SANDIA REPORT

SAND2013-3790
Unlimited Release
Printed May, 2013

Investigating An API for Resilient Exascale Computing

Jon Stearley, James Tomkins, Patrick Bridges, John VanDyke, Kurt B. Ferreira, James H. Laros III

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2013-3790
Unlimited Release
Printed May, 2013

Investigating An API for Resilient Exascale Computing

Jon Stearley, James Tomkins, Jon Vandyke, Kurt B. Ferreira and James H. Laros III
Scalable Computing Systems
Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-1319

Patrick Bridges
University of New Mexico
Department of Computer Science
Mail stop: MSC01 1130
1 University of New Mexico
Albuquerque, NM 87131-0001

Abstract

Increased HPC capability comes with increased complexity, part counts, and fault occurrences. Increasing the resilience of systems and applications to faults is a critical requirement facing the viability of exascale systems, as the overhead of traditional checkpoint/restart is projected to outweigh its benefits due to fault rates outpacing I/O bandwidths. As faults occur and propagate throughout hardware and software layers, pervasive notification and handling mechanisms are necessary. This report describes an initial investigation of fault types and programming interfaces to mitigate them.

Proof-of-concept APIs are presented for the frequent and important cases of memory errors and node failures, and a strategy proposed for filesystem failures. These involve changes to the operating system, runtime, I/O library, and application layers. While a single API for fault handling among hardware and OS and application system-wide remains elusive, the effort increased our understanding of both the mountainous challenges and the promising trailheads.

Contents

1 Introduction

1.1 System Scale and the Impact on Resilience

Current Systems and the Level of Reliability

The current generation of Petaflop scale supercomputers have about 20,000 processors sockets (for example, Cielo has just under 18,000 processor sockets) plus over 50 times as many memory chips and a corresponding number of network chips, VRMS, flash memory chips, etc. (Current Blue Gene systems have about five times more processor sockets to get to a Petaflop although each processor socket is considerably simpler and uses a lot less power. The total number of memory chips is similar since that is determined by the total amount of memory but the number of memory chips per socket is much less.) A Cielo node has two processor sockets with about 80 Gflops per socket or 160 Gflops per node. For Cielo the Job Mean Time To Interrupt (JMTTI) for any job running on the system is required to be at least 25 hours. This level of reliability has proven to be sufficient for the applications to make efficient use of the machine.

Current supercomputers utilize a wide variety of means to improve reliability. These include such things as ECC memory and data paths, redundant power supplies with automatic failover, redundant power chips in Voltage Regulation Modules (VRMs) with automatic failover, CRC checking and retransmission for detecting and correcting network errors, RAID with hot spare disks for disk storage, etc.

Most of the current applications get resilience primarily through defensive I/O. They write application checkpoint files to disk storage at predetermined points during the application run. When there is a system problem that results in a user job being interrupted, the application restarts from the last full, stored checkpoint file. This process looses all of the computation that was completed between the last completed checkpoint and the time of the interrupt. There is also some extra overhead associated with writing the checkpoint files, although, some application codes use the checkpoint files for graphical output as well as for resilience.

Exascale Systems

There have been a few projections for what the peak performance of an Exascale machine processor socket might be and they generally are in the 5 10 Teraflops range. If the 10 Teraflop value is assumed then an Exascale machine, one that would achieve an Exaflop on HPL, would have around 150,000 processor sockets. (For this discussion, it is assumed that the percentage of peak achieved on HPL is a little less than 70%.) This leads to about a factor of eight (8) in the number of parts in the machine provided that all other parts scale similarly. While this assumption is possible in the Exascale time frame it is not conservative since it requires that the performance of a single socket increase by over a factor of 100 in a period of about eight years. This assumption also implies that the amount of memory is increasing at about a factor of eight slower rate than the peak performance. These high performance processor sockets will be high power and complex. Other designs that use simpler, lower power processors might have another factor of ten or more in the number of processor sockets and total parts. Simpler and lower power parts are likely to be more reliable but it is not clear that the reliability would increase enough to make up for the increased number of parts.

As stated above, Cielo was required to have a JMTTI of 25 hours. If only part counts were involved and the part reliability did not improve this would lead to a JMTTI for the Exascale machine of only about 3.0 hours. When the overhead for defensive I/O and the time lost resulting from going back to previous checkpoints is factored in, it may be very hard for the application to make forward progress with this level of reliability. This situation is made even more difficult by the fact that the time required to perform checkpoints will increase because data storage system performance is increasing at a much slower rate than machine peak performance and system memory size.

1.2 Scientific Application Types

Loosely Coupled and Master/Slave Applications

These types of applications include Monte Carlo codes, ray-tracing codes, and other similar codes that involve one or more master nodes that assign work to other (worker) nodes. As the worker nodes complete their assigned work they feedback results to the master node/s, which accumulate the results. The master node/s continue to assign work to the worker nodes until all of the work is completed. This type of application can survive the loss of a worker node since each worker node is independent from the other worker nodes and only depends on the master node/s. If a worker node is lost its work can be assigned to another worker node.

This type of application tends to require very limited communication between nodes and can be fairly easily designed to survive the loss of a worker node and even a master node when there are multiple master nodes.

Continuum Applications

Continuum applications usually consist of a large computational mesh that is distributed across many nodes. The computations performed on each node, use information from the surrounding nodes as a boundary condition for their computations and provide information to the surrounding nodes for their computations. This process is usually iterative within a problem time step. As a result there may be several communication steps in each time step. Computational progress is dependent on keeping all of the nodes involved in the calculation working.

When a failure occurs on a node or a node fails that is part of a large computational mesh a part of the problem is lost and the boundary information for computations on neighboring nodes is also lost. Currently, the full application must be stopped and restarted from a previous checkpoint file. This application type is Sandias's and the other NNSA lab's current dominant type of application.

1.3 The Proposed Approach

Exascale systems will need to use all possible means of improving reliability to be successful. This includes improved hardware component reliability, more redundancy of components with automatic failover, improved error detection and correction, and improved system software reliability. However, the increase of a factor of ten or more in the number of parts over current Petaflop systems that will be needed to build Exascale systems in the 8-10 year time-frame may make it impossible to achieve Exascale system reliability that is sufficient for continuum exascale applications to make reasonable computational progress. As a result new approaches to achieving resilience may be needed.

The proposed approach is to build more fault tolerance into software, both system and application, through the use of a Resilience API. The Resilience API would make information about hardware and system software faults available to the system software and to application codes, and provide a mechanism to express responsibility for fault handling. The system software and application codes would then be able to make decisions on how to proceed. Currently, when a fault, that is not automatically corrected by the system is detected, the application is killed. Also, applications that reach a time-out for certain types of system services, for instance I/O, are killed. In some of these cases the application might be able to survive the fault or be able to adjust to a longer wait for the system services.

As an example, for applications that are master/slave in nature, a Resilience API could inform the application that a node has failed. (There could be a variety of reasons for the failure such as the processor died, or a 2-bit memory error occurred, etc.) The application could decide if it could and if it wanted to

proceed without the failed node based on the information provide to it through the Resilience API.

Implementation of a Resilience API depends on the identification and availability of the important system health information. This includes error data and other indicators of system health. It also depends on monitoring of the important system health information for both system hardware and software in near real-time and providing that system health information to the system software and application code in near real-time.

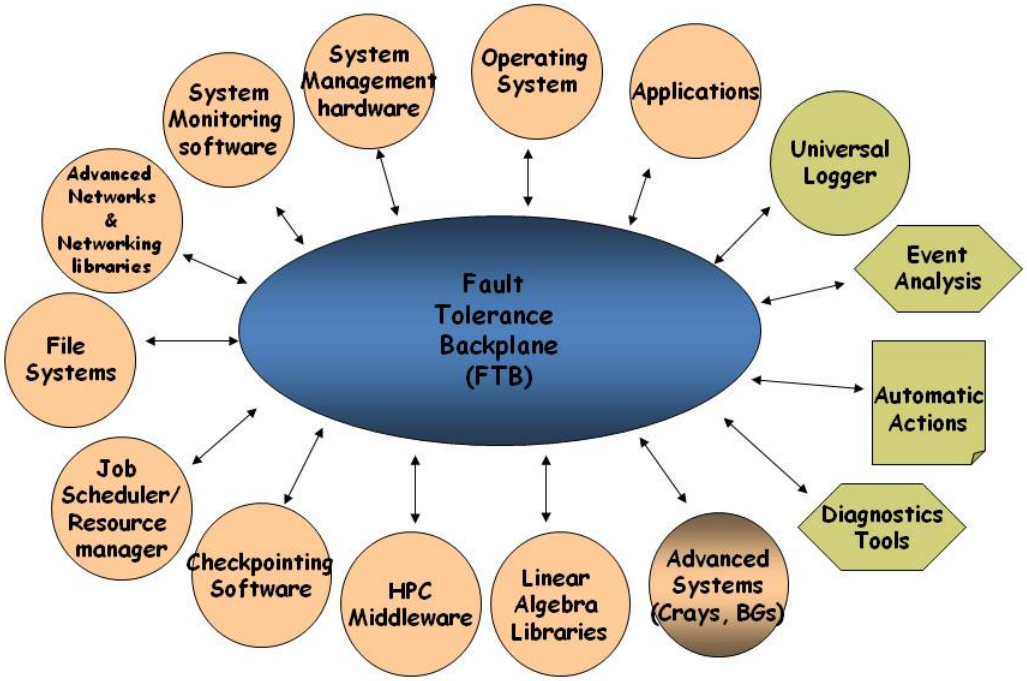

Figure 1. FTB diagram

2 Related Efforts

2.1 CIFTS/FTB

A Coordinated Infrastructure for Fault-Tolerant Systems (CIFTS) [18] is a collaborative effort by a number of agencies including; Argonne, Lawrence Berkeley and Oak Ridge National Laboratories and the Universities of Indiana, Ohio State and Tennessee. The effort was initiated to address a critical gap in current and next generation HPC platforms - fault responses are largely uncoordinated due to the lack of a standard mechanism to communicate fault information among all components throughout the system. In contrast to independent efforts to improve fault-resilience by individual layers or libraries [12, 31], CIFTS seeks to provide an interface to exchange fault-related information between software and thus facilitate coordinated responses to faults. CIFTS proposes and includes a design for a fault tolerance infrastructure specifically targeted at high-end computing systems. The key component of CIFTS is the Fault Tolerance Backplane (FTB), which is a publish/subscribe framework for communicating fault information. Software using the FTB API is said to be FTB-enabled. The type of information published to the FTB does not appear to be limited, but events are generally fault events which can range from hard errors to warnings with an associated severity level (fatal, warning or info).

Figure 1 illustrates a conceptual fully FTB-enabled framework. Not all system components would be required to be FTB-enabled, but those that are can act as a subscriber of information, a publisher, or both.

Conceptually, the framework presents a fabric whereby important information, typically hard to obtain, would be readily available at all levels of the software stack. The FTB framework is implemented using daemons that accomplish various FTB responsibilities. These daemons are distributed, in-situ, throughout the target platform. Communication with an FTB agent can be local or remote, meaning an FTB-enabled MPI application could obtain information from an agent executing locally on the node that it is running or obtain the information from a remote node. In the implementation described in [18] FTB daemons organize themselves in a tree structure to reduce communication overhead and can use message aggregation techniques to prevent adverse impact from message storms which would be common during failure event scenarios. Note it is possible for multiple FTB enabled components to detect the same failure in different ways and publish the event, this situation cannot be fully mitigated by message aggregation. The current implementation uses TCP/IP and typically employees the communication network used by the application.

Consider a typical HPC platform which includes IO via a parallel file-system component. An executing job using a version of MPI that is FTB-enabled would connect to the FTB and register for events that are generated from the file-system (also assumed to be FTB-enabled). CIFTS uses a namespace structure to partition classes of events (such as filesystem, MPI, etc), and components publish or subscribe to events of interest to them. The client has two choices of how to receive information: a polling or callback interface. The callback interface is likely the most used of the two options since it is asynchronous. The polling option is reportedly included for platforms that do not have the option of spawning a callback thread. Lets assume our example MPI job connects with the FTB and requests warning level events from the file-system namespace, such that it can avoid using that filesystem. Similarly, the job scheduler could notify for such events in order to not launch jobs which depend on that filesystem [18].

Conceptually, the CIFTS design seems reasonable. The following critiques regarding CIFTS mostly concern the current implementation of the FTB API (named FTB version 0.6). Since one of the goals of the team was to produce a generic solution that could be implemented on a range of platforms most of the following criticisms cannot be resolved, that is without the equivalent of a platform specific implementation. While small scale clusters would likely not be affected by yet another daemon being present on the active compute nodes, it has been shown that interruptions, especially those characteristic of system daemons can be highly detrimental to application performance at large scale [13] and [26]. Additionally, TCP/IP traffic traversing the high-speed communication network used by MPI applications will likely add an additional performance impact. The CIFTS team evaluated performance impact of their infrastructure in [18] and in some cases found significant impact which they were able to mitigate using various optimizations. While their experiments showed negligible overhead on up to 512 nodes, it is our opinion that larger scale studies with real scientific HPC applications are warranted. It is our opinion that while conceptually the CIFTS design is reasonable, using the resources meant for computation on a large scale platform is not practical for next generation systems. We will note, however, sometimes counterintuitive practices can be shown to be beneficial at extremely large scale. Process replication [14] which uses twice the amount of resource for a single application execution can be shown to be highly beneficial at large scale. It is possible that in an environment where faults are ubiquitous and applications have the ability to productively react to fault notifications the trade-off between the amount of *potential* overhead introduced by FTB and the benefit gained by fault tolerance (reduced checkpoint-restart time) could be shown to be a win overall. While possible, we feel this it not likely in this case if the FTB is implemented in-situ. It is our opinion that to be effective at large scale the FTB should be implemented on an out-of-band network.

At least CRAY and IBM capability systems have out-of-band Reliability Availability and Serviceability (RAS) networks. An implementation of the CIFTS design on the RAS sub-system that provides communication with compute nodes could very well deliver the benefits of CIFTS without the potential impacts to performance. A FTB that existed out-of-band on the RAS network separating the communication overhead from the computation network would eliminate application performance impact. The application would still need to respond to the fault information it registered for, which would likely be worthwhile, assuming a negligible false-positive rate. In the current implementation, there does not seem to be a way for an application to register for only those faults that affect resources upon which it depends. Instead, applications register for namespaces or fault types, and receive all such events from throughout the system. Applications do not have the intricate system knowledge necessary to determine the full set of components they depend on, so

receiving fault information from all components is hardly ideal. It is possible however that tight integration with a RAS system could address this. In addition, subscription to an event type does not express responsibility to handle events - lower layers would not know to alter their handling behavior unless they subscribed to subscription events, which is not provided for in the FTB API.

CIFTS is not currently in production use (according to the lead developer of CIFTS). While we have found at least one instance of a software product that is FTB enabled (OpenMPI version 1.5.3) adoption of FTB seems to suffer from a catch-22 situation. Subscribing software need to know what to subscribe for, and publishing software needs to know what subscribers are looking for - this is difficult given the distributed community of HPC software teams. To be affective, FTB must be adopted by many components.

Achieving wide-spread adoption of even the best concepts is often slow for innumerable reasons which often do not include the technical quality of the idea. Open literature suggests that CIFTS has been tested on both IBM Blue Gene and Cray XT platforms in addition to other commodity clusters. Additionally, CIFTS has been mentioned in some recent responses to government requests for information. Efforts like CIFTS are valuable for the community as a whole if for no other reason than to bring attention to the importance of fault tolerance and resilience for future platforms. While we feel that the current implementation is impractical even for current large scale platforms, valuable lessons learned could benefit future implementations.

FTB events are currently defined at compile-time, with run-time definitions planned for a future release. The set of currently defined FTB events are listed in Table 1, along with their severities and package support. The following three MPI distributions support FTB, listed from highest support to lowest: MVAPICH2 [2, 4, 3] from Ohio State University, MPICH2 [1] from Argonne National Laboratory, and OpenMPI [5]. From the events in Table 1, it appears that CIFTS has focussed so far on process health and checkpointing. Ohio State also has an effort to integrate the Intelligent Platform Management Interface (IPMI) with FTB, but very little information on this is available.

Berkeley Labs Checkpoint Restart (BLCR) is also FTB-enabled. MVAPICH2 depends on BLCR to checkpoint process states. By initiating a checkpoint via the MPI layer, it can flush in-transit messages and suspend further messages, ensuring a constent state. The BLCR events in table 1 regard individual processes within a job, whereas the MPI checkpoint events refer to the success or failure of the entire job's checkpoint. More information on the cooperation via FTB of BLCR, MVAPICH2, and Infiniband to checkpoint applications is available elsewhere [16].

2.2 MPI 3.0 Process Fault Tolerance Proposal

With the high probability of individual node failure in extreme-scale systems, the MPI forum is attempting to address support for failure and recovery at the MPI level [6]. This proposed API for inclusion in the MPI 3.0 specification provides applications with uniform semantics in the presence of fail-stop [17] process failures. The proposal requires the application to explicitly control when process failures are recognized as well as the actions taken upon those failures. The library provides mechanisms to notify when a process has failed, the ability to bring in processes to replace failed ones in an already running application, and mechanisms to isolate failures such that the failure of one process does not impact the other processes in an application.

This proposed API requires the MPI library or runtime software to provide what is termed an *eventually perfect failure detector* [8, 11, 15]. Eventually perfect failure detectors are a variation of consensus algorithms [21] from distributed systems. These detectors are defined as being strongly accurate, meaning no process is reported failed unless it is actually failed, and eventually strongly complete, in a finite amount of time every failed process is reported as such. Therefore, the application is guaranteed that once the library reports an MPI process as failed, that process will not return to the computation. If a process is incorrectly marked as failed by the library it must be excluded from the computation for the remainder of the application run by the system.

Event		Severity	MVAPICH2	MPICH2	OpenMPI	*BLCR*
Process status and migration						
FTB_MPI_PROCS_ABORTED		error	✓	✓	✓	
FTB_MPI_PROCS_COMM_ERROR		error	✓	✓	✓	
FTB_MPI_PROCS_DEAD		error	✓	✓	✓	
FTB_MPI_PROCS_RESTART_FAIL		error	✓	✓		
FTB_MPI_PROCS_MIGRATED		info	✓	✓		
FTB_MPI_PROCS_MIGRATE_FAILED		error	✓			
FTB_MPI_PROCS_RESTARTED		info	✓	✓		
FTB_MPI_PROCS_UNREACHABLE		error	✓			
PREDICTOR_NODES_FAILURE		info	✓			
REQ_MIGRATE		info	✓			
MIGRATE_DONE		info	✓			
Checkpoint-Restart						
CR_FTB_APP_CKPT_REQ		info	✓			
CR_FTB_CHECKPOINT	*CHKPT_BEGIN*	info	✓			✓
FTB_MPI_PROCS_CKPTED	*CHKPT_END*	info	✓	✓	✓	✓
FTB_MPI_PROCS_CKPT_FAIL	*CHKPT_ERROR*	error	✓	✓	✓	✓
CR_FTB_APP_CKPT_FINALIZE		info	✓			
RESTRT_BEGIN		info	✓			✓
RESTRT_END		info	✓			✓
RESTRT_ERROR		error	✓			✓
FTB Support over Infiniband						
FTB_IB_EVENT_PORT_ERR		error	✓			
FTB_IB_EVENT_DEVICE_FATAL		error	✓			
FTB_IB_ADAPTER_UNAVAILABLE		warning	✓			
FTB_IB_ADAPTER_AVAILABLE		info	✓			
FTB_IB_ADAPTER_INFO		info	✓			
FTB_IB_PORT_INFO		info	✓			
FTB_IB_EVENT_PORT_ACTIVE		info	✓			
FTB_IB_EVENT_LID_CHANGE		info	✓			
FTB_IB_EVENT_PKEY_CHANGE		info	✓			
FTB_IB_EVENT_SM_CHANGE		info	✓			
FTB_IB_EVENT_CLIENT_REREGISTER		info	✓			

Table 1. FTB events, severity, and support. BLCR uses *CHKPT* names, but the events are symantically equivalent to the FTB-standard names they appear with.

This proposal associates with each MPI process the concept of a *process state*. This state can take one of three values, MPI_RANK_STATE_OK, MPI_RANK_STATE_FAILED, and MPI_RANK_STATE_NULL. MPI_RANK_STATE_OK is the normal running state, with no failures recognized. MPI_RANK_STATE_FAILED is a state in which the failure detector has recognized the process as failed, but the application has not received the notification. Lastly, MPI_RANK_STATE_NULL is the state in which the process is failed and that failure has been recognized by the application. Applications transition from MPI_RANK_STATE_FAILED to MPI_RANK_STATE_NULL when a local process recognizes the failed state using an appropriate validation function defined in the next paragraph. Note that once a process moves into the states MPI_RANK_STATE_FAILED or MPI_RANK_STATE_NULL it can not transition back to MPI_RANK_STATE_OK. In addition, processes in the MPI_RANK_STATE_NULL state can not transition back to MPI_RANK_STATE_FAILED.

Notification of a process failure to the application is done through the validation API. This API updates, accesses, and modifies the state of a process in a given process group. Logically, the MPI library maintains two process state lists for each group and communicator, one that is local to the calling process and one that is global. The local list is maintained by a process and is only locally consistent. The global list, on the other hand, is updated collectively and accessed locally. This global list is guaranteed to be consistent within the associated group or communicator. The validation API provides both a locally consistent snapshot that is ensured to return immediately but is not necessarily globally consistent and a globally consistent collective snapshot that will eventually return with either success or some error to each surviving process. Therefore, this globally consistent interface is guaranteed to not hang indefinitely in the presence of process failure.

As stated earlier, an application must opt-in to the semantics described in this fault tolerance proposal. To opt-in the application must replace the default error handler (i.e. MPI_ERRORS_ARE_FATAL) with a different error handler (e.g. MPI_ERRORS_RETURN). If an MPI process attempts to communicate with a failed MPI process the outcome is dependent on whether it is a point-to-point or collective operation.

In point-to-point operations, if a process attempts to communicate with an unrecognized failed process (one which is in state MPI_RANK_STATE_FAILED), it will return a MPI_ERR_RANK_FAIL_STOP until the failed process is recognized by using one of the validation functions. Communicating with an already recognized failed process, however, has the same semantics as communicating with MPI_PROC_NULL [30]. Therefore, this communication operation will return MPI_SUCCESS without modifying buffers. If the communication operation is a receive with an MPI_ANY_SOURCE as the source, the receive operation will return MPI_ERR_RANK_FAIL_STOP if any process in the communicator fails. If the point-to-point operation is non-blocking, errors will not be delivered until the corresponding completion function (e.g. MPI_Wait() or MPI_Test()) is called.

For collective operations, any unrecognized failures within the associated communicator disables all collective operations on that communicator. Similar to point-to-point operations, collective operations on a disabled communicator will return MPI_ERR_RANK_FAIL_STOP. In order for operations to be re-enabled, all failed processes must be globally recognized using the validation interface.

While the goal of this API is to provide a uniform resilient interface between the MPI library and the application, it does have a number of limitations. The most important limitation of this proposed API, is its dependence on a eventually perfect failure detector. Construction of these detectors is straight-forward assuming a fail-stop model for failure of MPI processes. Using more realistic failure models however, for example Byzantine and crash failure [17, 9, 27], construction of these detectors becomes challenging due to the difficulty in determining the difference between a failed or misbehaving process. Also, the distributed consensus protocols required for these detectors are generally quite expensive performance-wise and untested for HPC workloads at this predicted scale. In addition, this API puts onus on the application to recover state lost to failure, it does not perform this recovery for the application. Therefore, each application will be required to implement its own recovery procedure specific to that application in order to take advantage of the functionality. Lastly, this API only covers MPI applications and failure/recovery at the MPI level. Failures in other levels of the software stack are not covered. In addition, coordination not specified in this API will likely be needed to propagate failure notices through various levels of the software stack.

3 Exascale System Potential Failures

3.1 General Approach for Evaluating Failures

An Exascale supercomputer will have hundreds of types of parts and millions of total parts. As described in section 1.1, even with very high levels of individual part reliability there will still be several failures during every day of operation. Successful system operation will depend on how well the failures can be managed and tolerated.

Table 2 provides an extensive list of different kinds of failures that have occurred in past and current supercomputers. An Exascale supercomputer system is likely to have similar failures because it will be built from a similar set of components. The main difference is that an Exascale system will have many more components. In Table 2 the set of potential failures are examined from the perspective of their impact on system reliability, on their estimated frequency of occurrence, on their estimated potential for being resolved through improvements to system hardware, on their estimated potential for being resolved through system software improvements, and on their estimated potential for being resolved through an Application Resilience API.

Table 2: Failure Types (Hardware, except final section)

	Importance to System Reliability	**Frequency of Occurrence**	**Potential to Mitigate by Hardware**	**Potential to Mitigate by System Software**	**Potential to Mitigate by Application**
System Failures					
Power Loss	High	Low	High	NA	NA
Clean Power	High	Possibly High	High	NA	NA
Cooling Loss	High	Low	High	NA	NA
File System Failure	High	Medium	Low	High	High
System Management Failure	High	Low	Low	High	NA
System Network Failures	High	High	High (Firmware)	Medium	NA
Node Failures: On-Processor					
Logic (Silent)	High	Unknown	Low	NA	NA
Memory	High	High	High (ECC)	Low	Low
Data Paths	High	High	High (ECC)	Low	Low
Off-Processor					
DRAM	High	High	High (ECC)	Low	Low
Flash	High	High	High	NA	Low
Data Paths	High	High	High (ECC)	NA	Low
Node Board					
PCB Boards	High	Low	High	NA	Low
Data Paths	High	High	High	NA	Low
RAS Components	High	Medium	High	Medium	Low
System Management Components	High	Medium	High	Low	Low
Flash Memory	High	Medium	High	Low	Low

Continued on next page

Table 2 – continued from previous page

	Importance to System Reliability	Frequency of Occurrence	Potential to Mitigate by Hardware	Potential to Mitigate by System Software	Potential to Mitigate by Application
Power - VRMs	High	High	High (Redundant)	NA	Low
Chassis and Cabinet Failures					
System Management Components	High	Low	High	NA	NA
System RAS Components	High	Low	High	NA	NA
Power Supplies	High	High	High (Redundant)	NA	NA
Management Network Components	High	Low	High (Redundant)	Low	NA
Primary Communication Network Failures					
NIC and Router Chips	High	High	Medium (Routing)	Medium	Low
Boards	High	Low	Medium		Low
Cables					Low
System Management Components	High	Low	Medium	Medium	NA
RAS Components	High	Low	Medium	Medium	NA
Transmission Errors	High	High	High (Firmware)	Medium (End to End)	Low
File System Failures					
Disk Failures	High	High	High (Hot spares)	High	High
Disk Controller Failures	High	High	High (Redundant)	High	High
Power	High	Low	High	NA	Low
Silent Errors	High	Low	Medium	NA	NA
System Software					
Operating System Failures on Service Nodes	High	Medium	Partial (Hot spare nodes with failover)	High (Software support for failover)	NA
Operating Systems Failures on Compute Nodes	High	High (Complex)	NA	High (OS Failure Management)	Low
Run-Time System Failures	High	High	NA	High (Control Resource Contention)	Low
I/O System	High	High	NA	High (Contention and Failover)	High

Continued on next page

Table 2 – continued from previous page

	Importance to System Reliability	**Frequency of Occurrence**	**Potential to Mitigate by Hardware**	**Potential to Mitigate by System Software**	**Potential to Mitigate by Application**
Communication Network	High	High	NA	High (Software Retransmission)	Medium
System Management	High	Low	NA	High (Minimize Dependencies)	NA
RAS System	High	Low	NA	High (Minimize Dependencies)	NA

3.2 Detailed Look at Failure Types

The first column in Table 2 is a comprehensive list of failure types. The failure types have been divided into two major categories, hardware and system software. Within the hardware category the failure types are divided into five subcategories. The first hardware subcategory is failures that can bring down all or a major part of the system, for example loss of system power will cause the full system to crash. The other subcategories involve failures that would normally not cause a full system interrupt. The second subcategory is node failures. Node failures are grouped into processor, off processor memory, and node board and component failures. The third subcategory is chassis and cabinet failures. The fourth subcategory is primary communication network failures. And the fifth subcategory is file system failures.

In the second column failures are categorized as to their importance. All of the failure types listed are considered to be of high importance because they can potentially cause an application code to be interrupted or the system to suffer an interrupt. There are other types of failures that occur but in current systems they are currently mitigated through redundant parts with automatic failover or hardware error correction. The hardware involved in these other failures is usually repaired during maintenance periods. (An example of a failure that would not normally cause an application or system interrupt is the loss of a power transistor in a VRM that has N+1 power transistors where N is the number of power transistors needed.) Column three of Table 2 is an estimate of the frequency of failure that might be expected in an Exascale system. Frequencies are rated as high, medium, and low with one exception, processor silent logic errors, which are rated as unknown. The frequency is likely to strongly correlate with the number of parts and the complexity of the parts. The frequency estimates are based on past experience for systems at Sandia and information from private discussions about the behavior of supercomputers at other sites.

Processor logic errors are not generally detected by hardware or software means in current systems and as a result the frequency of this type of error is unknown. This type of failure is very difficult to detect and, therefore, it probably will not be detected in Exascale systems either. (An exception is that messages sent with end-to-end message CRCs can detect logic errors that occur during message transmission.) For most processor logic errors, detecting the error requires a duplicate logic operation to compare the result to. (Even if the logic error were detected it would still be difficult to correct since it would require at least three calculations to have a vote to determine the correct result.)

The fourth column in Table 2 is an estimate of the potential for mitigating failure types through improvements to the hardware used to build an Exascale system. The potential is evaluated as high, medium, low, or Not Applicable (NA) for each failure type. For many hardware failure types, improvements to hardware

reliability through error detection and correction or through redundant parts with automatic failover are the best approach. In some cases they are the only practical approach. For example, loss of all cooling to the system can only be handled through improvements to cooling system hardware, either more reliable hardware or more capacity. However, there are a few hardware failure types, system management workstation failure for example, where hardware redundancy combined with sophisticated system software failover may be the best approach.

The fifth column is an estimate of the potential for mitigating failures through system software improvements. Again the potential for failure mitigation is estimated to be high, medium, low, or NA for each type of failure. System software improvements are the best way to resolve failures in system software, although, applications can be modified to tolerate some types of system software failure. System software improvements can be used to mitigate some hardware failures such as the loss of a management or service node through automatic failover capabilities that allow both the system and application to continue.

The last column is an estimate of the potential for failures to be mitigated through an Application Resilience API. Only a few failure types were identified as being suitable for an Application Resilience API. Most of these have to do with file systems or are only suitable for applications (section 1.2.1) that can continue with the loss of a compute node. A major reason why the potential is limited is that for many failure types the application can only find out about the failure after the computation has progressed to far along to recover except by restarting from an application wide known state, an application level checkpoint.

File system failures are a little different than most other types of failures because for most applications large scale I/O is highly synchronized within the application. In effect, the application is in a quiescent state when it is doing large scale I/O. As an example, in an Exascale system that has multiple parallel file systems, an application that suffers a file system error could fairly easily choose to switch file systems and start the checkpoint over using the new file system. Even for a file system that is not responding or responding slowly because of error recovery or contention, the application could choose to delay writing a checkpoint and continue with the computation until the file system becomes more responsive. Currently, for most systems, the application would be killed because the file system was unresponsive.

Supercomputers themselves are tightly coupled systems. The effects of faults may be observable at the faulting component itself, on directly connected components, or indirect/remote components. This is true for both hardware and software. In general, as one gets farther away from where a fault occurs, the more difficult it is to effectively monitor its occurrence and mitigate its effects. Most true faults originate in hardware - we do not consider software bugs to be faults per se. If hardware is unable to mitigate, the system software is the next best option. And only in a few cases can applications meaningfully respond to hardware faults.

4 Memory

4.1 Hardware DRAM Failures

DRAM memory modules are one of the most plentiful hardware items in modern HPC systems. Each node may have dozens of DRAM chips, and large systems may have tens or hundreds of thousands of DRAM modules. The combination of the quantity and the density of the information they store makes them particularly susceptible to faults, particularly at initial machine bring-up before bad DRAM modules have been burned in and replaced.

Because of this, essentially all HPC systems include some built-in hardware fault tolerance for DRAM. The most common hardware memory resilience scheme has the CPU memory controller write additional checksum bits on each block of data written (128-bit blocks are used on modern AMD processors, for example). The controller uses these bits to detect and correct errors reading these blocks of data back into the CPU.

Most modern codes use Single-symbol[1] Error Correction and Double-symbol Error Detection (SEC-DED) schemes, allowing them to recover from the simplest memory failures and at least detect more complex (and less frequent) ones. If the width of a symbol is greater than or equal to the width of a DRAM chip, such systems are referred to as *chipkill* ECC, as they can tolerate the loss of a complete DRAM chip.

In addition, modern hardware includes memory scrubbers that continually walk memory at a configurable rate reading each memory line and attempting to scrub out correctable errors before faulty memory lines incur additional errors that may not be correctable. For applications that frequently walk all of memory, such scrubbing is likely redundant. In addition, scrubbing may have a minor performance impact that has not, to our knowledge, been completely quantified.

Overall, recent research has shown that uncorrectable errors (e.g., double-symbol errors) are increasingly common in systems with SEC-DED memory protection [22, 29], with one study showing uncorrectable DRAM errors occurring in up to 8% of DIMMs per year in non-chipkill systems. Such errors result in a machine check exception being delivered to the operating system, which then handles these errors based on the OS's particular mechanisms and policies.

4.2 Current OS-level DRAM Error Recovery

DRAM error handling in current operating systems such as Linux and Catamount focus on handling DRAM machine check exceptions as necessary. Both Linux and Catamount log all corrected and uncorrectable DRAM errors for later hardware-level remediation (e.g. replacing faulty hardware DRAM modules). By default, both operating systems also kill the running application if an uncorrectable fault is in application space and potentially reboot the system if the fault is in OS memory.

Recent versions of Linux include somewhat more advanced recovery mechanisms [20]. Linux can now transparently handle a limited set of uncorrectable DRAM failures that occur in operating system memory and can propagate some failures that occur in application memory to the application for handling. At the OS level, Linux can now recover from DRAM faults in file system cache pages that are not dirty by discarding the pages containing failed addresses and marking the containing hardware frame as unusable; faults in other OS memory locations still result in error logging and system reboot.

At the application level, Linux now allows applications to use the `sysctrl` interface to request notification of failures in application-level pages via a `SIGBUS` signal. In this case, Linux replaces the user-level page containing the failed memory address with a new zero-filled page and signals the application of this action. This recovery action is only taken for scrubber-signalled errors; errors resulting from attempting to consume

[1] A symbol in modern DRAM systems typically comprises 4 or 8 bits of data.

```
/* Register callback for handling failure in specific allocation of
 * failable memory at a specified byte offset and length. arg is an
 * opaque user-supplied argument. */
typedef void (*memfail_callback_t)( void *allocation, size_t off,
                                    size_t len, void *arg);
void memfail_recover_init( memfail_callback_t cb, void *arg );

/* Allocate resp. free failable memory */
void * malloc_failable( size_t len );
void free_failable( void *addr );
```

Figure 2. Application / Library interface to handle DRAM memory failures

failed data currently result in the application being killed unconditionally even if it has stated that it can recover memory failures.

Overall, the current Linux interface is relatively coarse, requiring the application to discard an entire page of memory instead of just the failed memory line, and only allowing the application to recover from scrubber-signalled errors. It does not appear that this coarse level of recovery is required by the hardware, however. In particular, it appears that the hardware should support recovering only the failed memory line assuming that the error that occurred is soft and can be corrected by re-writing the failed line. In addition, it should be possible to recover attempts to consume failed pages instead of just scrubber errors.

4.3 Application-level DRAM Recovery

Outside of recent research conducted at Sandia and UNM described below, almost no Linux applications appear to use Linux's current coarse-grained DRAM recovery interface. In particular, the only Linux code we have found that uses this interface is the KVM virtual machine monitor, which simply propagates errors in application memory to running virtual machines. We believe that the low level of the application-level interface (asyncronous signals), the amount of data lost in the case of failure (a full page), and the general difficulty of recovering from such errors all contribute to the lack of applications attempting such recovery.

We have, however, recently conducted research on attempting to recover from DRAM failures in HPC codes, particularly in a sparse iterative linear solver in the Trilinos library[19]. In this work, described fully elsewhere [7], an application-level library interfaces with the signal-based OS notification of failures to provide a useable application-level interface to memory failures.

This application interface, shown in Figure 2, focuses on run-time memory allocation. In particular, the interface provides the application with separate calls for allocating *failable memory*—memory in which failures will cause notifications to be sent to the application. These calls work like `malloc()` and `free()`. In addition, the application also registers a callback with the library. The callback is called once for every active allocation when the library is notified by the OS of a detected but uncorrected memory fault in that allocation.

This research has shown that this interface is sufficient to allow certain numerical algorithms to recover from memory failures. In particular, the Trilinos team used this interface to demonstrate algorithmic innovations that allow a sparse linear solver to recover from these failures, even when the amount of data lost comprises an entire 4KB memory page. In the case of Trilinos, lost data was later recovered from redundant copies of the data kept by the library; more general checkpointing schemes (for example to local SDRAM-based storage) could also be used to recover from DRAM failures if the data being recovered was generally

read-only. Interested readers are referred to the paper cited above for additional details.

4.4 Discussion

While we have demonstrated a user-level API that allows some key HPC algorithms to recover from memory failures, current state-of-the-practice approaches for DRAM are not appropriate for many HPC applications. In particular, the current OS-level interface to memory failures are coarse and low-level, and the approach we have demonstrated for recovering from errors requires non-trivial numerical algorithm support and that the data being recovered is generally used in a read-only manner. This is insufficient for many explicit, non-iterative codes that have large amounts of data that is frequently written.

Recovering from DRAM failures in such codes appears to be particularly challenging, and we are currently unaware of any approach that will enable recovery from memory failures in such codes. Without key algorithm-level innovation for such codes, we believe that hardware for running such codes will require robust hardware-level support for detecting and recovering from DRAM memory failures

Despite these limitations, however, we do believe that the APIs discussed here are appropriate for inclusion in HPC runtimes and applications as a general DRAM recovery API for HPC applications and libraries that can support them. Production-level implementation of such APIs would need to address key limitations of current system-level recovery APIs, however. In particular, while the signal-based mechanism used by Linux is sufficient as an OS/runtime level interface, a usable implementation must allow the application or runtime to recover at the level of an individual memory line when possible as opposed to mandating that the application always discard a complete page of data. In addition, a usable implementation must allow the application to recover from attempts to consume failed data as well as scrubber-signalled errors.

5 I/O

5.1 Cooperative Checkpointing

The most common fault mitigation strategy in use today is for applications to write checkpoints at periodic intervals, such that if they are prematurely interrupted, they can restart from the most recent checkpoint. Fixed intervals are optimal if the time between interrupts is exponentially distributed [10]. However, in 2006 Schroeder and Gibson showed that "the time between failure is not modeled well by an exponential distribution" - at least for a variety of clusters at LANL [28]. The question then arises - what is the optimal pattern of checkpoint writing if job interrupts are not exponentially distributed?

Oliner introduced "cooperative checkpointing" (also in 2006, [24]), in which system software "uses global knowledge of the state and health of the machine to improve performance and reliability by dynamically deciding when to skip checkpoint requests made by applications." In cooperative checkpointing, applications request checkpoint writes at any pattern (e.g. periodic such that no code changes are necessary), and the system decides to either grant or deny based on awareness of current operating conditions, or knowledge of the system's interrupt distribution. If the distribution is indeed exponential, cooperative checkpointing can easily handle this. Furthermore, it may enable optimal intervals system-wide, versus the current situation of each job determining its own checkpoint interval - which may or may not be optimal from a system-wide throughput perspective. However, if interrupts are not exponentially distributed, cooperative checkpointing may be able to tune writes such that they occur in a manner closer to the optimal.

In section 5 of [24], Oliner provides "a case analysis demonstrating that, under realistic conditions, an application using cooperative checkpointing can make progress four times faster than one using periodic checkpointing." A competitive analysis is performed, where application progress via periodic and cooperative checkpointing is calculated and compared. The stated goal of the analysis is to prove that "periodic checkpointing can perform arbitrarily badly compared to cooperative checkpointing" - not realism. "Realistic" refers to an estimated mean time to interrupt of 25 minutes for a 4,096 node BG/L system [23], but ends there. The conditions used in the analysis are as follows:

1. The job interrupt distribution is a weighted sum of two Dirac delta functions, the first having a weight of 0.9988 and period of 872 seconds (14.5 minutes), and the second weighted at 0.0012 with a period of 504,000 seconds (5.83 days). This means that 99.88% of the time, an application is interrupted every 14.5 minutes, and 0.12% of the time it runs uninterrupted for 5.83 days. This results in an overall mean time to interrupt of 1,459 seconds (25 minutes). The authors of the current report consider this distribution to be entirely non-realistic, and the case to be an excellent example of the inadequacy of a single MTTI summary statistic to elucidate the reliability behavior of a system.
2. Periodic checkpoints are initiated every 872 seconds (the optimal interval, assuming exponentially distributed interrupts) - such that all work is lost 99.88% of the time.
3. The cooperative checkpointing algorithm checkpoints as late as possible before every interrupt (via full foreknowledge of interrupt occurrences), and never otherwise. This strategy is referred to as "offline optimal".

Under these conditions, cooperative checkpointing enables an application to make progress four times faster than periodic checkpointing. This is a worthwhile proof, but the practical relevance of "4X" is low, given the assumptions used.

In other work [25], Oliner evaluates the robustness of multiple cooperative checkpointing strategies under a variety of failure distributions. The purpose of the work was "not to argue the quality of one failure model over another, nor one cooperative checkpointing algorithm over another," but rather to show that "periodic checkpointing lacks the flexibility to handle even a small variety of non-exponential traces or scale with increasing failure rates." Distributions studied are exponential, a sum of Weibulls (to approximate a

bathtub curve), uniform, and a small trace of a partial BG/L system (having 124 failure events total). The strategies evaluated are periodic (checkpoint every S seconds), back-off (checkpoint in S seconds, then 2S seconds, then 4S seconds, then 8S seconds, etc), and "risk-based". In the latter, the amount of work being risked by skipping a checkpoint is compared with the probability of failure, using knowledge of the failure distribution. Due to this knowledge, it is not surprising that it outperforms the others.

These works demonstrate that a checkpointing strategy which matches the interrupt distribution outperforms a strategy that does not, and that cooperative checkpointing could provide a tuning mechanism which does not require application modifications. Yet, neither Oliner nor the authors of this report are aware of any implementation of cooperative checkpointing on a production system. In addition, HPC job interrupt distributions remain an open research topic. The door of opportunity regarding cooperative checkpointing is open, but its potential gains for real systems are uncertain (and 4X may overstate the practical benefits).

5.2 CTH and Libsysio

The CTH code is of key value to Sandia, and consumes a significant portion of nodehours on past and present systems. It is therefore reasonable to evaluate opportunities for a resilience API with CTH, in the area of checkpointing. The information in this section was gathered from discussions with Courtenay Vaughan, Dave Crawford, and Lee Ward.

Three types of CTH input/output are relevant: *plot* files, *restart* files, and *backup restart* files. All three types are written at user-specified intervals - *plot* and *restart* in terms of simulated time (e.g. every millisecond simulated) or cycle number, and *backup restart* in terms of wall clock time (e.g. every hour spent simulating). Analysts use *plot* files to record simulation variables of interest, which can then be used for visualization and correctness-checking. For example, if key variables exceed allowable design tolerances the model may be revised and re-simulated. *Plot* files are appended throughout the simulation, with the size of each append being very small. *Plot* writes are typically the most frequent, followed by *restart*, followed by *backup restart*. *Restart* writes are significantly larger than *plot* writes, as they contain all information necessary to restart the simulation. By default these are written only at the start and end of a simulation, but analysts typically use them liberally throughout a simulation because they are quick and useful. Useful because they enable re-simulation of intervals, in case additional variables or finer-resolution plots, or other adjustments are desired. Quick because they typically finish in 30-60 seconds. CTH uses on-the-fly compression, attains roughly a 10:1 compression ratio, and each core saves about 3/4 of available memory to a *restart* file - perhaps a few hundred megabytes (per node) are written. Like *plot*s, *restart*s are appended. In contrast, *backup restart* files are written to new files. By default, these are written to A and B filenames in round-robin fashion, such that only the latest *backup restart* is available. The round-robin sequence length can be adjusted (such as A,B,C,...Z), but the default A-B is typical. For more information see the CTH User's Manual, CTH CONTROL INPUT section, option RDU*MPF.

The authors' impression is that current systems work well enough that CTH analysts use checkpoints (*restart*s and *backup restart*s) more for quality control than fault mitigation - their utility far outweighs their cost. As described previously in this report however, future systems may have slower checkpoints and more frequent faults. Thus, the following ideas are aimed at future rather than current systems.

Although cooperative checkpointing advocates that the system determine whether to skip checkpoints or not, for a first investigation the authors of this report wondered how difficult it would be for CTH to skip checkpoints if the I/O subsystem were degraded or broken. Perhaps additional simulation parameters could be used to express how many consecutive checkpoints are allowed to be skipped. This would enable for example, an analyst to complete a simulation sooner, even if it is missing some *restart* files part way through (which could be filled in later if needed). Fortunately, all CTH checkpoints are initiated using a single subroutine (DBGDS) which calls another one to write the data (DBMWTF), so revision of these two subroutines would affect all checkpoints in CTH. Such an approach would not remove any control from the user, while perhaps reaping some of the benefits of cooperative checkpointing.

Currently, a failed CTH checkpoint attempt results in an aborted simulation. This could be relaxed, and logic added to CTH either to skip the checkpoint and proceed with simulation, or abort. Ideally, all nodes would receive consistent health indicators from the I/O layer such as libsysio. Lee Ward has indicated however that "libsysio is not, itself, cooperative. Underlying file system client implementations may generate and maintain cooperation and associated state, and libsysio does provide methods to enable this, but it does not enforce or mandate it. Thus, the only safe assumption from the application point of view is that different nodes may receive different return codes to the same function, even at the same time." Therefore, CTH would need to coordinate node's decisions on whether to skip a checkpoint or not, as it would be a waste of time for some nodes to checkpoint and others not to. Since CTH is already in a barrier at time of checkpoint, coordinating the decision should not be problematic.

The above assumes that CTH is not killed by an underlying layer, which can also occur. Lee has also indicated that a function could be fairly easily added to libsysio, which would return a health indicator regarding the I/O subsystem. For example, if the underlying RAID was performing a rebuild such that read/write performance would be greatly degraded, libsysio could pass this information to an application via this query function. A coordination example would be for each node to poll for filesystem health, return codes collected and inspected for consistency, and decision to write or skip the checkpoint scattered back to the nodes. Similar return codes could also be added to functions such as `open()` or `write()`. Functions which abort the application could be modified to relax this behavior under appropriate conditions, and allow the application to decide how to respond to the operational environment.

There are often multiple filesystems available, which may be in different operational states - one may be broken, one degraded, and another working fine. CTH runs currently depend on the single filesystem given via input parameters. However, if an ordered list of checkpoint paths were input, these could be attempted serially as described above. This would give the user control over checkpoint paths, CTH responsibility to decide among them based on libsysio responses, and minimize the interaction between CTH and libsysio. Another option would be for libsysio to offer an alternative in response to queries for a filesystem which happened to be broken or degraded at the time. This would enable more dynamic handling of conditions (e.g., administrators could add filesystems as needed, even during application runs), but would make the interaction between CTH and libsysio slightly more complicated, and risk user confusion regarding locations of files. With either approach, CTH and/or utilities would need to deal with the reassembly of data spread across filesystems.

6 Prototyping a Case Study - Catamount

6.1 Single Node Failure Case

Building a prototype implementation besides demonstrating that an API can be used, enables us to better understand the issues involved. Catamount was picked for the prototype because the operating system source code is readily available and quite familiar to us.

The first case considered on Catamount was to allow the application to decide what should happen following a node fault. The current action is for the application to be immediately terminated with no option available. The resilience response adds the option of allowing the application the choice to continue and have the other nodes be notified of the failure. The case is quite different from the one in the memory section of this report. In the memory case the resilience response all occurs on the node experiencing the failure. Here the resilience response does nothing with the failing node, but makes the failure known to all other nodes.

This only requires from the API a way to notify the operating system that the application wants to handle node faults and a way for the operating system to provide the information that a fault occurred on a particular node to the application. This is accomplished by the application registering with the operating system. In the registration process the application provides the address of a block of application memory for the operating system to return information in. The application also notifies the operating system that it wants to process a particular signal if it occurs.

When the RAS system detects a failure and notifies YOD[2], the added code now passes that information on to the compute node operating system of all the other nodes in the job, rather than the former action of passing a fatal kill signal. The information delivered is simply a code indicating that the failure was a node fault and the physical number of the failing node. If the application has registered, it is alerted to the situation by a signal. If the application has not registered with the API, the signal simply terminates the job as before.

6.2 Issues and Problems with the Catamount Proof-of-concept Implementation

On Catamount, the RAS system provides the physical node number of the failing node. The application probably needs to know the ranks on that node. It has not been decided whether an API should convert the physical node to a group of ranks or whether that should be left to the application or the API library, which on Catamount can to do that conversion just as easily in user space.

An issue not unique to Catamount is how the application responds to inter-node communication that involves the dead node. Collectives seem particularly troublesome. The Cray Catamount RAS system, in the author's opinion, is unduly patient with a non-responsive node. It takes more than 25 seconds for RAS to declare a dead node "down". This slowness increases the probability of having communication in progress that cannot be completed.

A Catamount MPI job cannot cleanly do a normal exit with a node missing. Catamount MPI does a global sync at MPI_Finish time.[3]

While the following list is probably not complete, these items are identified as missing from the proof-of-concept implementation:

- Ability to deal with multiple node failures. Perhaps it is sufficient to abort on multiple simultaneous

[2] Yod is Catamount program that loads and oversees a particular parallel job.

[3] To prevent job hangs on application errors, if a node exits via Yod before the sync has occurred, Yod terminates the job. The death reported by RAS does not follow this path. A global sync will hang if a node fails to participate. (This sync is a Catamount operation, not an MPI operation.)

failures. Should be able to deal with subsequent failure after processing one.

- Notification by signal should be optional. Trolling at a later time should be permissible.
- Proof-of-concept implementation assumes registration is for node fault. API should be structured to accommodate other events.

7 Conclusions and Recommendations

Our consideration of a wide range of failure modes on past, present, and future systems yielded only a small number of cases for which we felt additional interaction with the application would yield significant benefits. Most failure modes are sufficiently catastrophic from the application's perspective that it could do little more than restart from a previous checkpoint, which does not require a new API. The one appliation we did specifically investigate (CTH) does not appear to have a significant felt need for additional resilience mechanisms. Future efforts should include the ranking of resilience concerns from a variety of application teams, and API features that would address them. While this project did not strike the mother lode of resilience solutions for exascale systems, it was a worthwhile effort and we offer the following conclusions:

1. We were unable to identify a general API that would be useful to a broad set of scientific and engineering applications. The exception is for applications that are embarrassingly parallel which can fairly easily survive almost all forms of node failures. These applications could use an interface that allowed them to make the decision rather than the system doing it for them.

2. File System I/O is a fault area that seems to have potential because large scale I/O is generally done when the application is quiesced and because it already has a well defined software interface that would require minor changes to implement for most applications. For Exa-scale systems I/O will probably become more significant because the bandwidth performance is not scaling even close to the system performance.

3. Additional fault notification and handling is needed in HPC systems, and can effectively mitigate a number of fault types. The memory, filesystem, and node failure cases we investigated involved changes to the application, I/O, runtime, and OS layers, for which we developed case-specific APIs.

4. Most faults need to be handled by either improvements in hardware error checking and correction and/or through system software improvements in fault tolerance that are hidden from the application. An LDRD starting FY12 is investigating these fault-tolerant system software solutions for exascale-class systems.

References

[1] MPICH2 FTB Events. `http://wiki.mcs.anl.gov/mpich2/index.php/MPICH2_FTB_events`.

[2] Mvapich2 FTB events for checkpoint/restart. `http://nowlab.cse.ohio-state.edu/projects/ftb-ib/software/FTB-CR-MVAPICH2.txt`.

[3] Mvapich2 FTB Events for infiniband. http://nowlab.cse.ohio-state.edu/projects/ftb-ib/software/FTB-IB-Events-1.0.txt.

[4] Mvapich2 FTB Events for process migration. http://nowlab.cse.ohio-state.edu/projects/ftb-ib/software/FTB-MVAPICH2-Migration.txt.

[5] OpenMPI FTB Events. `http://osl.iu.edu/research/ft/cifts/api.php`.

[6] Run-though stabilization interfaces and semantics, July 2011. `http://svn.mpi-forum.org/trac2/mpi-forum-web/wiki/ft/run_through_stabilization`.

[7] Patrick G. Bridges, Mark Hoemmen, Kurt B. Ferreira, Michael A. Heroux, Philip Soltero, and Ron Brightwell. Cooperative application/os dram fault recovery. In *Proceedings of the 4th Workshop on Resiliency in High Performance Computing (Resilience 2011) in Clusters, Clouds, and Grids*, Bordeaux, France, August 2011.

[8] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996.

[9] Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34:56–78, February 1991.

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

[11] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35:288–323, April 1988.

[12] Graham E. Fagg, Jack J. Dongarra, Graham E. Fagg, and Jack J. Dongarra. Building and using a fault tolerant mpi implementation. *International Journal of High Performance Computing Applications*, 18:2004, 2004.

[13] Kurt B. Ferreira, Ron Brightwell, and Patrick G. Bridges. Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Supercomputing'08)*. IEEE Computer Society, November 2008.

[14] Kurt B. Ferreira, Rolf Riesen, Patrick G Bridges, Dorian Arnold, Jon Stearley, James H. Laros, Ron Oldfield, Kevin Pedretti, and Ron Brightwell. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the 2011 ACM/IEEE Conference on Supercomputing (SC'11)*. IEEE Computer Society, November 2011.

[15] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.

[16] Qi Gao, Weikuan Yu, Wei Huang, and D.K. Panda. Application-transparent checkpoint/restart for mpi programs over infiniband. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 471 –478, aug. 2006.

[17] Felix C. Gärtner. Fundamentals of distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[18] Rinku Gupta, Pete Beckman, Byung-Hoon Park, Ewing Lusk, Paul Hargrove, Al Geist, Dhabaleswar Panda, Andrew Lumsdaine, and Jack Dongarra. Cifts: A coordinated infrastructure for fault-tolerant systems. *Parallel Processing, International Conference on*, 0:237–245, 2009.

[19] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[20] Andi Kleen. mcelog: memory error handling in user space. In *Proceedings of Linux Kongress 2010*, Nuremburg, Germany, September 2010.

[21] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[22] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX'10)*, Boston, MA, June 2010.

[23] Y. Liang, Y. Zhang, A. Sivasubramaniam, R.K. Sahoo, J. Moreira, and M. Gupta. Filtering failure logs for a bluegene/l prototype. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 476 – 485, june-1 july 2005.

[24] Adam Oliner, Larry Rudolph, and Ramendra Sahoo. Cooperative checkpointing theory. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 132–132, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Adam J. Oliner, Larry Rudolph, and Ramendra K. Sahoo. Cooperative checkpointing: a robust approach to large-scale systems reliability. In Gregory K. Egan and Yoichi Muraoka, editors, *ICS*, pages 14–23. ACM, 2006.

[26] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of SC*, 2003.

[27] Fred B. Schneider. *What good are models and what models are good?*, pages 17–26. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[28] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, June 2006.

[29] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. *Communications of the ACM*, 54:100–107, February 2011.

[30] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[31] Rajagopal Subramaniyan, Vikas Aggarwal, Adam Jacobs, and Alan D. George. Fempi: A lightweight fault-tolerant mpi for embedded cluster systems. In *Proc. International Conference on Embedded Systems and Applications (ESA), Las Vegas*, pages 26–29.

DISTRIBUTION:

1 MS 1319 Kurt Ferreira , 1423
1 MS 0899 Technical Library, 9536 (electronic copy)
1 MS 0359 D. Chavez, LDRD Office, 1911

Sandia National Laboratories