Understanding Fault Scenarios and Impacts through Fault Injection Experiments in Cielo

Valerio Formicola*, Saurabh Jha*, Daniel Chen*, Fei Deng*, Amanda Bonnie†, Mike Mason†,
Jim Brandt‡, Ann Gentile‡, Larry Kaplan§, Jason Repik§, Jeremy Enos¶, Mike Showerman¶, Annette Greiner||,
Zbigniew Kalbarczyk*, Ravishankar K. Iyer*, and Bill Kramer*¶
*University of Illinois at Urbana-Champaign
Urbana-Champaign, IL 61801
†Los Alamos National Laboratory (LANL)
Los Alamos, NM 87544
‡Sandia National Laboratories (SNL)
Albuquerque, NM 87123
§Cray, Inc.
Seattle, WA 98164 and Albuquerque, NM 87112
¶National Center for Supercomputing Applications (NCSA)
Urbana, IL 61801
||National Energy Research Science Computing Center (NERSC)
Berkeley, CA 94720

***Abstract*—We present a set of fault injection experiments performed on the ACES (LANL/SNL) Cray XE supercomputer Cielo. We use this experimental campaign to improve the understanding of failure causes and propagation that we observed in the field failure data analysis of NCSA's Blue Waters. We use the data collected from the logs and from network performance counter data 1) to characterize the fault-error-failure sequence and recovery mechanisms in the Gemini network and in the Cray compute nodes, 2) to understand the impact of failures on the system and the user applications at different scale, and 3) to identify and recreate fault scenarios that induce unrecoverable failures, in order to create new tests for system and application design. The faults were injected through special input commands to bring down network links, directional connections, nodes, and blades. We present extensions that will be needed to apply our methodologies of injection and analysis to the Cray XC (Aries) systems.**

***Keywords*—fault injection; resilience assessment; network recovery.**

I. Introduction

As we move to exascale systems, we expect to observe much higher error rates [1], [2]. To overcome the "reliability wall" [2], i.e., the upper bound of the reliability of an HPC system, we need to understand fault-to-failure scenarios and identify optimal places to instrument the system for detecting and mitigating faults. In our previous study [3] on the Blue Waters supercomputer, we showed the criticality of network-related failures and failures of the network recovery in Cray XE platforms by providing empirical evidence of the impact of those failures on applications and system. Understanding fault-to-failure scenarios based on production data is difficult because the analysis is constrained to naturally occurring events, and there is a lack of information on fault locations, the health state of the system, and the workload conditions. In particular, multiple errors and failures make it difficult to diagnose and understand the reasons for some fault-to-failure propagation paths. In this work, we focus on improving the understanding of fault propagation in interconnection networks by presenting the results of fault injection experiments conducted on Cielo, a petaflop Cray XE system with nine thousand nodes designed and developed jointly by Los Alamos National Laboratory (LANL) and Sandia National Laboratories (SNL) under the Advanced Computing at Extreme Scale (ACES) partnership. After production, but before retirement, the ACES partnership gave us exclusive access to Cielo for performing our fault-injection experiments.

Fault injection (FI) methods have been widely used to investigate fault-to-failure propagation and its impact on applications and systems, since it is possible to control fault conditions, and decide on workload and instrumentation on the target system. To support our fault injection experiments, we developed *HPCArrow*, a software-implemented fault-injection (SWIFI) [4] tool. We perform fault injection experiments that emulate permanent faults at the hardware component level. Such a fault injection approach can create and test various failure scenarios (such as failures during recovery) by injecting combinations of one or more faults. To the best of our knowledge, this is one of the largest fault injection studies to date.

The contributions and results of this work are summarized below:

- **Network fault injection tool for large-scale supercomputers:** We designed and developed *HPCArrow*, a tool to execute fault injection experiments systematically. *HPCArrow* allowed us to inject faults on a petaflop supercomputer. We executed 18 fault injection experiments, which led to failures of 54 links, 2 nodes, and 4 blades. The tool was successfully used to investigate and validate failure scenarios presented in [5], [6], [3] and establish in-depth fault-to-failure propagation and delays.

- **Recommendation for notification and instrumentation at application and system levels:** FI experiments revealed a lack of instrumentation of network-related hardware errors. That lack resulted in a lack of real-time feedback to applications. The long time it takes to recover presents a unique opportunity to feed information to an application/system to improve its resiliency to network-related failures. For example, application and system resource management software does not get a notification when a network deadlock occurs, leading to waste of computing resources and application hang. Placing additional detectors and/or a notification system on the health supervisory system (HSS), which is unaffected by failures on the high-speed network (HSN), could be used for communication or triggering of higher-level mechanisms in addition to transmission of low-level fault information to the SMW and recovery directives from the SMW. In addition, as the use of node-local non volatile storage becomes more common, the options for checkpointing to local disk/memory without requiring network access should be explored
- **Identification of critical errors and conditions:** The analyses of error data obtained from FI experiments helped us identify critical errors and conditions that can be used to provide real-time feedback to applications and resource managers. For example, 1) at the system level one can detect and send notifications of network deadlock conditions, and 2) at the application level, one can send notifications of critical errors that can lead to corruption or abnormal termination of applications.

This paper is organized as follows - Section II outlines the motivation for this work. Section III describes our approach to the fault injection campaign. Section IV provides details of the targeted FI scenarios and Cray's automated recovery mechanisms. Section V describes the fault injection tool we developed for this work, and Section VI describes our event and impact analysis methodology and tools. Section VII presents the results of the fault injection experiments. Section VIII explains how the results of our experiments can be used to improve resilience in HPC systems. Section IX describes our progess in extending our work to Aries systems. Section X presents related work, and we conclude in Section XI.

II. Motivation

In HPC systems to date, application resilience to hardware and system software failures has largely been accomplished using the brute-force method of checkpoint/restart [7], which allows an application to make forward progress in the face of system faults, errors, and failures independent of root cause or end result. It has remained the primary resilience mechanism because of the difficulty to design and implement effective fault tolerant program models (e.g., the MPI User Level Failure Mitigation approach). However, as we move from petascale to exascale, shortened mean time to failure (MTTF) may render the current checkpoint/restart techniques ineffectual. Instrumentation and analysis methods that provide early indications of problems and tools to enable use of new windows of opportunity for mitigation by system software and user applications could offer an alternative, more scalable solution.

Because of the evolutionary nature of HPC technologies, it is expected that systems, for the foreseeable future, will continue to have fault mechanisms and behaviors similar to those found in current deployments [8]. Thus, comparisons of well-explored failure scenarios across multiple generations of systems should enable identification of persistent high impact fault scenarios. Tailoring instrumentation and resilience techniques to enhance system and application resilience characteristics in these high impact scenarios can enhance the efficiency and throughput of both current and future platform architectures.

Even system recovery mechanisms that are defined and implemented by HPC platform vendors are typically not well understood or characterized by their signatures in log files and platform measurables in terms of durations, impacts, and success rates, particularly for complex fault scenarios. A number of studies have explored system logs from large-scale HPC systems [9], [5], [10], but connecting the failures with the root causes or precursor faults has proven difficult at best. The resulting fault-to-failure path models are rarely complete, and typically there is a significant amount of associated uncertainty. In addition, built-in, automatically triggered recovery mechanisms can further obscure failure paths and may leave no trace in the log files typically used by system administrators and made available to researchers.

The research community needs a way to verify, and possibly augment, failure models through testing in a controlled environment. In particular they need tools to enable documented and repeatable HPC environment configuration, including instrumentation and repeatable applications placement, and injection of known faults in a repeatable non-destructive manner on large scale HPC systems.

III. Approaches

In order to gain a well-informed, data-driven understanding of fault behavior characteristics and fault-to-failure paths we utilize a combination of two approaches: 1) log file analysis to identify recurring and catastrophic failure scenarios and 2) Fault Injection (FI) for testing/validation/augmentation of hypothesized root causes and fault to failure paths.

In order to make our approach generally applicable to multiple generations of large scale HPC platform architectures, we have developed generalized tools in both of these areas. Our log analysis tools, LogDiver [6] and Baler [11], are used to identify and prioritize failures and to identify correlative associations among faults/failures. Our FI toolkit, *HPCArrow*, is used to cause (inject) and log faults and service restorations on targeted HPC system components in a consistent manner. More detail about *HPCArrow* tool is presented in Section V.

The subject of this paper is FI which utilizes artificially induced, hypothesized or previously observed, initial fault scenarios to induce reactions expected to lead to failure or invocation of automated recovery mechanisms. Use of a dedicated machine for FI experiments enables better control over initial system state than typically exists during production operation of a large scale HPC system. This also makes observation of the resulting failures and, if they exist, corresponding resilience mechanisms more straightforward. These types of experiments can be repeated many times in order to provide a statistically significant set of results. In order to utilize FI in a consistent way we have developed a generalized FI toolkit. This toolkit, *HPCArrow*, enables us to inject targeted faults into system components, such as nodes and network links, which are controlled in terms of location and timing, e.g., can inject additional faults during recoveries from earlier faults.

The injected faults are based on scenarios derived through prior use of LogDiver on system logs containing fault and failure information.

The remainder of this paper is devoted to description of FI experiments run on Cielo [12] (described below) for targeted failure scenarios, use of *HPCArrow* to trigger those scenarios, and analysis of the data collected across the system during the experiments.

Basic observable data consists of a variety of system log data collected during the fault injection experiments. Job impact data includes job output information such as completion status, nodes used, and run times. Where possible we augment these with additional system wide, periodic (1 second) collection of system resource utilization and state data, such as network traffic and link state.

For each experiment, we correlate contextual information on the fault injection with the system logs and other observables. We calculate metrics, such as recovery duration, time spent in each phase of recovery, and retries from fine-grained statistics of the experiments. We then assess the metrics in order to determine high-impact fault scenarios with potentially actionable timescales in order to make recommendations for improving resilience through additional instrumentation and notification mechanisms or improved architectural designs in future systems.

Our approach is architecture-independent; however, the details of the injections, measurements, and recoveries are architecture-specific. In this work, we target the Cray XE systems. We leverage our characterization of operational faults and failures in Blue Waters [3] to design and analyze our FI experiments. The platform used for these FI experiments was Cielo, a petaflop Cray XE system at the Advanced Computing at Extreme Scale (ACES) system (an initiative of the Los Alamos National Laboratory (LANL) and Sandia National Laboratories (SNL)) which consists of 8,944 compute nodes with a Gemini 3D torus with dimensions 16x12x24. Each blade in Cielo includes two Gemini application-specific integrated circuits (ASICs), each housing two network interface controllers (NICs) and a 48-port router. Each ASIC acts as a router and is connected to the network by means of directional connections X+, X-, Z+, Z-, Y+, and Y-. Directional connections X+, X-, Z+, and Z- are made of 8 links, and Y+ and Y- are made of 4 links each. Each link is composed of 3 *channels* (or *lanes*). In this paper we refer to directional connections simply as *connections*.

The elements of our approach, particularly as they pertain to Cray XE systems, are described in more detail in the next three sections. Section IV describes the investigated faults, Section V describes our FI tool *HPCArrow*, and Section VI describes the log and numeric data, analysis tools and methodologies, and results.

IV. Failure Scenarios

In this work we investigated failures in compute (nodes and blades) and network (links, ASICs, and connections) components in isolation and in combination (shown in Figure 1). These particular types of failures were targeted as they occur frequently enough in production systems to be responsible for significant performance degradation. The Cray XE system is designed to handle these types of failures by triggering automatic *recovery procedures* (as shown in Figure 2). Failures, depending on the occurrence location, are detected by a supervisory block on the Gemini ASIC, a blade controller (BC) on the blade, or a System Management Workstation (SMW). Each BC is locally connected to a supervisory block on the Gemini ASIC, and remotely connected to the SMW through the Cray Hardware Supervisory System (HSS) network. Information about critical failures is delivered to the SMW by a failure impacted BC for initiating any necessary recovery. Upon detection of a network-related failure, the SMW initiates a system-wide recovery. Actions taken by the SMW during the recovery depend on the failure type. A connection failure, for example, would lead to a loss of connectivity between two Gemini ASICs. For connection failures the SMW recalculates the routes, quiesces the network (i.e., injection into the network is paused), installs the newly calculated routes on all ASICs, and unquiesces the network. In the case of a single link failure that does not result in a connection failure, the failed link is masked (i.e., removed from service) and the ASICs maintain connectivity through the remaining functional links of that connection. In some cases, recovery mechanisms can mask failure(s) without causing a major interruption of the system. Analyses of field failure data indicate that: 1) recovery mechanisms handling complex failure scenarios may not always succeed and 2) protracted recoveries that eventually succeed may still have a significant impact on a system/application(s). In this study, we created failure scenarios using FI in order to understand the system's reaction and susceptibility to some fault/failure scenarios seen to occur on production systems.

Failure scenarios are defined by specifying the location and timing of the faults. The failure scenarios studied in this work are described below.

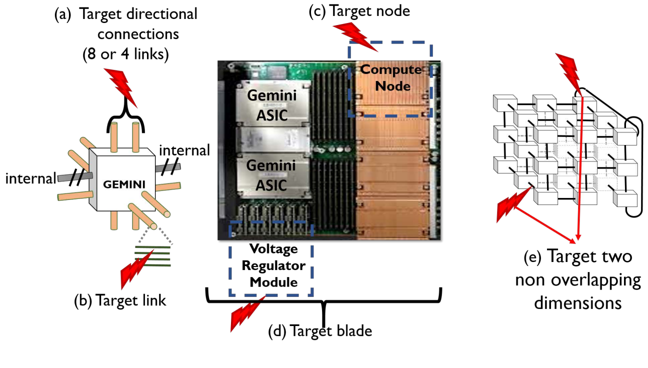

Fig. 1: Target components of fault injection experiments. A failure of a blade leads to failure of two ASICs.

Node failure: We recreate a *node failure* by powering off one of the nodes running an application on the system. The failure of the node will result in the failure of the application.

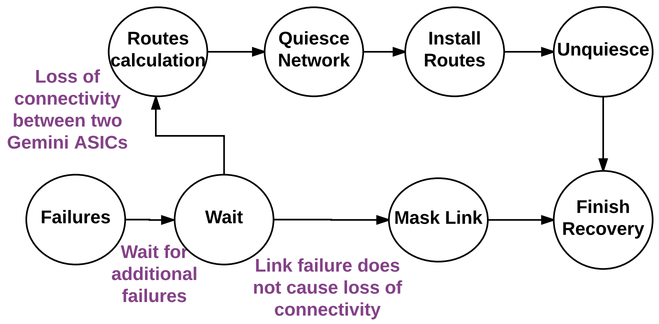
Routes calculation
Quiesce Network
Install Routes
Unquiesce
Loss of connectivity between two Gemini ASICs
Failures
Wait
Mask Link
Finish Recovery
Wait for additional failures
Link failure does not cause loss of connectivity

Fig. 2: Recovery procedures of the Cray Gemini high-speed network: main stages of the recovery. Additional failure(s) at any stage will restart the network recovery operation.

No automatic network recovery response is expected in this case, as the failure of one or more nodes does not impact the routing paths in the network. However, there can be effects on network traffic balance across remaining nodes and network elements which can, for other networks and topologies (e.g., Aries dragonfly), affect routing decisions.

Link failure: We recreate a *link failure* by deactivating one of a connection's links via modification of a status flag on one of the two Gemini router ASICs connected by the link. Injecting faults in this way emulates a scenario in which the flag is modified to deactivate a link that is physically damaged or is unavailable due to other problems (e.g., a high soft-error rate). When a link is taken down by modifying a status flag on the router on one end, the router on the opposite end of the link is also affected. The link failure is detected by the Hardware Supervisory System (HSS); the hardware masks that link and traffic automatically uses the other links in the connection. After the automated recovery completes, the link is disabled (marked down) on the SMW.

Blade failure: In this scenario, we recreate a *blade failure* by turning off the voltage regulator of the *mezzanine* in the blade. When the entire blade is powered off, there is a *concurrent* shutdown of the four associated compute nodes and two Gemini router ASICs, each with 40 network fabric links. When the fault is injected, the blade becomes unavailable for computation and routing network traffic. Automatic recovery is expected to reroute around the failed routers.

Multiple sequential link failures: In this scenario, a sequence of link faults are injected one after another over a user configurable duration of time (typically sub-second). Depending on the failure detection latency and time between injections, the SMW may recover all of the failed links together (i.e., a single recovery procedure), or recover each one of them sequentially (when the time between two successive failure detections is longer than the *aggregate_failures step* [13] time of 10 seconds), or identify additional faults during a recovery. In the case of identification of additional failures during recovery, the SMW aborts the current ongoing recovery and starts a new recovery that addresses the new failure(s) in addition to the previous failures.

Single and multiple connection failures: We defined two modes for *connection* failures: *single connection* and *multiple connections*. For *single connection failures*, we sequentially inject faults into all links of a target connection. In a torus topology, each router *connection* consists of 8 links for each of X+, X-, Z+, and Z- directional connections, and 4 links for each of Y+ and Y-. Failing a *connection* creates a hole in the routable topology. The associated recovery is expected to reroute the network paths around the hole. In the case of *multiple connection failures* we target two *connections* which do not share a common Gemini router ASIC. To create this failure scenario, we randomly chose two blades whose location differs in all dimensions X, Y, Z. The automatic recovery should be able to route around the failed *connections*. Note that unrouteable topologies [13] do exist and will cause a reroute failure.

V. Fault Injection Tool: HPCArrow

We have developed ***HPCArrow***, a software-implemented fault-injection (SWIFI) [4] tool and methodology that can inject one or more faults into specific target locations (currently nodes, blades, and/or links) in the system. Those faults may in turn invoke various recovery procedures in the system, as discussed in Section IV.

HPCArrow (refer to Figure 3) consists of three major modules for systematically studying the effects of faults/failures on HPC systems/applications: (1) a *Workload manager* generates workloads and submits all applications to the selected nodes for execution; (2) a *fault injector* selects the fault type along with the target location and timing of injections; and (3) a *restoration manager* restores the system to a healthy state (i.e., resets the system to the state before the injections) or, in critical scenarios, it issues a notification stating that the entire system must be restarted manually[1]. The tool supports execution of arbitrary failure scenarios consisting of network- and compute-related failures (refer to Table I). Currently, HPCArrow is preconfigured with all fault injection campaigns discussed in this work. A fault injection *campaign* specifies failure scenario(s) and the workloads to run concurrently during the FI experiment(s). For this study, we ran the tool in a supervision mode in which all commands to be executed by the tool could first be verified by the user in order to reduce wasted time spent in error recovery and bug fixes.

To conduct a fault injection experiment[2] using *HPCArrow*, a user selects one of the available preconfigured campaigns through a simple user interface (step 1, **S1** in Figure 3). *HPCArrow* then launches a set of applications (defined by the workload) to be executed on the system (step 2, **S2**), and verifies their execution (step 3, **S3**). In step 4 (**S4**), *HPCArrow* injects faults in sequence as defined in the *campaign* configuration file. During this phase of operation, there may be automated system responses to the injected faults along with associated log output.

Upon completion of the experiment(step 5, **S5**), the user invokes the restoration manager (step 6, **S6**), which restores the system to a healthy state. After restoration completes, the user

[1] The ability to detect critical scenario such as a deadlock by active monitoring of logs is available only in the latest version of the tool and was a result of this study. This feature was not present during the experiments discussed in this paper.

[2] *HPCArrow* launches only one campaign during an experiment and collects data for that experiment.

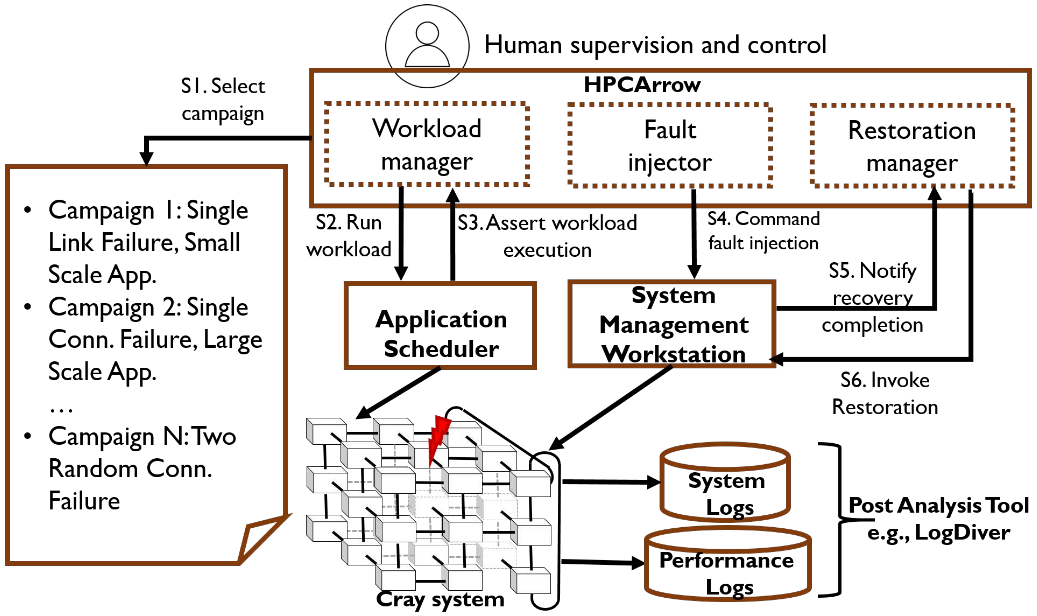
Human supervision and control
HPCArrow
S1. Select campaign
Workload manager
Fault injector
Restoration manager
Campaign 1: Single Link Failure, Small Scale App.
Campaign 2: Single Conn. Failure, Large Scale App.
...
Campaign N: Two Random Conn. Failure
S2. Run workload
S3. Assert workload execution
S4. Command fault injection
S5. Notify recovery completion
S6. Invoke Restoration
Application Scheduler
System Management Workstation
System Logs
Performance Logs
Post Analysis Tool e.g., LogDiver
Cray system

Fig. 3: *HPCArrow*: A network fault injection tool for HPC systems. Data produced during an experiment or campaign can be further analyzed using tools like LogDiver. The steps taken by *HPCArrow* to launch fault injection experiments are shown as S1, S2, ..., S6.

is allowed to run further fault injection campaigns. *HPCArrow* reports the results of the campaign on the user console and collects all the relevant data for further analysis of the system/application behavior during the experiment. HPCArrow allows to verify the execution of the steps using the output on the administrator console, and the logs generated in the system. For example, when a restoration is interrupted, the administrator console shows errors. In that case, the administrator can retry a warm swap or blade reboot commands. A more in-depth description of the *HPCArrow* modules, along with examples that illustrate the scenarios, follow.

A. Workload Manager

To activate the injected faults, *HPCArrow* launches a mix of applications (specified in the campaign via the *Workload manager* module) at various scales. The scale in this context is defined by the number of nodes occupied by an application job. In this study, we defined three application scales: a *nano-scale* application executed on fewer than 512 nodes; a *small-scale* application executes on more than 512 but fewer than 1,024; a *medium-scale* application executes on more than 1,024 but fewer than 4,096 nodes; and a *large-scale* application executes on more than 4,096 nodes. In all fault injection experiments, we applied the same workload, which consisted of five small-scale, two medium, and one large-scale application. We used Intel MPI Benchmarks (IMB) [14] as the benchmark application. IMB performs a set of MPI performance measurements for point-to-point and global communication operations for a range of message sizes. Use of this application enabled us to measure the effects of injected faults on the performance and resilience of network-intensive applications.

B. Fault Injector

The *Fault Injector* module is responsible for executing the commands that inject faults into the system components. In the case of multiple overlapping fault injections, this module is responsible for timing the injection of faults with respect to each other. These commands and the underlying mechanisms employed are system-specific. The commands for fault injections (refer to Table I) recreate the failure scenarios described in Section IV. Each one indicates the type of target component (i.e., link, node, connection, or blade/mezzanine). Faults can be injected either manually at a specific location (selected by the operator) or randomly (selected by the tool) in the system. The injection commands are issued from the SMW by the system administrator. Each target component is uniquely identified by a symbolic name (Cray physical ID or *Cname*).

C. Restoration Manager

The system restoration commands (refer to Table II) are issued by the user/administrator to restore the system to the state that preceded the execution of the fault injection experiment. Those commands are based on the Cray XE commands [13]. Each campaign, along with the workload and fault injection, specifies the steps to be run (automatically) for the restoration of the component injected. Note that the *Restoration Manager* recovers blades (BR) and links (LR) by executing the *warm swap* command.

VI. Analysis Methodology

We analyze the system-generated logs and measurement data in the context of the fault injection experiment to quantify metrics (e.g., recovery duration, time spent in each recovery

TABLE I: Failure scenario commands and acronyms used in this paper

Target	*Failure scenario commands*	*Concurrency*	*Description*
Node	*NF* *(Node Failure)*	Single	Power-off of a node in a blade.
Link	*LF* *(Link Failure)*	Single	Activation of link alarm status on the router.
	SCF *(Single Connection Failure)*	Sequential	Activation of the alarms for a set of links composing an entire directional connection X, Y, Z between two routers.
	2CF *(2 Connections Failures with Non-Overlapping Dimensions)*		Activation of the alarms for a set of links composing 2 directional connections among routers with different coordinates in the network topology.
Blade	*BF* *(Blade Failure)*	Concurrent	Power-off of the voltage regulator in a blade mezzanine.

TABLE II: Restoration commands issued by the administrator with the support of the *HPCArrow* tool.

Restoration commands	*Description*
BR *(Blade Restoration)*	BR is a sequence of commands executed after an injection to a blade: crayadm -c ‘xtwarmswap –remove blade_Cname’ + crayadm -c ‘xtwarmswap –add blade_Cname’ + crayadm -c ‘boot CNL0 blade_Cname’
LR *(Link Restoration)*	Warm swap on links addressed using their Cnames: crayadm -c ’xtwarmswap -s link_1,. . . ,link_N -p p0’

phase, number of retries to restore the system operation, and network pause time) for assessing the impact of the failure scenarios injected on the applications/system. The data collected and the analysis techniques are described in this section.

A. Event Analysis: LogDiver

LogDiver [6] is a tool for the analysis of system- and application-level resiliency in extreme-scale environments. The LogDiver approach is to create a unique data set that encapsulates events that are essential to performing resiliency and performability measurements. In the context of this study, the tool allows us to (1) extract network-recovery operations, determine the completion status of the recovery, and diagnose the cause of recovery failures [3]; and (2) identify application termination status and potential reasons behind abnormal terminations of applications. Specifically, LogDiver filters the logs (collected from the fault injection experiments) that match the regular expressions configured in the tool. Once filtered, data are used by LogDiver to compute metrics of interest.

B. Network Performance Counters

The main source of the numerical monitoring data provided by Cray XE systems is the Systems Environmental Data Collection (SEDC) [15] system, which mainly collects data such as temperature, fan speeds, and voltages. While such data can potentially be used in a resilience context for detecting abnormalities that might be indicators of degrading components, their use in this work is limited. More relevant indicators of the effects of the fault injections on the system and application state, such as network traffic, stalls, and link status, are exposed on nodes, but are not normally collected or transported. In this work we augmented the available system data with our own collection and transport of these network data.

Low-level network counters are available via Cray’s Gemini Performance Counter Driver [16] (*gpcd*) and by entries in a `/sys fs` interface available via Cray’s *gpcdr* kernel module. The latter method in Cray’s default configuration presents the metrics as directional-link aggregated quantities. We are using the *gpcdr* interface as our data source, with a refresh rate of one second. Details on these quantities can be found in [17].

In the Gemini network, any given router may be responsible for handling traffic both for jobs allocated to the nodes directly attached to the router and for other jobs’ traffic that passes through that router along their communication paths. In the Gemini network, routing is primarily deterministic. Traffic goes first from source X coordinate to destination X coordinate, then from source Y coordinate to destination Y coordinate, and then from source Z coordinate to destination Z coordinate. This most likely means that a router will handle traffic for multiple applications. The values shown in this work for routers are thus the aggregates of traffic handled over the *connection* of a router; they cannot be attributed to any particular job(s) nodes and are subject to instantaneous demands of those jobs, including job starts and stops.

Numerical data were collected at 1-second intervals via the Lightweight Distributed Metric Service (LDMS). Details of LDMS data collection on Gemini systems, including overhead assessments demonstrating that there is no significant detrimental system impact, are provided in [18], [19]. Of relevance here is the fact that data are collected by on-node daemons and held in memory on a node until they are overwritten by the next sample. Data are pulled from that memory location by other daemons via RDMA. If network connectivity is lost, as it would be during a full-system quiescence, data points collected during that period are lost.

While in our other XE/XK systems with more current CLE versions, the link status was observed to change in response to downed individual links, we did not see this behavior on Cielo. This did not allow us to monitor traffic on single links, but on entire connections.

C. Application Data

As mentioned in the previous section, we ran IMB benchmarks to study the impact of faults/failures on applications. Variability in the application run times was significant enough to impact our ability to draw conclusions from run-time performance. However, since the jobs were network-intensive, we can consider the impact of the injections on the instantaneous network traffic and communication. In addition, information in the job output files provides some insight on MPICH errors and some network events (e.g., throttles) of interest. As part of this work, we discovered that the reporting of node quiesce event counts is not always accurate, and thus it is not included in our assessments.

D. Output of Analysis

We summarized each fault injection experiment in the form of a report like the one given in Table III. This schema allows us

TABLE III: Example summary for one fault injection experiment.

Entry	***Value***
Experiment ID	5
Start Time	1473176186
End Time	1473176880
Experiment Window [hours]	0.192777778
Failure Scenario	SCF (Random)
Components Targeted	8
Errors on Admin Console	No
Recovery Time [seconds]	630
Number of Recovery Procedures	4
Number of Procedures: Success	2
Number of Procedures: Failure	2
Is Last Recovery Failed?	No
Application Errors	1
Warm Swap Failed	0
Gemini Link Failed	32
EC Node Failed	0
Gemini Link Recovery failed	2
Gemini Lane Recovery failed	0
Gemini Channel Failed	32
Blade Recovery Success	0
Warm Swap Success	0
Link Recovery Success	4
...	

to identify experiments that showed anomalous logs (e.g., high volumes or unusual hardware error logs) and to characterize the impact on the applications in terms of network traffic. Each experiment is represented with a set of parameters: an ID to uniquely identify an experiment, an acronym for the injection/restoration performed, the links/nodes/blades targeted by the fault injection command or the object of the restoration operation, the presence of errors on the administrator console during the experiments, the number of network recovery operations in the time interval of the experiment, the number of successful and failed recoveries, and the counters for the events[3] collected from the data sources (system-generated logs). In Section VII, we discuss the results of the fault injection experiments conducted on Cielo.

VII. Results from Fault Injection Experiments

We analyze the results of fault injection experiments by characterizing failures, recoveries, and application/system impact. We first summarize the results across all experiments and then present example cases for each failure scenario studied in this work.

Each case study is supported by a graphical depiction of the key events corresponding to a given fault injection experiment (i.e., time and location of the injected fault), system behavior (in terms of the network traffic) at each step of a fault propagation, and the system response (i.e., recovery actions) to injected faults. For example, Figures from 4 to 8 show the traffic distribution captured using LDMS (top subplot), injection and network activity as seen from SMW filtered out using LogDiver (middle subplot), and hardware error events reported by the health checker systems of the Cray system and filtered by LogDiver (bottom subplot). The traffic distribution subplot shows the fraction of traffic volume that has passed through a connection between time T=0 and any given time T=t compared to the total volume passed through it during the fault injection experiment. In the top subplot the "injected connection" line represents traffic flowing through a Gemini router ASIC on the connection targeted with fault injection, and the "other connections" line represents average traffic flowing through the same Gemini router ASIC on other connections. During the experiments, hardware error logs were generated after the fault injection. Some errors showed anomalies in terms of number of occurrences (count) and persistence (duration over which an error was reported). We reported the distributions of those hardware errors, calculated as the fraction of hardware error events that occurred between time T=0 and any given time T=t compared to the total number of errors (of the same type) encountered throughout the fault injection experiment (bottom subplots in Figures 4 to 8). We summarize the information available for those hardware error logs below.

- *ORB RAM Scrubbed Upper Entry*: The Output Request Buffer (ORB) frees the upper 64 entries in the ORB RAM by monitoring the number of clock cycles an entry has been in the ORB RAM since the entry was written. An error entry is logged for every network request that was scrubbed because of a timeout. Similarly, ORB can free the lower 64 entries of the ORB RAM in an action called an *ORB RAM Scrubbed Lower Entry*. Because the two have similar behavior, only *ORB RAM Scrubbed Upper Entry* is shown in the figures. It is a transient error that could be indicative of critical network issues if continuously generated, e.g., in the case of a deadlocked network.

- *ORB Request with No Entry*: This error is generated when a response packet comes into the receiver response FIFO buffer that does not correspond to a full request entry in the ORB RAM. These are critical errors that require the killing of uGNI Generic Network Interface threads. uGNI threads are used by MPI applications, and thus this error indicates a critical condition for MPI applications as well.

- *Receiver 8b10b Error*: This error indicates a transmission error and is reported by the completion queue.

- *LB Lack of Forward Progress*: A lack of forward progress is detected on NIC0 or NIC1, indicating that all subsequent requests destined for those NICs will be discarded, thereby stopping any traffic flows through those NICs. If this error message is reported by several components, it may indicate a critical issue in the network.

- *NW Send Packet Length Error*: This error is generated by packet corruption during the transmission.

- *SSID Stale on Response, SSID Stale*: These messages can be caused by problems in the ORB of the network cards. Usually, they are not related to transmission timeouts.

- *NIF Squashed from Tile Request*: Packets are squashed because of failed consistency check (e.g., ECC, CRC, misroute). Possible causes for these errors are packet corruption or bad routing.

[3] LogDiver encodes each event type in the form of a regular expression that matches one or more lines in the logs.

A. Summary of Fault Injection Experiments

We used HPCArrow to execute 18 fault injection campaigns, coordinated from the SMW of Cielo by the system administrator. Faults were injected into *links, connections, nodes*, and *blades*. Table IV(a) summarizes the fault injections classified by target type (node, link, connection, blade) and acronym (according to Table I). We indicate the number of experiments executed (Exp), and the number of network recovery procedures (RP) successful or failed (S/F), as extracted from the system logs using LogDiver. We also reported if the overall recovery succeeded (ORS). LogDiver reconstructed 37 automatic recovery operations after the fault injections (26 successful and 11 failed). Finally, we report the mean and standard deviations of the recovery durations calculated using LogDiver[4].

Restoration commands to return the system to the state it was in the start of the campaign were executed 7 times on the blades and 10 times on single links. Table IV (b) summarizes the restorations initiated by the administrators and gives the occurrence of errors on the administrator console after the restoration command were issued (EAC). Manual link restorations completed successfully every time. Blade restorations failed five times due to misconfiguration of the *HPCArrow* restoration manager: once because of an error during the blade boot, twice because of errors during blade removals, and twice because of errors during blade additions, in the case of the restorations shown in the Table IV(b) for the BR case. This bug has since been fixed. Restorations of the failed components resulted in fifteen network recovery operations, two of which did not complete successfully because of critical network conditions (i.e., deadlock in the network) during an experiment.

Next, we describe example cases corresponding to each failure scenario studied in this work.

B. Node Failures and Single Link Failures

Execution of link injections with HPCArrow effectively recreates failure conditions that quiesce network traffic, trigger automatic recovery operations, generate system recovery and error logs, and are seen in production in similar Cray HPC systems.

The plot in Figure 4 shows the profile of network traffic when a link failure injection experiment (scenario LF in Table IV) is executed, when an automatic network recovery is executed by the system to reroute around the failed links, and when a successive manual restoration is issued by the operator to reintegrate the failed link into the system. The traffic on the two Gemini router ASICs connected by the link targeted by the injection was monitored using LDMS. The top part of the figure shows the traffic on the connection with the fault-injected link (solid line), and the average of traffic received by the router connected to the other end of the injected link (dashed line). The average is calculated on all the connections of that particular router. Drop-outs of data occur in two time intervals, i.e., after the link fault injection (since the system automatically recovers the network) and after the *warm swap* executed by the operator to restore the failed link. A network quiesce command sent to all the controllers of the Gemini router ASICs on the blades results in traffic being quiesced globally for 30 seconds. In both the initial link injection and successive restoration, data were lost for around 30 seconds (recovery intervals covered all the phases of the recovery). LogDiver reconstructed an automatic recovery procedure with a duration of 1 minute after the fault injection, and about 90 seconds after the link restoration (warm swap). In particular, the tool extracted logs reporting an automatic recovery operation for a failed channel, link inactive, network quiescence, a rerouting operation, and dispatching of new routes. For the *warm swap*, the tool reconstructed a recovery procedure consisting of a *warm swap start*, a *quiesce* and *unquisce*, a *reroute*, a dispatching of new routes, and a message of successful *warm swap*. The hardware error logs (bottom figure) reported only an event of "*NW Send Packet Length Error*", generated after the failure of the link.

TABLE IV: Summary of fault injection experiments (a) and restoration commands (b). Statistical parameters are not meant to imply that this set of experiments constitutes a statistically significant set.

[FS = Failure scenario, #Exp = Number of experiments, #RP (S/F) = Number of recovery procedures (succeeded/failed), ORS = Overall recovery success, EAC = Errors on admin console]

(a) Fault injection experiments on Cielo.

Target	**FS**	**# Exp**	**# RP**	**# RPS**	**# RPF**	**ORS**	**Duration (μ,σ) [seconds]**
Node	NF	2	0	0	0	Yes	-
Link	LF	6	7	7	0	Yes	(50,21)
Connection	SCF	4	15	9	6	Yes	(64,157)
	2CF	2	10	5	5	Yes 1/2	(32,29)
Blade	BF	4	5	5	0	Yes	(134.5, 82.5)

(b) Summary of restorations executed by the administrators after fault injections.

Target	**# Exp**	**# RP**	**# RPS**	**# RPF**	**ORS**	**Duration (σ,μ) [seconds]**	**EAC**
Link (LR)	10	10	10	0	Yes	(91,5)	No
Blade (BR)	7	5	3	2	Yes 2 No 5	(515, 365)	Yes 6/7

The system logs collected during the experiments of a *node failure* (scenario NF) show the system did not react to this injection at the network level (i.e., the number of recovery procedures is 0 for NF in Table IV), since no network rerouting was required for a compute node failure, as expected. The experiment for node failure generated a log from the application placement framework (ALPS) indicating that an application was killed for "*ec_node_failed*", i.e., there was a hardware error and the node was correctly marked down by the system.

C. Multiple Sequential Link Failures

To understand the impact of causing *connection* failures by inducing multiple sequential link failures, *HPCArrow* was used

[4] LogDiver does not provide accuracy within 1 second on recovery duration.

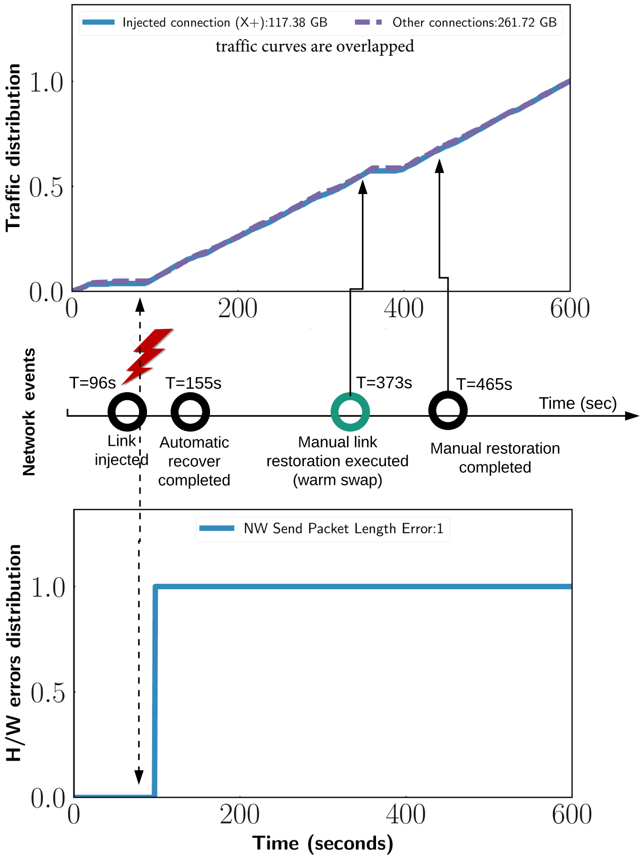

Fig. 4: Single link failure on connection X+ and restore. *Top*: Cumulative distribution of network traffic measured in the time interval of the experiment on the connection X+ and on the other router connected along this diraction . *Middle*: Time series of network events including the automatic recovery, and the warm swap executed by the operator after the fault injection (enlarged). *Bottom*: Cumulative distribution of anomalous hardware error logs during the experiment.

to emulate two different failure modes (refer to Section IV): single connection failures (**SCF**, executed in four experiments), and two non-overlapping connection failures (**2CF**, executed in two experiments). The relative timings and modes (2CF or SCF) of the injections resulted in three different recovery behaviors: (1) an invocation of a single network recovery with the recovery ending in success (observed in one experiment); (2) an invocation of multiple network recoveries with a final recovery ending in success (observed in four experiments); and (3) an invocation of multiple network recoveries with a final recovery reporting success, but the system ends up in a deadlock state (observed in one experiment).

1) Single network recovery completed successfully: Figure 5 shows the result of one of the sequential multiple *link* failure scenarios for a single *connection* failure experiment. In this specific experiment, an X+ *connection* was targeted. As can be seen from the top subplot in Figure 5, the network traffic volume that passed through the connections starting from time $T = 0$ continued to grow until $T = 230$ seconds. At $T = 230$ seconds, the campaign was executed, causing all eight links to fail within a 3 second window. The failure of all links on the injected *connection* triggered a recovery action requiring a route recalculation, and network *quiesce* for the new route instantiation (following the recovery procedure shown in Figure 2). The automatic recovery action resulted in the suspension of network traffic flow on all links throughout the system until the successful completion of the recovery (shown in the top subplot). This recovery took 630 seconds (ending at $T = 867$ s in the figure) to complete successfully. After the completion of the recovery, traffic started to flow in connections other than the failed one (increasing traffic on the remaining *connections*).

Analyses conducted on hardware error logs revealed anomalous behavior for certain error types. We had not observed any "*ORB RAM Scrubbed*" errors because other links, in the same direction, were available for sending traffic. In this case, these errors were observed during the entire recovery duration because of the loss of all links. Our analyses confirm the observation, as none of the *Link Failure (LF)* campaigns had produced "*ORB RAM Scrubbed*" errors; however, those errors were widespread in connection-failure (single or multiple) campaigns. Certain hardware errors (i.e., "*ORB Request with No Entry*" and the "*8b10b error*") appeared in only two experiments (this experiment and the *deadlock* failure scenario). The overall occurrence of "*NIF squashed request for a tile*" was also high, in this, relative to all other experiments. These anomalies, and the observed long recovery completion time for this experiment, are indicative of a problem in the network. However, we did not see any abnormal application terminations due to these errors. We did observe an MPICH2 error reporting a transaction failure for a large-scale application (4,096 nodes) running during this experiment. Since our application runs were limited to IMB benchmarks, we cannot determine if other MPI application will be able to tolerate "transaction failure" errors, but Cray's MPI is designed to be resilient in the face of many of these errors. ***Handling link recoveries can be a lengthy process whether single or multiple links of a connection have failed.***

2) Multiple network recoveries completed successfully: Figure 6 shows the effects of one of the sequential multiple *link* failure scenarios, in which a *SCF* fault injection experiment was conducted. In this experiment, a Z+ *connection* was targeted. The sequential failures of the eight links in the failed *connection* did not occur within a 10 second window, which resulted in additional faults occurring during recovery. In such cases, the SMW restarts the recovery, to account for any additional faults it encounters during recovery. However, this extends the recovery time. As shown in Figure 6, at $T = 100$ seconds, a fault was injected. Traffic was globaly *quiesced* (at around $T = 100$ seconds in the figure), and no network traffic flow was seen in the network until the completion of the recovery. The first two fault injections were within 10 seconds of each other (*1st link injected* and *2nd link injected*) and hence were handled together by the automatic network recovery response. Injection of the third fault (*3rd link injected*) caused a new *link* failure, triggering another automatic recovery response (*2nd recovery*). At $T = 185$ s, i.e. 28 seconds after the third fault was injected, a fourth fault was injected into the network (*4th link injected*), causing a *link* failure and abortion of the ongoing recovery (*2nd recovery fail*) a few seconds later. The ongoing recovery was aborted because the time difference between detections of the corresponding two link failures was more than 10 seconds. A retry of the recovery handling both the failures completed

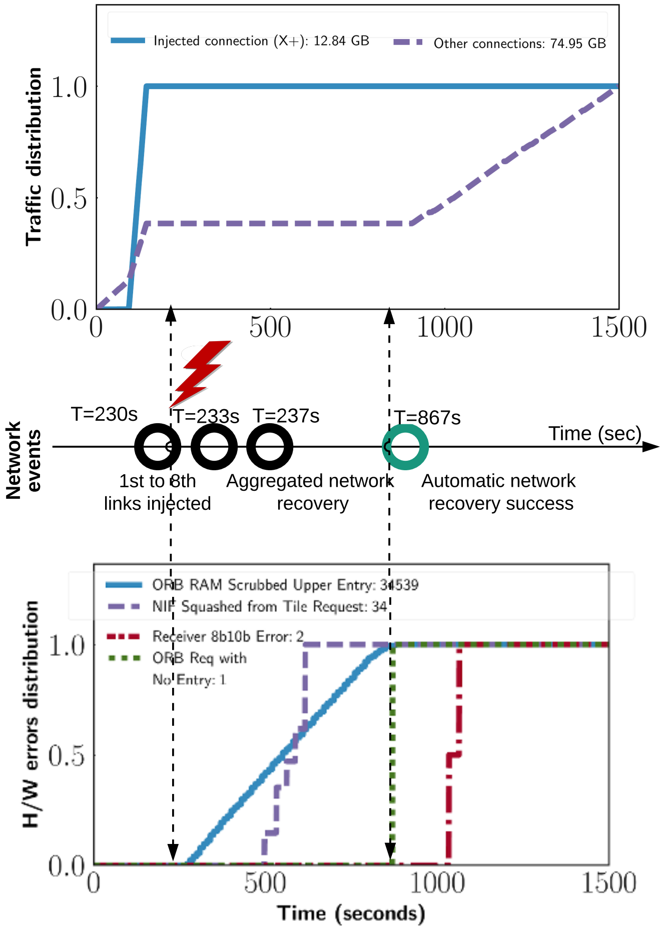

Fig. 5: Sequence of link injections in a short time. *Top*: Cumulative distribution of network traffic measured in the time interval of the experiment. *Middle*: Time series of network events (enlarged). *Bottom*: Cumulative distribution of anomalous hardware error logs during the experiment.

successfully. Injection continued on the rest of the links in this connection; observations for them were similar and hence are not shown or discussed here any further. During the campaign, a significant number of *"ORB RAM Scrubbed Upper Entry"* hardware errors were observed that were due to loss of links that could have been used by the ORB.

Another *sequential multiple link failure* scenario, in which a *2CF* (two connection failures with non-overlapping dimension) campaign was executed, led to observations similar to those described above (although in this case the recovery handled failures of links on two separate connections). Figure 7 shows the result of this campaign. Overall, the recoveries eliminated all traffic for around 150 seconds. The system was able to recover the network functionality, and the ORB messages disappeared after the recovery. However, not all *2CF* campaigns recovered the system successfully, which is discussed in Section VII-D.

A failure during recovery can lengthen the time to recover the system and may lead to multiple network *quiesce* and *throttle* events, which in turn can impact the system and application traffic.

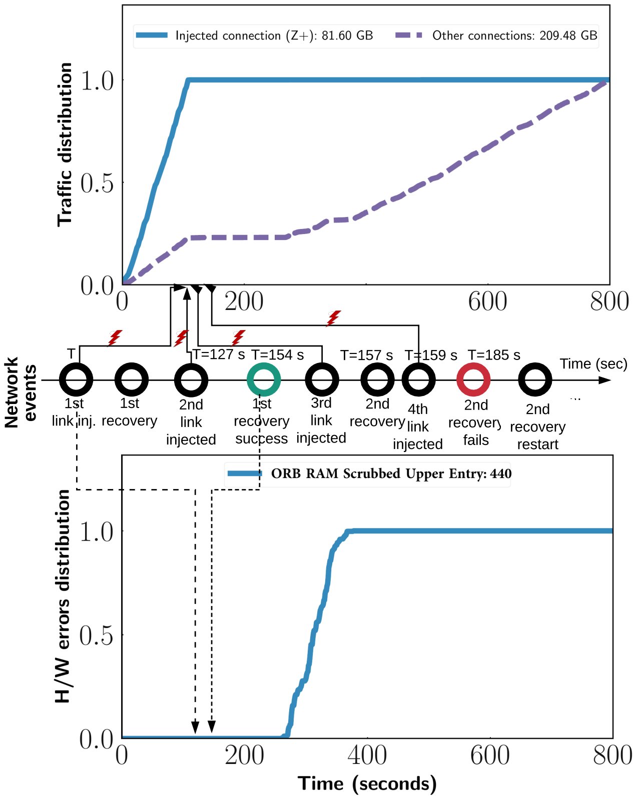

Fig. 6: Single connection fault injection as a sequence of link failures. *Top*: Cumulative distribution of network traffic measured in the time interval of the experiment. *Middle*: Time series of network events (enlarged). *Bottom*: Cumulative distribution of anomalous hardware error logs during the experiment.

D. Multiple Network Recoveries, Final Recovery Ending in Report of Success but System Deadlocks

Figure 8 shows the effects on the system and application of one of the sequential multiple link failures scenarios, in a *2CF* (two connection failures with non-overlapping dimension) campaign. Unlike the previous experimental campaign for *2CF* (refer to Figure 7), in this experiment two targeted *connections* belonged to different dimensions (one in X+ and another in Y+). As in all other cases discussed so far, the network traffic flow stopped during the recovery. However, unlike the other cases, traffic flow was never successfully restored after the recovery seemingly completed successfully (as reported by the SMW) within 120 seconds of the start of the campaign. A large number of "*ORB RAM scrubbed*" errors were continuously observed, despite the successful completion of automatic network recovery (which also was not the case in other campaigns). The analyses of hardware error logs revealed that ORB messages were being generated by increasing numbers of components over time. This is anomalous behavior, as the system is not expected to generate "*ORB RAM Scrubbed*" errors after the successful installation of correct routing tables. "*LB Lack of Forward Progress*" messages from the Gemini router ASICs further strengthens the belief that traffic could not be routed to its destination even after the recovery. The "*LB Lack of Forward Progress*" errors are

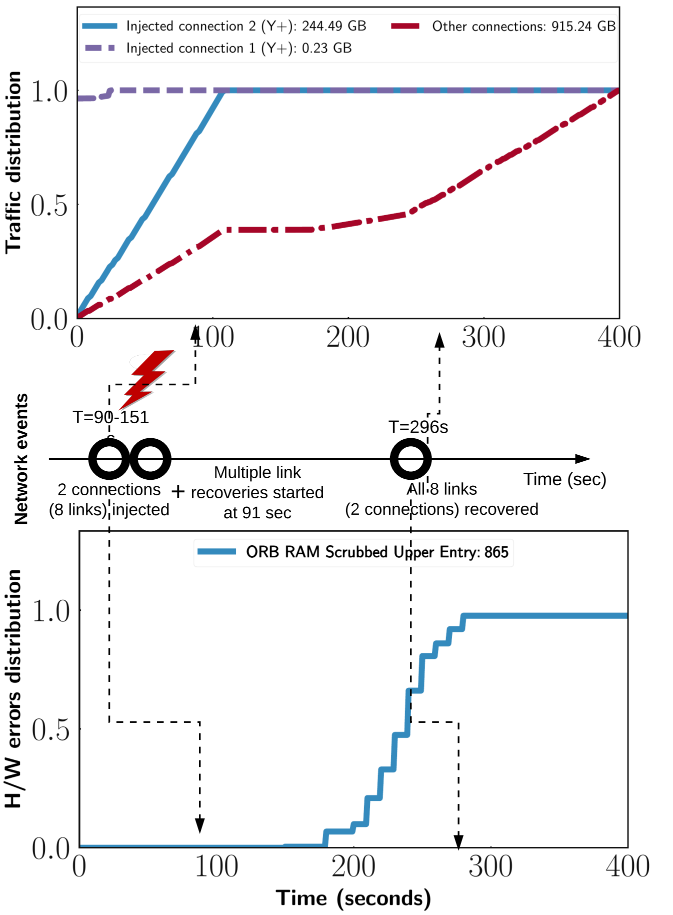

Fig. 7: Double connection injection (8 links in total) with successful recovery. *Top*: Cumulative distribution of network traffic measured in the time interval of the experiment. *Middle*: Time series of network events (enlarged). *Bottom*: Cumulative distribution of anomalous hardware error logs during the experiment.

not critical if contained in a small set of routers, but they can indicate a more severe situation if generated across the entire network. An attempt to do a warm swap on the links (after 24 minutes) in an attempt to restore the system to a working state, was also reported as successful in the logs. Nevertheless, the huge number of hardware errors persisted. Eventually, additional hardware errors were observed ("*SSID Stale on Response*" and "*ORB Request with No Entry*"). The situation above is typically considered a *deadlock* as packets are stuck in the output buffer of the Gemini router ASICs (which in turn leads to severe congestion in the network). We hypothesize that a corruption of the routing tables in one of the routers could be the cause of this deadlock. In this scenario, the routing tables would indicate incorrect connection paths that are not consistent with the real state of the network and of the working links. This type of problem would not be detected by the health checker daemons in the system, and the SMW would assume the system state to be healthy. This kind of critical scenario is typically assessed by a human operator, who must manually analyze the system error logs. The system needs to be rebooted after a deadlock, leading to unsuccessful termination of all the applications running in the system.

A network deadlock in the Gemini network can be detected by analysis of hardware error logs and SMW logs. A temporally increasing number of Gemini router ASICs reporting "*ORB RAM Scrubbed Entry*" along with "*LB Lack of Forward Progress*" errors is indicative of a critical state in the network. Automated analysis could enable early detection of the *deadlock* network state.

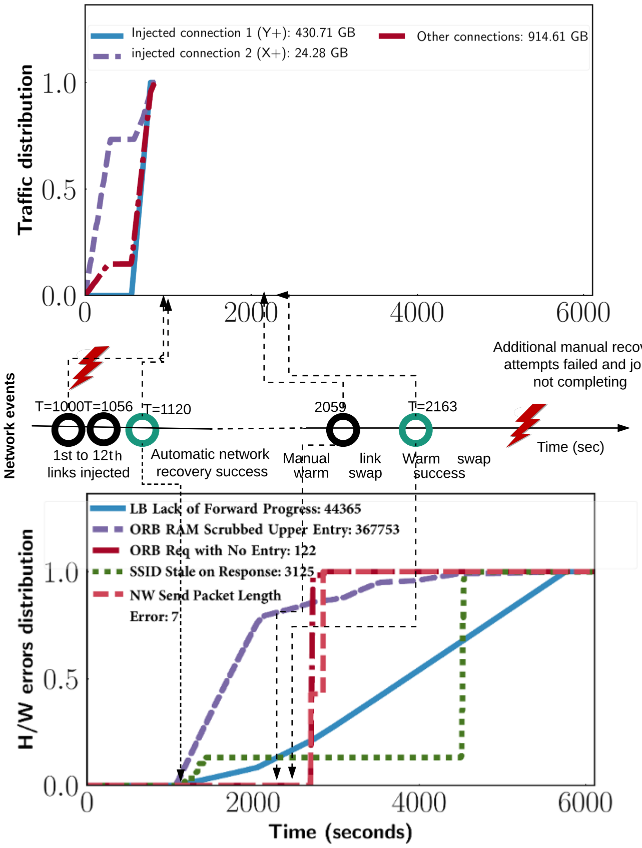

Fig. 8: Double connection injection and deadlock. *Top*: Cumulative distribution of network traffic measured in the time interval of the experiment. After the injection, the failure stalled the LDMS. *Middle*: Time series of network events (enlarged). *Bottom*: cumulative distribution of anomalous hardware error logs during the experiment.

E. Multiple Concurrent Faults

This scenario shows the effects of a concurrent multiple *link* failure scenario, in a *BF* (blade failure) campaign. In this case, a fault targeting a blade led to permanent failure of the blade, causing two Gemini router ASICs to fail, and hence multiple communication *links* to become unavailable instantaneously. This, in turn, caused failure of additional *links* (connected to six different Gemini router ASICs) on the other ends of the physical *links* of the failed blade. Software daemons on the *Blade Controller* (BC) (one for each blade) detect failure of links and of the blade associated with the controller itself. Therefore, failures of *links* on different blades are detected and reported by the associated BCs asynchronously. In total, 72 *links* were reported in two seconds, as two groups of 36 failed *links* each. The recovery completed successfully, but the network was unavailable for 68 seconds.

Blade failure recovery times might be improved by handling the failure of both ends of a *connection* when either end fails as failure of *links* at one end implies unavailability at the other end.

VIII. Resilience Recommendations

Our fault injection experiments have enabled us to better understand the details of failure scenarios we have seen in large-scale production Cray systems, such as Blue Waters. In the case of the Gemini High Speed Network (HSN), failures and automated recovery mechanisms largely operated as expected [13]. However, the more complex scenarios described in Section IV resulted in large variations in time to impact, time for recovery, and success of recovery. Duration of some scenarios was long enough (e.g., tens of minutes) that additional mechanisms could be developed to enable improved resilience and/or reduce adverse impact.

Potentially the HSS, which is unaffected by failures on the HSN, could be used for communication or triggering of higher-level mechanisms in addition to transmission of low-level fault information to the SMW and recovery directives from the SMW. In addition, as the use of node-local non volatile storage becomes more common, the options for saving state without requiring network access should be explored.

In general isolated events (e.g., link and node failures) were well handled and of short duration. Notification of job-killing events, such as node failures, appear in the job output. For these cases we currently see no obvious instrumentation that could be used to provide advance warning to the system or applications of impending failures.

Cray's aggregation of faults that co-occur within a system-defined time window (e.g., multiple HSN link failures) can ensure orderly response to cascading faults, faults that derive from the same root cause but are slightly displaced in time, or faults that are unrelated but temporally close. In the best case scenarios, faults and attempted recoveries occur over a number of minutes. However, fallout from the recovery mechanisms can still result in substantial job run-time impact, even when full recovery is ultimately successful (e.g., as in Figure 5). This provides a potential window of application resilience opportunity, as these time scales are long enough to enable sending of actionable notifications to affected applications and/or system software components to initiate defensive mechanisms in preparation for a likely failure in recovery. For example, post-recovery-process system conditions resulting in "*ORB RAM Scrubbed*" messages were clearly related to extended impact duration. The conditions that trigger such events could be used to pro-actively notify applications and system software. For example, task-based applications could speculatively farm off a duplicate task, rather than wait for a task response to a heartbeat. Higher-priority indicators could be used to mark prolonged events within certain time windows.

Higher-level aggregation and notification of system-wide events would also be of particular benefit. More "*ORB RAM scrubbed*" messages occurred over a greater spatial extent for the longer-duration and ultimately unsuccessful recovery attempts (e.g., Figure 8). Higher-priority notification of such cases could enable more accurate indication of potentially severe faults. For example, the jobs in the deadlock case were not killed, so the user was not informed that their application progress had ceased; typically only system administrators have access to the error logs where this information is available. In addition, the lack of information aggregation made it harder to assess the system state in post-processing analysis. Faster diagnosis would have been made if the notification and logging for the successful implementation of the link were more integrated with the notification and logging for the resulting continuous hardware errors.

Finally, some system-wide events, such as network *throttles* and *quiesces*, occur as a result of defensive actions taken because of faults on unrelated nodes. Wider notifications could be used to inform run-time mechanisms about nodes that may be indirectly affected.

Some of the items above require only increased notifications from already existing instrumentation and system-logging mechanisms. In addition, had the *link* state counter been accurately identifying degraded *connection* capacity, this instrumentation could also have been used to trigger notifications based on multiple sequential reductions of the same directional *connection's* capacity or on a global assessment of the overall network state. The faults injected were relatively instantaneously detected, and thus additional instrumentation would not necessarily have helped.

IX. Aries

The next phase of this work involves applying our FI approach to the Cray XC platform which employs the Cray Aries router ASIC (an evolution of the Gemini router ASIC). The Aries HSN resiliency mechanisms [20] have some similarities to those of the Gemini. Cray has stated that many of the FI mechanisms used here for the Gemini router ASIC will also work for the Aries router ASIC. Thus, the injector plugin components in HPCArrow for the XE can be easily leveraged for performing FI experiments on the XC.

Our approach requires the ability to discover and attribute log messages relating to events of interest. As was the case for the Gemini router ASIC, steps in the recovery of a single failure of an Aries router ASIC are well described in Cray's documentation [20]. However, the steps for handling more complex scenarios, such as additional failures during recovery, are not as well-described. Determining the relevant messages for complex, infrequent failures requires searching large numbers of logs for possibly unknown messages. Search of the BlueWaters logs provided the initial basis for determining the expressions and sequences in LogDiver for Gemini [3]. For the Aries, five months of logs for the ACES Trinity Phase 2 system (approximately 9000 nodes), including pre-production time, contain over 4.5 billion log lines (not including the job related data), which would be time-consuming to search.

In order to aid us in this task, we utilize our Baler [11] tool for log analysis. Baler converts log messages into deterministic *patterns* of interest without requiring any apriori knowledge of the messages. A dictionary is used to convert messages into a pattern consisting of dictionary words, with non-dictionary words resulting in variables. For example, lines such as `found_critical_aries_error: handling failed PT c11-8c1s3a0n0 (blade c11-8c1s3)` become pattern `found_critical_aries_error: handling failed • •-• (blade •-•)`. Baler can then aggregate similar patterns into a single higher level *meta-pattern*. This process can result in a substantial reduction in the patterns to search. Optional use of the dictionary to include domain-relevant words, such as *ORB*, and weight words of significance, such as *failed* can further reduce the search space. We have used Baler to reduce the Trinity Phase 2 log data set down to 1350 meta-patterns. (More details can be found in [21]). Examination of these patterns and surrounding events can then help us identify new messages and sequences for inclusion in LogDiver to address the Aries.

In general, many messages and sequences are similar, for example, some events involved in the computations of new routes and handling of additional failures during recovery. However, there are some failure messages from the XC platform that we have not seen in 4 years of Blue Waters data, for example "*Warm swap aborted due to hardware failure during link initialization*", in addition to the similar message seen in this work but for "*failure during route computation*". There also appear to be differences in some of the handling and reporting of ORB events. Of particular interest, given our observations in this work, is that the form and location of the messages pertaining to the ORB scrubbing have changed. In general, we expect that advances in the Aries router ASIC, particularly because of the flexibility in the routing, will result in less adverse impact in the handling of single and multiple failures. We will be comparing the duration of recoveries and the use of the ORB scrubbing messages as useful resiliency notification and triggering for the Aries router ASIC.

In addition, the expected handling of applications in the face of certain critical errors, such as those seen in Section VII-C1 is more well-defined in the Aries router ASIC documentation [20] than in the public Gemini router ASIC documentation. We will be comparing application behaviors between the XE and XC platforms under production conditions resulting in observed critical errors (e.g., `critical_aries_error`) that we can induce using our FI tools.

Finally, we have implemented Aries performance counter data collection using LDMS [22]. In contrast to Gemini, Aries provides more counters, which will enable more detailed understanding of the impacts in various parts of the routers and NICs.

X. Related Work

Fault injection: Fault injection is a technique used to study the system behavior by systematically exposing the system to faults. Such a technique allows system designers and developers (1) to assess the correctness of fault-handling mechanisms; (2) to understand fault-to-failure propagation paths; and (3) to assess system vulnerability. Fault injection techniques can be categorized as hardware-based or software-based [23]. Hardware-based fault injection techniques normally require specialized hardware support for the target systems. Software-implemented fault injection (SWIFI [4]) techniques typically enable emulation of hardware faults using software techniques via perturbation of code or data. SWIFI techniques are easy to deploy and can be highly tuned to emulate complex fault scenarios. For this reason, we built *HPCArrow*, a SWIFI-based HPC interconnection network fault injection tool that (1) hard injects faults, (2) monitors the system at appropriate levels to enable understanding of the effects, and (3) restores the system health. The design of *HPCArrow* is based on the NFTAPE [24] fault injector design. *HPCArrow* is architecture-independent; architecture-specific injections, such as the Cray XE injections studied here, are supported through different instantiations of the Fault Injector component.

Fault injection in HPC systems: In the past, fault injection experiments in HPC systems have mostly focused on injecting faults in the memory [25], [24], processor [26], [24] and application run-times/processes [27], [28] of the system. The chances that such faults will propagate to the other nodes are much smaller than for other faults and failures in the network. There has been a dearth of studies investigating the effects of network-related faults on applications and systems. In [29], [30] the authors only investigated the effects of faults in a message-passing interface that were caused by either network-related failures or corruption in the memory/process. These studies characterized application resiliency to message corruption and message loss. However, recovery from one or more link failures can take several minutes, during which time the system and applications are in a vulnerable stage. In this work, therefore, we focused on understanding the susceptibility of recovery mechanisms of HPC networks to faults and failures. In our study, unlike other fault injection studies, faults were injected on a real petaflop-scale system consisting of nine thousands nodes running an HPC workload. This allowed us to understand the fault-to-failure path of network faults/failures on the system and applications. To the best of our knowledge, the previous largest fault injection study was conducted on a Teraflops supercomputer [31] that injected faults on Intel processors at the pin level.

XI. Conclusion

In this work we presented a fault injection campaign on Cielo, a Cray XE petascale HPC system with 8,944 nodes and a Gemini topology, jointly developed by Los Alamos National Laboratory (LANL) and Sandia National Laboratories (SNL) under the Advanced Computing at Extreme Scale (ACES) partnership. We had a unique opportunity to execute the experiments after Cielo's end of production, but before its complete retirement. To execute the experiments, we developed *HPCArrow*, a software fault-injection tool capable of recreating failure scenarios on nodes, links, and blades, as well as more severe combinations of failures, like sequential link failures and failures of entire directional connections. We proved *HPCArrow* effectively interrupted traffic on the injected links and on failed nodes and blades running in the system. The traffic profiles and log events observed in Cielo correspond to those observed for Blue Waters at UIUC. The experiments have generated anomalies in the hardware error logs that can be used as indicators for critical conditions in the network. Further, the relative times and locations of injections have generated critical scenarios with longer recoveries or unrecoverable problems (deadlock). For some critical scenarios, recovery durations were long enough to provide the opportunity to pro-actively notify applications and system software to initiate defensive mechanisms in preparation for a likely failure in recovery. Finally, similarities in logs and recovery operations for Cray XC Aries and Cray XE Gemini suggest that it will be possible to recreate similar failure scenarios in the newer Cray network. This will allow us to compare fault-to-failure paths and failure handling capabilities in the two systems.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number 2015-02674. This work is partially supported by NSF CNS 13-14891, an IBM faculty award, and an unrestricted gift from Infosys Ltd. This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Application. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. We thank Mark Dalton (Cray), Forest Godfrey (Cray), and Gregory Bauer (NCSA) for having many insightful conversations.

References

[1] M. Snir *et al.*, "Addressing failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[2] X. Yang, Z. Wang, J. Xue, and Y. Zhou, "The reliability wall for exascale supercomputing," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 767–779, 2012.

[3] S. Jha, V. Formicola, Z. Kalbarczyk, C. Di Martino, W. T. Kramer, and R. K. Iyer, "Analysis of gemini interconnect recovery mechanisms: Methods and observations," *Cray User Group*, pp. 8–12, 2016.

[4] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using FIAT," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, 1990.

[5] C. Di Martino, F. Baccanico, J. Fullop, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *Proc. of 44th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, 2014.

[6] C. D. Martino, S. Jha, W. Kramer, Z. Kalbarczyk, and R. K. Iyer, "Logdiver: a tool for measuring resilience of extreme-scale systems and applications," in *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*. ACM, 2015, pp. 11–18.

[7] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen, "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems," *The Journal of Supercomputing*, vol. 65, no. 3, pp. 1302–1326, 2013.

[8] U.S. Department of Energy Office of Science, "Resilience for extreme scale supercomputing systems," DOE National Laboratory Announcement Number LAB 14-1059, 2014.

[9] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[10] L. Bautista-Gomez, A. Gainaru, S. Perarnau, D. Tiwari, S. Gupta, C. Engelmann, F. Cappello, and M. Snir, "Reducing waste in extreme scale systems through introspective analysis," in *Proc. of IEEE conference in Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 212–221.

[11] N. Taerat, J. Brandt, A. Gentile, M. Wong, and C. Leangsuksun, "Baler: deterministic, lossless log message clustering tool," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 285, 2011.

[12] C. Lueninghoener, D. Grunau, T. Harrington, K. Kelly, and Q. Snead, "Bringing up cielo: experiences with a cray xe6 system," in *Proceedings of the 25th international conference on Large Installation System Administration (LISA)*, 2011.

[13] Cray, "Gemini network resiliency guide," http://docs.cray.com/books/S-0032-E/.

[14] "Getting started with intel mpi benchmarks 2017, intel software," Intel Corporation, September 2016. [Online]. Available: https://software.intel.com/en-us/articles/intel-mpi-benchmarks

[15] Cray Inc., "Using and Configuring System Environment Data Collections (SEDC)," Cray Doc S-2491-7001, 2012.

[16] ——, "Cray Linux Environment (CLE) 4.0 Software Release," Cray Doc S-2425-40, 2010.

[17] ——, "Using the Cray Gemini Hardware Counters," Cray Doc S-0025-10, 2010.

[18] M. Showerman, J. Enos, J. Fullop, P. Cassella, N. Naksinehaboon, N. Taerat, T. Tucker, J. Brandt, A. Gentile, and B. Allan, "Large Scale System Monitoring and Analysis on Blue Waters using OVIS," in *Proc. Cray User's Group*, 2014.

[19] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, "Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *Proc. Int'l Conf. for High Performance Storage, Networking, and Analysis (SC)*, 2014.

[20] Cray, "Aries network resiliency guide," http://docs.cray.com/PDF/XC_Series_AriesNetwork_Resiliency_Guide_CLE60UP02_S-0014.pdf.

[21] A. DeConinck *et al.*, "Runtime collection and analysis of system metrics for production monitoring of trinity phase ii," *Proc. Cray User's Group*, 2017.

[22] J. Brandt, E. Froese, A. Gentile, L. Kaplan, B. Allan, and E. Walsh, "Network performance counter monitoring and analysis on the cray XC platform" in *Proc. Cray User's Group*, 2016.

[23] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[24] 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 *IEEE Proc. in Computer Performance and Dependability Symposium*. IEEE, 2000, pp. 91–100.

[25] T. Naughton, W. Bland, G. Vallee, C. Engelmann, and S. L. Scott, "Fault injection framework for system resilience evaluation: Fake faults for finding future failures," in *Proceedings of the 2009 Workshop on Resiliency in High Performance*. ACM, 2009, pp. 23–28.

[26] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 221–230.

[27] D. M. Blough and P. Liu, "Fimd-mpi: a tool for injecting faults into mpi application," in *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE, 2000, pp. 241–247.

[28] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Symplfied: Symbolic program-level fault injection and error detection framework," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 472–481.

[29] T. Naughton, C. Engelmann, G. Vallée, and S. Böhm, "Supporting the development of resilient message passing applications using simulation," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*. IEEE, 2014, pp. 271–278.

[30] K. Feng, M. G. Venkata, D. Li, and X.-H. Sun, "Fast fault injection and sensitivity analysis for collective communications," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2015, pp. 148–157.

[31] C. Constantinescu, "Teraflops supercomputer: Architecture and validation of the fault tolerance mechanisms," *IEEE Transactions on Computers*, vol. 49, no. 9, pp. 886–894, 2000.