	Replication		
		System	
		Validation	
Diagnostics (4.5.5)		Root Cause Analysis	
			Error Forensics / Provenance
Notification (4.5.6)	O/S & Runtime Modifications		
		Registration of Application Handlers	
	Tools		
Containment (4.5.7)	Quarantine Errors		
	Separation of Concerns		
Recovery (4.5.8)	Reliable, Persistant Storage & Software		
		Reconfigurable Logical to Physical Mapping	
		Virtualization	
	Rollback		

Figure 4: Roadmap for resilience R&D indicating phases for various research tasks as the stand in relationship to one another in the Exascale timeframe.

4.5.1 Characterization

Requirements (1b, High): Elicit "real" requirements of different kinds of algorithms and applications

- For a very limited set of applications [**r/$**]
- For a broad set of applications [**R/$$**]

Error Studies (1b, Med): Identify what kind of errors exist in the wild that we do not even know about (inquisit people and codes, do experiments) to gain a better understanding of the types of errors we are up against

- Obtain very limited set of errors [**r/$**]
- Obtain comprehensive set of errors [**R/$$$**]

4.5.2 Resilience Policy

Define Interfaces (1a, High): Create a resilience interface for applications users and developers to complement exception handling that already exists (e.g., "try" / "except" in Python)

- Define and develop the interfaces [**r/$**]
- Determine whether and how well the interfaces work [**R/$$**]

Dependency Model (2, Med): Develop full system / full stack model to define dependencies [**R/?**]

Socialization and Adoption (2, High): Buy-in from application writers, open source community, facilities and system vendors [**r,R/?**]

Socio-Economic Concerns (2, Med): How do we convince the community to use this? [**R/?**]

4.5.3 Governance Mechanisms / Framework

Global System Manager (2a, High): Create a resilience manager - every system component must participate

- Current crash recovery mechanisms & RAS system, embedded systems, dependable software technologies [**r/$**]
- Making it efficient and cost effective [**R/$$**]

Performance / Power / Reliability Model (3, Med): Develop a model to define tradeoffs [**R/$$**]

System Software Hardening (2b, High): Address the issues that the system software must be at least as fault-tolerant as the applications it is protecting from errors

- Adopting architecture patterns and practices for dependable software [**r/$$**]
- Restructuring of system software (ref. Section 4.4) [**R/$$$**]

4.5.4 Detection (i.e., Something Happened)

ABFT (1b, Med-High): Algorithm-based fault-tolerance schemes that permit applications to self- detect and self-correct errors

- Focus on very limited algorithms [**r/$**]
- Apply more generally [**R/$$**]
- Automatic FT code generation [**R/$$$**]
- Implement selective reliability [**R/$$**]

Replication (1b, High): Implement N-modular redundancy schemes in system software to run multiple instances of an application and compare results

- Manual implementation or coarse grained auto- replication [**r/$**]
- Automatic implementation and optimization [**R/$$**]

System (2, Med-High): System level error self-checking

- SECDED, parity, checksum, etc. [**r/$**]
- Chinese remainder theorem [**R/$$**]
- Performance, energy or cost impact and automated optimization [**R/$$$**]

Validation (2, High): Creating methods and infrastructure for measure fault-tolerance of algorithms and effectiveness of a resilience strategy

- Testing infrastructure for resilience [**r/$$**]
- Formal methods - possibly very hard [**R/$$$**]

4.5.5 Diagnostics (i.e., What Happened?)

Root Cause Analysis (2, High): Working backwards from the failure to determine the fault

- Historical (post-mortem) or manual [**r/$**]
- Real-time, automated [**R/$$**]

Error Forensics / Provenance (3, Med): Working forward from faults to all potential errors

- Historical (post-mortem) or manual [**r/$**]
- Real-time, automated [**R/$$$**]

4.5.6 Notification

O/S & Runtime Modifications (1b, High): Changes to the software stack necessary to support information exchange from hardware to the application

- Getting more sophisticated about posting events [**r/$**]
- Bounding the errors and doing attribution [**R/$$**]

Registration of Application Handlers (2, High): Mechanisms to allow the application to communicate error classes for which it desires notification

- Handling registration [**r/$**]
- What types of things need to be registered [**R/$**]

Tools (1b, High): To notify the users and developers and administrators of system state

- Notification system [**r/$$**]
- Analysis and reporting [**R/$$**]

4.5.7 Containment

Quarantine Errors (1b, Med): Prevent further propagation of errors once detected

- Hardware vendors do this to some extent on a component basis [**r/$$**]
- Across the system or across the stack (R) [**R/$$**]

Separation of Concerns (1a, High): System software and tools should do no harm to application; application should do no harm to system software and tools

- Leverage security solutions [**r/$**]
- Containment domains to prevent irresponsible use of things like RDMA [**R/$**]
- General strategies for component interactions and implementation – this is going to touch every other aspect of the resilience framework [**R/$$$**]

4.5.8 Recovery

Reliable, Persistent Storage & Software (1a, High): Hardware and supporting software to facilitate the error recovery process

- Local Storage [**r/$**]
- New global high performance, resilient solutions as current solutions are inadequate for Exascale with existing applications [**R/$$$**]

Reconfigurable Logical to Physical Mapping (2, Medium): Necessary to support reallocation and rescheduling of resources after a failure

- Reallocation mechanism for preallocated resources [**r/$**]
- Dynamic coordinating and optimizing [**R/$$**]

Virtualization (2, Medium): Facilitates movement of an application away from failed resources

- Static translation [**r/$**]
- Dynamic translation [**R/$$**]
- Extending the concept through the execution model [**R/$$$**]

Rollback (1a, High): Mechanisms that return the system or application to a known good state

- Global and local with existing transactions [**r/$**]
- General local [**R/$$**]

5 Summary and Conclusions

As the HPC community embarks on the path to Exascale computing, several uncertainties remain. Perhaps the most glaring of these is the status of resilience eight or more years out at a system scale not contemplated before. This report outlines the issues that need to be tackled to fully address those uncertainties and provide a path forward for computing in the face of increasingly unreliable systems. It dictates a roadmap and should motivate an active research and development agenda to meet the needs of the community and realize the goals of computing in the Exascale timeframe.

5.1 Scope of the Challenge

The collective opinion expressed by workshop attendees predicts that the number of errors, and particularly soft errors, occurring in HPC systems will continue to rise with deeply scaled process technologies. The situation is exacerbated by the power consumption challenge that these systems must meet to produce a system with realistic resource and operational parameters. Sub-threshold voltage operation, increased heat concentration and continuous change in voltage will be needed to produce optimal energy consumption, but will create problems of intermittent failures and may also affect the longevity of a system's components.

However, the resilience issues are not confined only to hardware. At the software level, the predominant programming models today are based on unconstrained message traffic, a model characterized by poor error containment properties. Thus the failure of one component can propagate quickly and recovery generally requires a total system abort. This arrangement was found inefficient at Terascale, problematic at Petascale, and we predict will be unworkable at Exascale. A programming model that is based truly resilient, allowing applications to continue running to correct solutions in spite of errors, it what is needed.

Software lacks a robust application interface to failures and recovery. Today, programmers deal with systems that provide unrealistic abstractions of perfect operation. Failure is modeled with clean semantics of an abrupt stoppage that does not cause any permanent corruption of data. Immediate error detection is assumed in the abstractions available today, and thus techniques such as synchronized global checkpointing have been used to restore a sane system state and resume computing. With the rise of error rates at the component level and at the system scale level, the timely detection of these errors during runtime is going to be of paramount importance and is a challenge by itself.

Further, programmers have no tools to either predict or assess the fault-tolerance of an application. Faced with these challenges, programmers have resorted to adhoc techniques that are programmed into their software to provide rudimentary recovery when an error occurs. However, this often leads to code obfuscation and reduced robustness.

5.1.1 Building on Success

It is important in the midst of discussing all the work to be done that the HPC community not lose sight of the good news and sometimes surprising successes. For example, the fact to the resilience strategy can be effectively decoupled from final system architecture decisions, because reliability is effectively proportional to component count, could be a huge win. It means that work can proceed toward development of a practical resilience infrastructure long before final details of the Exascale system architecture have been nailed down. It was also a welcome relief to discover that even though global, shared checkpoints are a non-starter in the Exascale timeframe, multiple solutions based on concepts of writing checkpoints to some form of local or nearest neighbor storage appear that they will be viable in the Exascale timeframe.

Other areas where the workshop uncovered hopeful indicators included the area of application fault-tolerance. Workshop participants discovered several classes of applications that were either "embarrassingly fault-tolerant" or at least had the potential to self-check and even self-correct for errors. This will translate into less work to be done at the system level to capture and correct soft-errors, and more efficient methods of error detection and recovery. In fact, it did not appear to the experts in attendance that there was any class of applications of interest to the DOD or DOE that could not substantially improve their ability to tolerate errors through careful consideration of algorithms and methods. And perhaps even more surprising was the fact that a great deal of the communication that needs to occur between the system and the application to negotiate errors can be accomplished via a comparatively simple and non-disruptive assertion based interface.

As a result of the findings of this workshop, there is good reason to be confident that a practical resilience framework is implementable at reasonable cost in the Exascale timeframe. By starting with what already is known to work reasonably well and focusing attention on the "low hanging fruit" it should be possible to implement a resilience strategy that, while it will not address every conceivable error scenario, will nevertheless demonstrate quantifiable improvements in both user productivity and time and energy to a correct solution on systems that encounter failures frequently.

5.1.2 An Overarching Strategy to Address the Challenges

Workshop attendees identified a strategy to address the challenges outlined above, and a roadmap of technical milestones on the way to formulating a workable resilience framework for the Exascale timeframe. To begin with, it is imperative to assess the situation of component resilience, and therefore work in fault characterization will be necessary to understand the impact on application performance and correctness. A taxonomy of failures and their rates needs to be developed. Fault detection needs a more realistic abstraction to enable the development of more effective failure recovery.

Further up the system stack, resilience mechanisms should exploit application characteristics to improve fault-tolerance and reduce overhead. We identified three classes of fault-tolerant algorithms, namely those that are inherently fault-tolerant, those that enable self-checking, and those that have nefarious characteristics that prevent self checking or resilient operation. Work is needed in establishing the application patterns of each class, and there is a need for transformation that move an algorithm from less fault-tolerant class to a better one.

At the programming model level, there is a need for strong error containment characteristics. We propose to revisit programming models with transactional semantics, assertion checking, self-checking and facilities for failure recovery. These models should allow the programmer to interact with system notifications of failure at a high level, enabling redirection of resources or application-level specific actions. Interfaces that enable this two way communications between the system and the application need to be developed. Such interfaces not only allow the user to improve reliability of the hardware and software components most critical to their application, but in cases where the application is embarrassingly fault-tolerant such interfaces provide users the flexibility to "trade off" reliability for energy efficiency or performance. There is also an important consideration of ensuring that error management is not communicated at such a low level so as to make programming more complex and reduce programmer productivity.

The proposed solutions for software handling of fault detection and for low-overhead fault correction, presuppose a fairly sophisticated set of system services for resilience. There is a real risk of "solving" the problem posed by rare hardware failures by implementing a software infrastructure that is even more failure-prone. The need for more sophisticated energy management and more dynamic resource management exacerbates the problem. Therefore, it will be essential to invest in robust designs for the Exascale software stack, ensuring that the inherent unreliability of large, complex and concurrent system software does not increase the unreliability of the system.

Finally, resilience will require developing tools that can be used to quantify the effects of failure and recovery, predict the inherent resilience of an application, and enable the simulation of faults to support code development, test and debugging. Developers who have years of finely honed intuition with regards to performance tuning cannot be expected overnight to tackle the challenges of implementing fault-tolerant algorithms without robust, fault-tolerant tools that give genuine and useful insight into the types of errors they will be facing and the impact of those errors on the performance and correctness of their applications. Validated fault injection tools are needed in particular to simulate all classes of faults, for the purpose of hardening both system software and applications.

5.1.3 Cultural Hurdles in the Commodity Space

Workshop attendees identified an important cultural issue that may pose a hidden hurdle to progress in resilience. It has been assumed at various Exascale forums that the industry will work to ensure that systems are going to be resilient, out of necessity as commercial customers will not put up with unreliable systems. Therefore, it is assumed that the "resilience problem will be solved". This line of reasoning points to the long history of HPC systems and how industry has always solved the resilience problem by providing what are basically reliable systems. When you have reliability, resilience is not needed. And even as reliability has seen a marked decline on some of the communities largest HPC systems, vendors have usually managed to step up to the plate with a fault-tolerance solution to compensate.

While this basis of confidence has some historical precedent, it will ultimately lead to false hopes and expectation as the needs of the commodity computing market continue to diverge from needs at the highest end of HPC. Thus, it is important to understand the unique characteristics of HPC especially at Exascale compared to commercial computing. As a consequence, industry participants of the workshop unanimously stated that they do not see commercial systems growing at the scale required to meet Exascale requirements. Even very large cloud based systems will be carved up to serve many independent workloads. Therefore, the notion of a single capability system that uses all its resources to solve a single problem is unique to HPC, and there will be no specific motivation to over-engineer those systems to meet the resilience requirements at Exascale. Cost and power were cited as particular concerns.

5.2 Final Recommendations

In summary, the number of errors, particularly soft errors, occurring on HPC systems will continue to increase. A right-sized and well-conceived resilience strategy in the Exascale timeframe will be ultimately far more cost effective for HPC than continuing to rely on ad-hoc resilience solutions. The industry will not do it as they do not see the incentive, and therefore it is important for the HPC community to invest in researching the unique problems of HPC at extreme scale. Academia, government, national laboratories and industry must work in concert to develop new, cost effective approaches to resilience beyond what will be available from "commodity" systems in the Exascale timeframe. Workshop participants identified six technical areas from which to launch such an effort: fault characterization, error detection, fault-tolerant algorithms, resilient programming models, fault-tolerant system services and tools. Addressing these areas will prove a smart risk mitigation strategy for HPC in the Exascale timeframe and a necessity for realizing the full potential of computing at extreme scale.

References

[1] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *The International Parallel and Distributed Processing Symposium*, Long Beach, CA, 2007.

[2] N. Azizian, S. Sarkani, and T. Mazzuchi. *A Comprehensive Review and Analysis of Maturity Assessment Approaches for Improved Decision Support to Achieve Efficient Defense Acquisition*, volume II. 2009.

[3] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in linda. *IEEE Trans. Parallel Distrib. Syst.*, 6(3):287–302, Mar. 1995.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of SC09*, Nov. 2009.

[5] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. P. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar. The CnC Programming Model. *Journal of Scientific Programming*, 2010.

[6] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov. 2009.

[7] Z. Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 73–84, New York, NY, USA, 2011. ACM.

[8] C. M. Christensen. *The innovator's dilemma: when new technologies cause great firms to fail.* Harvard Business School Press, 1997.

[9] J. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):293–302, 2006.

[10] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High Performance Linpack Benchmark: A Fault Tolerant Implementation without Checkpointing. In *ICS 11 Proceedings of the international conference on Supercomputing*, pages 162–171, 2011.

[11] N. DeBardeleben, J. Laros, J. T. Daly, S. L. Scott, C. Engelmann, and B. Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. Whitepaper, Dec. 2009.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41(6), 2007.

[13] J. B. Dennis and D. P. Misunas. A computer architecture for highly parallel signal processing. In *Proceedings of the 1974 annual ACM conference - Volume 2*, ACM '74, pages 402–409, New York, NY, USA, 1974. ACM.

[14] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[15] A. Gainaru, F. Cappello, J. Fullop, S. Trausan-Matu, and W. 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*, page 4. ACM, 2011.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[17] G. Grider. Exa-scale fsio - can we get there? can we afford to? Presented at the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O, May 2011.

[18] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS-III*, pages 54–63, New York, USA, 1989. ACM.

[19] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, and J. Dongarra. Cifts: A coordinated infrastructure for fault-tolerant systems. *Parallel Processing, International Conference on*, pages 237–245, 2009.

[20] S. Han, H. A. Rosenberg, and K. G. Shin. Doctor: An integrated software fault injection environment, 1995.

[21] HDFS. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/core/docs/r0.18.0/hdfsdesign.html.

[22] K.-h. Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984.

[23] G. Kanawati, N. Kanawati, and J. Abraham. Ferrari: a flexible software-based fault and error injection system. *Computers, IEEE Transactions on*, 44(2):248 –260, feb 1995.

[24] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, and Others. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, University of Notre Dame, CSE Dept., 2008.

[25] J. R. Larus and R. Rajwar. *Transactional Memory.* Morgan & Claypool, 2006.

[26] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons learned at 208k: towards debugging millions of cores. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[27] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Proceedings of the 2012 IEEE Conference on Massive Data Storage*, Pacific Grove, CA, Apr. 2012 (to appear).

[28] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system(ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE '08)*, pages 15–24, 2008.

[29] F. T. Luk and H. Park. Fault-tolerant matrix triangularizations on systolic arrays. *IEEE Transactions on Computers*, 37(11):1434–1438, 1988.

[30] W. Ma and S. Krishnamoorthy. Data-driven Fault Tolerance for Work Stealing Computations. In *ICS 12, Proceedings of the international conference on Supercomputing*, 2012.

[31] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 45–54, New York, NY, USA, 2009. ACM.

[32] S. E. Michalak, A. J. DuBois, C. B. Storlie, H. M. Quinn, W. N. Rust, D. H. DuBois, D. G. Modl, A. Manuzzato, and S. P. Blanchard. Neutron Beam Testing of High Performance Computing Hardware. In *IEEE Radiation Effects Data Workshop*, pages 1–8, 2011.

[33] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 2010 (SC10)*, November 2010.

[34] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, Mar. 2011.

[35] P. C. Roth, D. C. Arnold, and B. P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *SC '03*, Phoenix, AZ, 2003.

[36] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, A. Bramnik, R. Allmon, V. Anbrose, and Q. Shi. Soft error susceptibilities of 22 nm tri-gate devices. In *Proceedings of 2012 IEEE Nuclear and Space Radiation Effects Conference*, NSREC '12, 2012. Forthcoming.

[37] N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra, B. Gill, and J. Maiz. *Radiation-induced soft error rates of advanced CMOS bulk devices*, volume 44, pages 217–225. 2006.

[38] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[39] A. G. Shet, W. R. Elwasif, S. S. Foley, B. H. Park, D. E. Bernholdt, and R. Bramley. Strategies for fault tolerance in multicomponent applications. *Procedia CS*, 4:2287–2296, 2011.

[40] J. Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08*, pages 277–288, New York, NY, USA, 2008. ACM.

[41] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer. Nftape: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *In Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 91–100, 2000.

[42] M. Sullivan, D. H. Yoon, and M. Erez. Containment domains: a full system approach to computational resiliency. Technical Report TR-LPH-2011-001, Department of Electrical and Computer Engineering The University of Texas at Austin, 2011.

[43] J. P. Walters, R. Kost, K. Singh, J. Suh, and S. P. Crago. Software-based fault tolerance for the maestro many-core processor. In *Proceedings of the 2011 IEEE Aerospace Conference*, AERO '11, pages 1–12, Washington, DC, USA, 2011. IEEE Computer Society.

[44] P. R. Woodward. The piecewise parabolic method (ppm). *Journal of Computational Physics*, 201:174–201, 1984.

[45] M. young Kim, S. jae Kang, and J. hyeok Lee. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Scientific Statistical Computing*, 7:856–869, 1986.

[46] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. Daft: decoupled acyclic fault tolerance. In V. Salapura, M. Gschwind, and J. Knoop, editors, *19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010), Vienna, Austria, September 11-15, 2010*, pages 87–98. ACM, 2010.