RESILIENCY OF HIGH-PERFORMANCE COMPUTING SYSTEMS: A FAULT-INJECTION-BASED CHARACTERIZATION OF THE HIGH-SPEED NETWORK IN THE BLUE WATERS TESTBED

BY

SHARON S. TANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Research Professor Zbigniew T. Kalbarczyk

ABSTRACT

Supercomputers have played an essential role in the progress of science and engineering research. As the high-performance computing (HPC) community moves towards the next generation of HPC computing, it faces several challenges, one of which is reliability of HPC systems. Error rates are expected to significantly increase on exascale systems to the point where traditional application-level checkpointing may no longer be a viable fault tolerance mechanism. This poses serious ramifications for a system's ability to guarantee reliability and availability of its resources. It is becoming increasingly important to understand fault-to-failure propagation and to identify key areas of instrumentation in HPC systems for avoidance, detection, diagnosis, mitigation, and recovery of faults.

This thesis presents a software-implemented, prototype-based fault injection tool called HPCArrow and a fault injection methodology as a means to investigate and evaluate HPC application and system resiliency. We demonstrate HPCArrow's capabilities through four fault injection campaigns on a Cray XE/XK hybrid testbed, covering single injections, time-varying or delayed injections, and injections during recovery. These injections emulate failures on network and compute components. The results of these campaigns provide insight into application-level and system-level resiliencies. Across various HPC application frameworks, there are notable deficiencies in fault tolerance. Our experiments also revealed a failure phenomenon that was previously unobserved in field data: application hangs, in which forward progress is not made, but jobs are not terminated until the maximum allowed time has elapsed. At the system level, failover procedures prove highly robust on small-scale systems, able to handle both single and multiple faults in the network.

To my mother and father, for their love and support.

ACKNOWLEDGMENTS

The work presented here would not have been possible without the help and support from my adviser, mentors, fellow colleagues, and friends here at the University of Illinois at Urbana-Champaign.

First and foremost, I thank my adviser, Zbigniew Kalbarczyk, for his constant guidance and unending patience throughout my graduate studies. I am grateful for all the opportunities and advice he has given me to advance myself as a teacher and as a researcher. I also acknowledge my fellow DEPEND colleagues Lavin Devnani, who was there in the trenches with me building the foundations of this work, and Saurabh Jha for his technical guidance and insights in this world of supercomputers.

I also thank my colleagues at the National Center for Supercomputing Applications for providing assistance and resources on Blue Waters: Mike Showerman, Gregory Bauer, Jing Li, and Bill Kramer. Special thanks also go to my colleagues at Sandia National Laboratories, Jim Brandt and Ann Gentile, for their technical insights and resources.

I acknowledge my support group for helping me in the non-technical spaces of this work. This includes my dear friends on the ECE 391 staff holding the line in 3026 ECEB; the ECE 199 staff; my Illinois family Janet Sanoica and Gillian Smith; and most importantly my family. To my mom who always answered my phone calls, to my dad who always picked me up from the airport, and to my brother who always called me: Thank you for everything.

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 Applications. This report is based upon work supported by the Department of Energy, Office of Science, Office of Basic Energy Sciences, under Award Number DOE 2015-02674. This thesis

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivation

High-performance computing (HPC) has played an essential role in the progress of many scientific and engineering fields that require massive amounts of computations. These areas cover a wide spectrum from quantum physics to molecular dynamics, brain modeling and simulations, weather and climate research, and security. The machines used for these large amounts of computational work are large-scale, HPC systems, more commonly known as supercomputers. Current HPC systems are capable of petascale computing performance, meaning they are able to perform one or more petaflops.[1] However, there remain many scientific problems that require computational resources beyond what HPC systems currently offer. The United States Department of Energy and other agencies across the world have begun conducting development of the next generation of supercomputers, called extreme-scale or exascale computing systems [1].

However, as the moonshot of the HPC community, exascale computing faces several challenges. One challenge is reliability of HPC systems and their tolerance against inevitable faults. It is expected that exascale systems will incur excessively higher error rates [1], which has serious ramifications for a system's ability to guarantee reliability and availability of its resources. Too many failures may hinder progress of computations or affect accuracy of results because mean time between failures (MTBF) will decrease to the point where traditional application-level checkpointing may no longer be a viable fault tolerance mechanism. An increasing number of component failures may also lead to other more complex scenarios of system failures. Thus it becomes increasingly important for HPC vendors and facilities to understand fault-to-

[1]One petaflop is defined as one quadrillion floating point operations per second (flops).

failure scenarios and to identify key areas of instrumentation for avoidance, detection, diagnosis, mitigation, and recovery of faults. It may no longer be enough to solely analyze field data from live production systems. Several months of field data may not fully capture all known or unknown failure scenarios. This reliability challenge underscores a critical need for other effective techniques to investigate and evaluate HPC system resiliency. One such technique is fault injection.

This thesis presents a software-implemented, prototype-based fault injection tool called HPCArrow[2] and a fault injection methodology as a means to evaluate HPC system resiliency and to provide insight toward improving HPC fault tolerance. We target network and compute components as our fault models. We demonstrate HPCArrow's capabilities through four fault injection campaigns executed on the Blue Waters[3] small-scale 96-node Cray XE/XK hybrid testbed system, covering single injections, time-varying or delayed injections, and both single and multiple injections during recovery. The results of these campaigns provide insight into application-level and system-level resiliencies.

1.2 Fault Injection

Fault injection is a technique widely employed to study system resiliency and reliability through deliberate and methodical introduction of faults into the system. This method offers a couple advantages. (1) It provides control over fault conditions, such as timing, location, system state, etc., and (2) it can be automated to perform experiments in a repeated, reproducible manner. Having such control and automation allows system designers to validate error handling and fault-tolerant mechanisms, determine coverage of error detection and recovery mechanisms, reproduce failure scenarios observed in field data, trace the fault-to-failure propagation paths, and identify reliability vulnerabilities and deficiencies [2]. Insight gleaned from fault injections can then

[2]HPCArrow was developed by Lavin Devnani and Sharon Tang (author). HPCArrow is based on past work by Fei Deng and is part of the Holistic, Measurement-Driven Resilience (HMDR) project, a collaboration between the University of Illinois at Urbana-Champaign (UIUC); Sandia (SNL), Los Alamos (LANL), and Lawrence Berkeley (LBNL) National Laboratories; and Cray Inc.

[3]Blue Waters is a petascale 26868-node Cray XE/XK hybrid system at the National Center for Supercomputing Applications (NCSA) located in Urbana, Illinois.

be utilized to identify optimal places where the system can be instrumented for detection of faults and mitigation of their effects.

Fault injection techniques can be hardware-based or software-based [2]. Hardware-based injections typically require specialized and expensive hardware support. On the other hand, software-implemented fault injection (SWIFI) techniques offer a degree of control in terms of injection location and time and are less expensive to deploy. Additionally, on HPC systems, most hardware functionality is visible through software, allowing software-based fault injection techniques to emulate faults at various levels of the system, including hardware. These advantages are the reasons for building a SWIFI-based HPC fault injection tool in this work, allowing faults at the hardware and network levels to be emulated.

1.3 Related Work

There has always been a need for understanding failure scenarios and the ramifications of errors on HPC systems. Much of this understanding has come from measurement-based analysis of field data collected from live production systems. In [3], over 5 million HPC application runs on Blue Waters were analyzed to understand the impact of system errors and failures on applications and assess application fault tolerance. In [4], data and manual reports from Blue Waters were used to investigate and characterize single-node failures, system-level failovers, and system-wide outages. The resiliency of the Gemini interconnect against faults, errors, and congestion and the impact of recovery procedures were assessed using Blue Waters data in [5] and [6] and using Titan[4] data in [7].

However, all of these studies, utilizing field data, are thus constrained to naturally occurring and known failure events. Basing understanding of fault-to-failure paths solely on production data is limiting, especially when multiple errors and failures occur simultaneously. Information about fault locations, health of the system, and workload conditions can be incomplete in field data, whereas our fault injection method can provide control over the fault conditions and environment to bring clarity to the full fault-to-

[4]Titan is a petascale 18688-node Cray XK system at Oak Ridge National Laboratory located in Oak Ridge, Tennessee.

failure paths. Furthermore, this production data must be collected over a long period of time because specific instances of errors and failures may occur infrequently. For example, Titan data was collected over the course of one year to examine Gemini resiliency [7]. Fault injection can repeatedly recreate the fault conditions and produce multiple instances of the same failure scenario. Additionally, production data of petascale systems may not fully reflect the conditions and scenarios that will arise on exascale systems. Fault injection is useful not only for recreating failure scenarios observed in production data, but also for simulating new scenarios that may not have been encountered so far in current systems.

Other studies of HPC systems have analyzed resiliency behaviors through methods such as stress testing. A microbenchmark application was developed in [8] to stress the network and uncover performance problems on Titan's Gemini interconnect. In [9], standard benchmarks and scientific applications were utilized to understand application performance and runtime consistency. Our fault injector incorporates a similar idea in running a set of benchmarks and scientific applications to emulate a workload environment that is similar to real workloads on production systems. This use of real applications also provides a perspective on the fault tolerance of HPC application framework and helps to uncover any resiliency vulnerabilities.

Previous works have also featured software-based fault injection studies on HPC systems, but these faults model low-level errors and target HPC applications. In [10], a hierarchical injection methodology is adopted, focusing on assembly-level and register transfer level (RTL) or gate-level faults injected into HPC applications. FlipIt is a LLVM-based compiler-level fault injection tool, focused on memory bit-flip errors that target HPC applications [11]. It is tested on Blue Waters [11]. F-SEFI is a fault injector that focuses on injection of soft errors targeting instructions of HPC applications and their subroutines [12]. F-SEFI is tested on the QEMU virtual machine (VM) and its hypervisor [12]. A machine learning framework is used in [13] to diagnose node-level anomalies while programs were used to stress a single node of a multi-node HPC application. These injections were performed on VMs as well as a Cray XC testbed. FINJ presented in [14] is a high-level fault injection tool for HPC systems that targets a node via tasks, which can be a benchmark or a fault-triggering program. This design allows FINJ to be integrated with any low-level fault injection framework that can be triggered by an executable program or a shell script [14]. FINJ is tested on a single-node prototype system [14].

From among these HPC fault injection studies, there is a noticeable lack of fault injections at the network level. Our fault injector, on the other hand, not only generates faults at the node level, but also targets components at the network level. While we do not look at fine-grained node-level aspects such as memory congestion, our tool can be extended to include these programs designed to stress node resources or to target application instructions. We also evaluate application-level fault tolerance, using multiple HPC benchmarks or applications, and we assess resiliency at the network and system levels by analyzing network traffic data, system logs, and recovery behaviors. These other fault injection studies also do not all inject on full production HPC systems; some inject on prototype systems or VMs. The injection experiments presented in this thesis are all executed on a small-scale Cray XE/XK hybrid testbed, similar to the one in [13]. Details of the experiment environment are discussed later.

The work and results presented in this thesis are based on previous fault injection experiments performed on a much larger 8944-node system called Cielo.[5] Those experiments were able to produce critical system failures due to faults at the network level [15]. The work presented in this thesis extends Cielo's fault injection experiments to Blue Waters with the aim to recreate similar failure scenarios. To the best of our knowledge, this work combined with the work performed on Cielo is the largest fault injection study conducted on HPC systems.

1.4 Contributions

The main goal of this research study is to evaluate the resiliency of HPC systems in the presence of faults and to improve understanding of fault-to-failure propagation and failure impact on the performances and behaviors of HPC applications and systems. To that end, the key contributions of this thesis include:

- **Fault injection toolkit HPCArrow for large-scale HPC sys-**

[5]Cielo was the (now retired) Cray XE petascale system developed jointly by LANL and SNL under the Advanced Computing at Extreme Scale (ACES) partnership.

tems. We developed a fault injector that is system-independent, as long as Python, a scripting language, is supported and as long as locations targeted for fault injection are accessible to software. HPCArrow provides a controlled environment to inject one or more faults into the system by executing system-specific commands. These commands mimic failures of network and compute components: network links, directional connections, compute nodes, and compute blades. HPCArrow also provides the ability to conduct fault injection experiments in a repeatable, reproducible manner by being able to control timing and location of faults. The injector is also able to restore failed components back into service and return the system to a fault-free state. In addition to simply injecting faults at user-specified times and locations, HPCArrow can monitor network events in real time and conduct multiple injections during an automatic recovery process initiated by an earlier fault injection. Injections are targeted at a user-specified recovery stage. This injector is successfully used to perform four different types of fault injection campaigns on a Cray XE/XK hybrid testbed with a Gemini interconnect and can successfully provide insight into fault-to-failure paths. It has also been verified to work on several Cray XC systems that use the Aries interconnect, and fault injection experiments are currently underway with SNL.

- **A set of HPC applications configured and compiled to study their susceptibility to network-level faults.** A total of nine unique applications, spanning three HPC application frameworks, were compiled and configured to run as real workloads on the system. These frameworks include Charm++, Message Passing Interface (MPI), and Partitioned Global Address Space (PGAS), all of which offer a wide range of fault tolerance and other communication features. From the results of the fault injection campaigns, we observed a variety of application behaviors such as crashes, hangs, and no impact. We also observed that certain frameworks are more susceptible to network-level faults than others with Charm++ being the most susceptible and MPI being the least.

- **A set of fault injection experiments to demonstrate the system and application behaviors in the presence of faults.** Across almost 300 fault injection experiments, we observed application crashes and hangs caused by a simple failure of the network, despite successful failover recovery procedures. These simple failures include a single link fault or a single connection (multiple links) fault. Originally, we expected all applications would be able to tolerate link and connection faults due to there being other pathways to take once failover recovery recomputes and installs new routes. Unexpectedly, however, many applications were not fault tolerant against even these simple failures.

- **Identification of an application failure behavior previously unseen in field data.** Hung applications was one application-level behavior observed as response to network-level faults. This kind of behavior due to simple faults had not been observed before in field data, but our fault injector and methodology uncovered this somewhat alarming phenomenon. Hung applications are a waste of system resources and detract from availability and performance. Network and system logs are insufficient to differentiate between application crashes and hangs. Application logs and job scheduler information are the only indicators of an application hang, but this can only be diagnosed and determined after the fact. Real-time detection and handling of such behavior should become a priority in future work.

- **Understanding and assessment of HPC systems failover responses to network-level faults.** Through HPCArrow, we are able to inject single or multiple faults during an ongoing recovery procedure triggered by a previous fault. This is achieved through real-time monitoring of network events communicated in constantly updated and rotated network logs. This allowed us to conduct a fault injection campaign to observe and assess the failover mechanisms that respond to failures in the network. These experiments showed a deeply robust failover procedure that can handle all faults injected in our experiments. Since we could not reproduce the deadlocks and system-wide outages caused by fault injections on Cielo, we suspect that the window of recovery on our small-scale testbed is too short to allow any propagation of faults throughout the network and system. Failovers may also have a window of vulnerability that is either nonexistent or too short to be targeted by faults on small-scale systems. However, we do observe that simple network failures during recovery yield longer recovery durations. Faults during failover cause the recovery process either to automatically restart or to start a second recovery instance, which in turn causes the recovery time to increase. An increased time in failover recovery means an increase in the probability of the occurrence of additional failures, which may lead to catastrophic failures in the system [5].

- **Foundational work for a comparison study on two network fabrics and their resiliency behaviors in the presence of faults.** The Aries Dragonfly interconnect is a more modern network fabric than the Gemini interconnect. However, Aries still shares similar resiliency mechanisms with Gemini. The work and results in this study have laid the foundation for this next step of comparison between Aries and Gemini. HPCArrow has already been extended to work on Cray XC systems utilizing the Aries interconnect. Applications have also been selected, compiled, and configured to run on Cray XC systems. Despite some architectural and network differences, the same fault injection methodology used in this study on Gemini also works on Aries.

1.5 Challenges

Throughout the course of this study, we encountered many very involved challenges that required mostly trial-and-error type approaches to solve. These include:

- **Compiling and configuring HPC applications to be scaled down and to run for a desirable amount of time.** While this work involved nine unique applications, there is a much longer list of applications that we tried and failed to get running on the Cray XE/XK testbed. This testbed is a small-scale system of 96 nodes,[6] which is much smaller than a large-scale production system like Blue Waters. We suspect that many scientific applications are inherently too large to run on small-scale systems. The applications we did manage to get

[6]For this research, we were limited to 64 nodes (top two chassis) on this one-cabinet system as requested by NCSA.

working still required a great deal of configuration time to scale to our various workload needs and to adjust to our experiment runtime needs. Adjusting parameters is very difficult across nine applications because some level of understanding of each application is necessary. Because each HPC system may be outfitted with different software versions and hardware components, porting applications from one system to another is also nontrivial as it requires re-compilation and re-configuration. This step of the experiment setup process is the most time consuming. We are grateful for the help and insights of NCSA application specialists on a couple of these applications.

- **Limitations of development and of running statistically significant number of experiments.** Due to the need to have controllable environments and repeatable experiments, the injector was also equipped with the ability to restore the system back to a fault-free state. This restoration is the warm swap process, which can take up to 10 minutes, depending on the injected component. Given that the fault injection occurs around five minutes into an application's run time, we decide to limit an experiment's run time to be around 30 minutes. However, since all applications and their parameters cannot be precisely adjusted in the same manner, some applications may take around 40 minutes. In this work, an experiment can be as short as 33 minutes. If an experiment produces a hung application, that application will not terminate until it hits the maximum amount of time, set by us to be two hours. This amount of time severely limits the number of experiments we can run. Additionally, two thirds of the testbed (top two chassis) was reserved for us on certain days of the week for a time period between 7pm and 7am as this was when other users were unlikely to use the testbed. Our access to the SMW was further limited due to requiring supervised root access, which extends our development and test cycle significantly.

- **Collecting the logs needed for analysis (network data, system logs, application logs, injection logs, experiment logs).** All the logs necessary for analysis are scattered all over the system. Application logs are stored on a user's local account; network data is protected behind privilege levels; and both system logs and injector logs sit on the SMW, which requires root access. We developed a log collector module in HPCArrow. Since our injector sits on the SMW, running in the background, it is designed to collect all logs and transfer them to our user accounts and modify privileges on a daily basis. This module significantly cut our experiment and analysis time as we no longer had to make daily trips to NCSA or wait on a system administrator to package and send the logs to us.

- **Small window of recovery that limits injections during recovery.** Because our Cray XE/XK testbed is a small-scale system, its failover recovery duration is very short. Most recovery stages are less than three seconds and there are delays due to recovery stage detection and time to fault injection. This means that not all recovery stages are good candidates to target for injection. Certain optimizations were made to HPCArrow to remove as much unnecessary overhead as possible, which ensured that recovery stages lasting about two seconds or more are still eligible targets for certain fault types.

1.6 Thesis Organization

The remainder of this thesis is organized as follows: In Chapter 2, we provide background information on the architecture and system software of Cray XE/XK systems with specific focus on Blue Waters, the Gemini interconnect, and the system's fault tolerance mechanisms. In Chapter 3, we introduce and detail our fault injection methodology, detailing fault models, benchmark applications run as workloads on the system, and the fault injector toolkit HPCArrow. In Chapter 4, we describe our experimental setup and present our four fault injection campaigns and their results. We conclude this work in Chapter 5 with a summary discussion of results and suggestions for future work.

CHAPTER 2

BACKGROUND

This chapter provides background information on the architecture and system software of Cray XE/XK petascale systems, using Blue Waters as the example. These machines are highly scalable supercomputers, able to deliver one or more petaflops to scientific and engineering computing applications. Blue Waters can deliver up to approximately 13.3 petaflops at peak speed [16]. This chapter also includes background information on the fault-tolerant features and mechanisms of Cray XE/XK systems, including fault detection and recovery mechanisms.

This study is based on past works that have analyzed data generated by Blue Waters [4],[6],[5] and fault injection experiments performed on Cielo [15]. This study contributes to that body of work by conducting fault injection experiments on the Blue Waters test and development system called JYC,[1] which is built and managed identically to Blue Waters. These heterogeneous petascale systems, running on the Cray Linux Environment (CLE) operating system, combine the best of AMD's multicore processors, NVIDIA's many-core graphics processing unit (GPU) accelerators, and Cray's Gemini interconnect.

2.1 System Architecture

A general Cray XE/XK system is organized into a hierarchy of cabinets, chassis, blades, nodes, and links, as shown in Figure 2.1. At the highest level are cabinets, physically arranged in rows and columns. Larger systems have more rows. Each cabinet typically has an L1 cabinet controller, blower fan, and power conversion electronics [5]. At the next level of granularity in the hierarchy, each cabinet contains three chassis (also called cages). A chassis

[1]JYC is named for Jacques-Yves Cousteau.

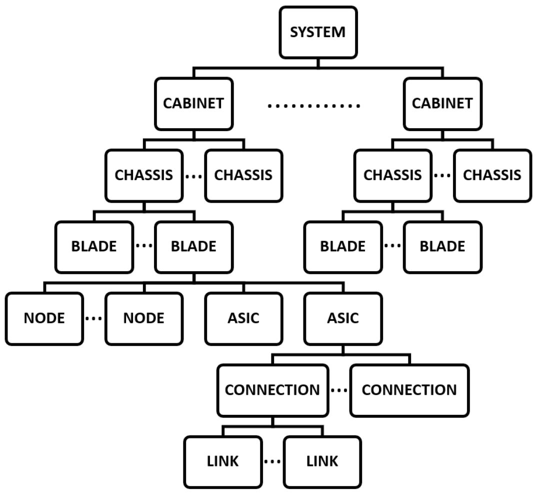


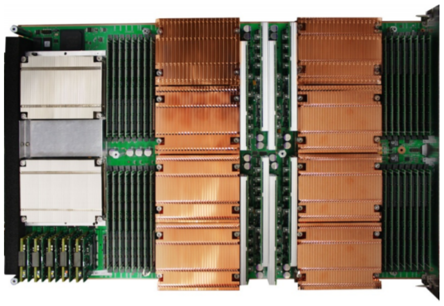
Figure 2.1: Gemini hierarchy of a typical Cray HPC system.

consists of eight blades. Each blade consists of a network mezzanine card that houses a pair of Gemini application-specific integrated circuits (ASIC), which act as network routers [5]. The Gemini ASICs are connected such that each blade provides a 1x4x1 network of nodes in the overall folded 3D torus topology [17]. Thus each Gemini ASIC is shared between two nodes. Additionally, each blade also packages four nodes, which can be XE, XK, or service nodes.

2.1.1 Nodes

- **Services nodes** are primarily used as boot nodes for system-wide reboots, as system database (SDB) nodes to collect event logs, as MOM nodes for scheduling jobs, as Lustre Filesystem Network (LNET) nodes to handle metadata and file I/O data for file system servers and clients, or as network gateway nodes to connect external networks through Infiniband QDR IB cards [4]. Service nodes on Blue Waters can be hosted on Cray XIO blades or Cray XE6 blades. Each XIO Service node consists of a 6-core AMD Opteron 2435 “Istanbul” with 16 gigabytes (GB) of DDR2 memory in 4 GB DIMMs protected by x4 Chipkill [4]. In this study, we were asked to not target service nodes.

- **XE nodes** are two-socket compute nodes. On Blue Waters, XE nodes are hosted on Cray XE6 blades with four nodes per blade. On each


(a) Cray XE6 Blade

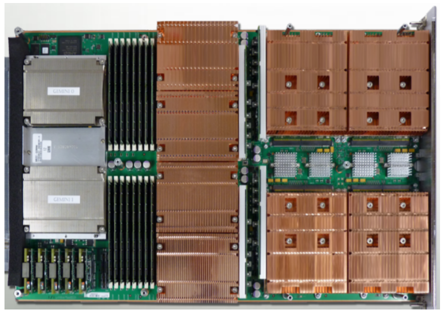

(b) Cray XK7 Blade

Figure 2.2: Cray XE6 and XK7 blade hardware.

node, both sockets are each occupied by a 16-core AMD Opteron 6276 "Interlagos" processor [16]; each Opteron holds 8 dual-core AMD Bulldozer modules, each of which has an 8x64 kilobyte (KB) L1 instruction cache, a 16x16 KB L1 data cache, an 8x2 megabyte (MB) L2 cache, and a 2x8 MB L3 cache [4]. With two 16-core Opteron 6276 processors, a node has a combined 64 GB of DDR3 RAM in 8 GB DIMMs, protected by x8 Chipkill [4].

- **XK nodes** are two-socket GPU nodes. On Blue Waters, these nodes are hosted on Cray XK7 blades with four nodes per blade. On each node, one socket is equipped with one 16-core AMD Opteron 6272 "Interlagos" processor with 32 GB of DDR3 RAM in 8 GB DIMMs [16]. Note that an XK node has half the RAM of an XE node. The other socket on the node contains an NVIDIA K20X "Kepler" accelerator [16]. This GPU is the main difference between XK and XE nodes. The accelerators house 2880 single-precision CUDA cores, 64 KB of L1 cache, 1536 KB of L2 cache, and 6 GB of DDR5 RAM memory that is protected with ECC [4].

2.1.2 Blades

All blade types (XIO, XE6, XK7) are powered by four Cray Verty voltage regulator modules, one for each node on the blade, and the power distribution unit in a cabinet. Every blade also holds an L0 controller that monitors the general health of the blades components. The two blades of interest in this study are the XE6 and XK7, as shown in Figure 2.2a and 2.2b, respectively.

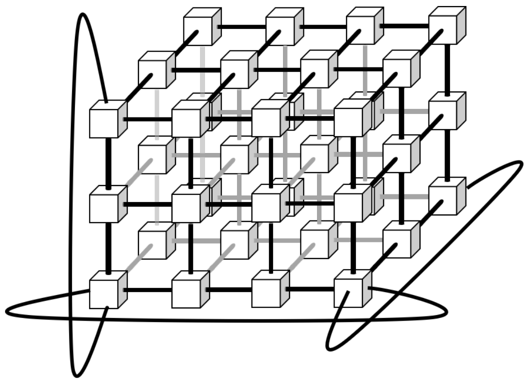


Figure 2.3: Cray Gemini interconnect with a folded 3D torus topology. Each cube represents a Gemini ASIC. The folded aspect is depicted by the loops (only three are drawn).

2.2 High-Speed Network

Communication on Cray systems occurs over the high-speed network (HSN). There are various vendor proprietary network designs such as Cray's Gemini and Aries or IBM's Blue Gene [5]. The work presented in this thesis focuses on the Cray Gemini interconnect and the Gemini ASIC, the basic building block of Blue Waters' HSN. The ASICs are arranged in a folded 3D torus topology [16], as shown in Figure 2.3. This folding minimizes the maximum cable length to connect all nodes in a single dimension [8]. The advantages of the Gemini interconnect include high performance on MPI applications and filesystem traffic, hardware support for global address space programming, and efficient implementation of programming languages on massively, parallel systems such as HPCs [17].

2.2.1 Connections

On Blue Waters the network topology is of dimension 24x24x24 (X×Y×Z) [16]. In the X and Z directions (rows and columns of cabinets, respectively), every other cabinet is directly connected with loopback cables at the ends for a full torus; the Y direction loops back from the top chassis to the bottom chassis within a cabinet, connecting blades in the same dimension [8]. As shown in Figure 2.4, each node on a blade is connected in the Y dimension, which is called the mezzanine, and each node across blades is connected in the Z dimension, which is called the backplane. Note that if there are more

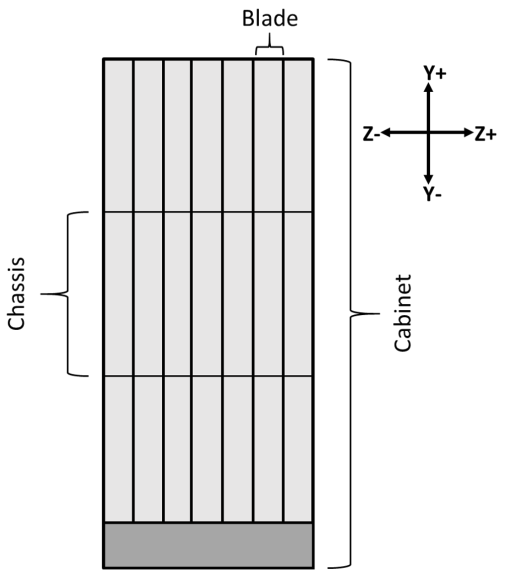


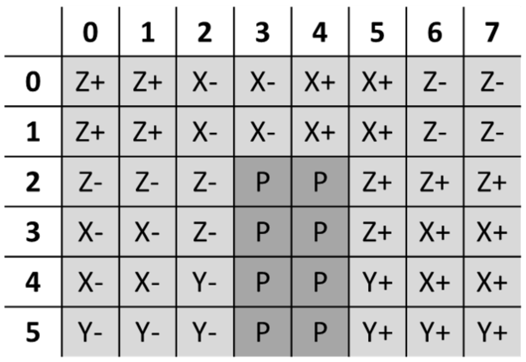
Figure 2.4: A single cabinet with three chassis and eight blades per chassis. The Y+/- and Z+/- connection directions are shown. The unlabeled X dimension goes into and out of the page.

cabinets in the Z direction, cables will connect the nodes across blades in that dimension. Connections in the X direction (not explicitly shown in Figure 2.4) are all cables.

As mentioned, each blade consists of a network mezzanine card that holds two Gemini ASICs. Each of these ASICs communicates within the HSN in six directions: X+, X-, Y+, Y-, Z+, and Z-. There are 10 torus connections: two in each of X+, X-, Z+, and Z- directions (eight total) and one in each Y+ and Y- directions (two total) [17].

2.2.2 Gemini ASIC Router

A Gemini ASIC consists of two network interface controllers (NICs), a network link block, and a 48-port router [17]. Each NIC has its own HyperTransport3 (HTC3) interface, which is used by a node to attach to the ASIC and send its requests as packets through the HSN [17]. There is a network link block that connects the NICs to the router and there is a supervisor block that monitors the ASIC and connects to the L0 blade controller, which in turn relays information to the Cray Hardware Supervisory System (HSS) [17]. The ASIC router follows a tile-based design [17], using a 6x8 array of



	0	1	2	3	4	5	6	7
0	Z+	Z+	X-	X-	X+	X+	Z-	Z-
1	Z+	Z+	X-	X-	X+	X+	Z-	Z-
2	Z-	Z-	Z-	P	P	Z+	Z+	Z+
3	X-	X-	Z-	P	P	Z+	X+	X+
4	X-	X-	Y-	P	P	Y+	X+	X+
5	Y-	Y-	Y-	P	P	Y+	Y+	Y+

Figure 2.5: The 6x8 array of tiles: eight are ptiles (P) and the other 40 are ntiles, labeled by the connection direction.

tiles, eight of which are called ptiles and are reserved for the NIC. The rest, 40 ntiles, are left for the network connections [8]. Figure 2.5 depicts the 48 tiles and their assignments.

2.2.3 Links

Each tile in the ASIC router provides a link. According to Figure 2.5, there are eight links each for the X+, X-, Z+, and Z- directions and four links each for the Y+ and Y- directions. Thus each torus connection consists of four links. Each link is mapped and uniquely identified on the Cray system and logs by its physical location in the tile layout. Each link is comprised of three single-bit bi-directional lanes, which means each torus connection is comprised of 12 lanes [17].

2.3 Hardware Supervisory System

The HSS is a hardware and software system responsible for monitoring the health of Cray Systems, detecting errors in its hardware components and network, and mitigating these errors and their effects. It also controls system startup and shutdown. The HSS contains HSS managers, which oversee various monitoring responsibilities, including boot process, component states, system routing, and more [18]. The HSS also includes the L0 blade and L1 cabinet controllers that monitor nodes and the HSN, respond to periodic heartbeat requests, and log health data such as blade temperatures, power supplies, network performance counters, and runtime software exceptions [18]. The HSS communicates with these management controllers through its own private network, which is separate from the HSN [18], and connects them to the System Management Workstation (SMW).

2.4 System Management Workstation

The SMW orchestrates reliability, accessibility, and serviceability tasks, such as recovery operations in response to failures, and it manages the HSS network [18]. It is a single point of control for the HSS and provides a console for system administrators to manage the Cray system [18]. Logs related to the system health, such as HSS events or HSN failures, are stored on the SMW. These logs are discussed in more detail in the next chapter. The fault injections performed in this study are triggered from the SMW.

2.5 Lustre File Systems

Lustre provides a high-performance, highly scalable parallel distributed file system [18]. It is deployed on many HPCs, including Blue Waters, which houses three Lustre-based file systems with a little over 26 petabytes (PB) of combined usable storage, 34 PB of combined raw storage, and about 1.1 terabytes (TB) per second of storage bandwidth [16]. Further detail on Blue Waters' file system and Lustre may be found in [9].

2.6 ALPS, TORQUE, and Moab

Jobs, which are comprised of applications that a user runs on an HPC system, are scheduled, placed, and launched by a software suite that can vary across HPCs. On Blue Waters, this suite includes the Cray Application Level Placement Scheduler (ALPS) and the Terascale Open-source Resource and Queue (TORQUE) Resource Manager integrated with the Moab Workload Manager [19].

ALPS is the mechanism for application placement, launch, and management [18]. Applications are either submitted through batch jobs or through interactive sessions. ALPS also provides an Extensible Markup Language (XML) interface in order for users to communicate batch job requirements and components to third-party batch systems. The batch system on Blue Waters uses a combination of TORQUE and Moab managers. TORQUE manages the system resources by reserving them for jobs while Moab manages batch job queuing or scheduling. Further detail on TORQUE, Moab, and ALPs can be found in [20], [21], and [22], respectively.

2.7 Fault Tolerance and Resiliency

Failures in large scale systems are inevitable and expected, but applications are still expected to survive and continue running. To account for the high likelihood of errors, Cray systems are designed to provide several layers of protection against errors in both hardware and software. The Gemini interconnect, in particular, was designed to be fault tolerant against network failures. This is achieved through hardware-level error checking and corrections as well as through network hardware redundancy. In the inevitable event of a failure, the system must be able to detect it, mitigate its effects, and recover from it, ideally without requiring a full system reboot so as to minimize system downtime. The system adds its own layers of fault tolerance through the dimension ordered routing protocol, the HSS, and the failover recovery and warm swap restoration procedures.

2.7.1 Hardware-Level Error Detection and Correction

Packets in the HSN are protected by a 16-bit cyclic redundancy check (CRC) and Gemini links reinforce reliable traffic delivery via a sliding window protocol [17]. When a Gemini receives a packet, it checks the CRC and reports if it is incorrect. Before a packet leaves a Gemini, during the transition from router to NIC, the CRC is checked once more in order to detect any corruption from within the router itself [17], such as the routing table. If the CRC fails, the packet is marked as bad, passed along, and then finally dropped at the destination Gemini. Additionally, other error correcting code (ECC) techniques, such as single error correction-double error detection (SEC-DED) and Chipkill are used to detect and correct major errors in memory, data paths, and processors [4]. All such error events are reported to and logged by the HSS.

2.7.2 Network Hardware Redundancy

Redundancy is another common and effective technique to provide a fault-tolerant network. As discussed previously, there are several directions of torus connections with one redundant torus connection in each of the X+, X-, Z+, and Z- directions. Each torus connection consists of four redundant links and each link consists of three redundant bi-directional lanes. Packets traveling through the HSN are spread out over the links by the Gemini adaptive routing hardware [17]. If a lane fails, the adaptive routing hardware masks it out and load balances the traffic over the remaining two lanes in a link [17].

2.7.3 Dimension Ordered Routing

The Gemini interconnect routes traffic based on dimension ordered routing. In a faultless HSN, the protocol is as follows in order [23]:

1. Route traffic in X+/-, Y+, or Z+ directions, until the X dimension is resolved and it reaches the X coordinate of the destination node
2. Route traffic in Y+/- or Z+ directions, until the Y dimension is resolved and it reaches the Y coordinate of the destination node
3. Route traffic in Z+/- directions, until it reaches the Z coordinate and the final destination node

In an HSN with one or more faults present, the routing can no longer be the optimal path and traffic may become unbalanced, possibly leading to network congestion and poorer application performance. Following the protocol, if the network is trying to resolve in the X dimension, but finds that it is blocked in both X+ and X- directions, it will try the Y+ direction next. If that is blocked, then it attempts the Z+ direction [23].

Based on these rules, there are failures that may cause unroutable scenarios, i.e. there are holes or gaps in the HSN that cannot be routed around. For example, using the dimension ordered routing protocol, if two non-adjacent

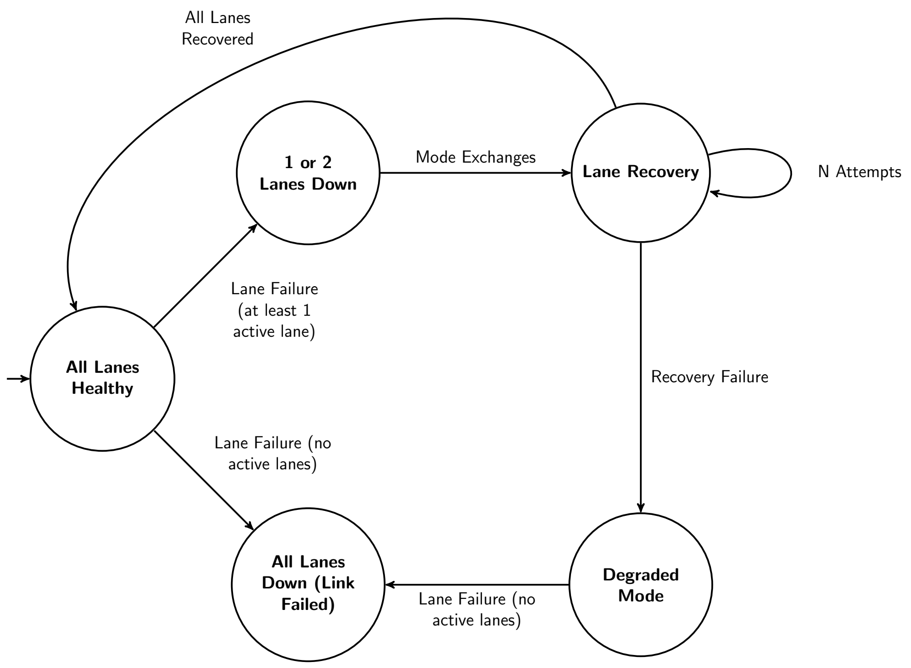

All Lanes Recovered
1 or 2 Lanes Down
Mode Exchanges
Lane Recovery
N Attempts
Lane Failure (at least 1 active lane)
All Lanes Healthy
Recovery Failure
Lane Failure (no active lanes)
All Lanes Down (Link Failed)
Lane Failure (no active lanes)
Degraded Mode

Figure 2.6: State transition diagram for the lane recovery procedure. Adapted from [6].

routers in a single Z dimension loop have failed, then the network is unroutable. When this happens, recovery procedures that involve rerouting will fail. Thus the routing algorithm automatically performs dimension order retries if route computations continue to fail [23]. It will attempt different permutations of dimension orders (e.g. YZX or ZYX, instead of the initial XYZ). This feature can be disabled [23].

2.7.4 Lane Recovery

The redundancy of three lanes per link provides a level of network resiliency, allowing the Gemini interconnect to tolerate up to two lane failures. The HSN runs in degraded mode so that if a lane fails, the network can continue functioning through the other two lanes, albeit at a reduced HSN bandwidth [8]. Without running in degraded mode, a single lane failure would mean an entire link would go down [23].

When a downed lane is identified, it is handled by the L0 blade controller and logged in the SMW's netwatch log [23]. In degraded mode, if all three lanes fail, then the whole link containing the lanes is marked as inactive and the link failover procedure, rather than lane recovery, is initiated [23]. If lane recovery is unsuccessful, the L0 will invoke the lane recovery procedure a set number of times (configurable parameter in a settings file) before giving up [23]. The lane recovery procedure, as shown in Figure 2.6, attempts to bring up all lanes on the channel; it does not target individual lanes for efficiency [23].

While this lane bring-up procedure is a recovery mechanism supported by the system, the ability to manually remove or take down specific lanes in a link using system administrator commands is absent. Given this, we currently do not have any means to trigger this automatic lane recovery in a controlled environment. Thus we leave exploration of this fault and its recovery mechanism out of this study.

2.7.5 Automatic Failover Recovery

When a link fails (or when all three lanes in a link go down), the L0 detects the failure and handles it automatically by triggering the failover recovery procedure [23]. There are many reasons a link may become unavailable, including power loss to a Gemini mezzanine on a blade, power loss to a blade itself, or power loss to a whole cabinet [23]. Link failures may also happen due to faulty cables, routing table corruption, or software deadlock [6].

When the failover is triggered while the system is in an initial state with no other failed components present, a series of recovery actions are automatically executed as follows [23]:

1. **Aggregate failures.** Waits 10 seconds by default in case any more link failures occur. If there are any more failures, the failed links will be processed and added to a list.

2. **Alive.** Determines which blades are alive. If there is a failure due to power loss to a blade, there will be no response to the alive request; a 30-second timeout will occur instead for the powerless blade. If the failure is due to a power loss to a cabinet, there will again be no response to the alive request. Twenty-four blades timeout for 30 seconds and are

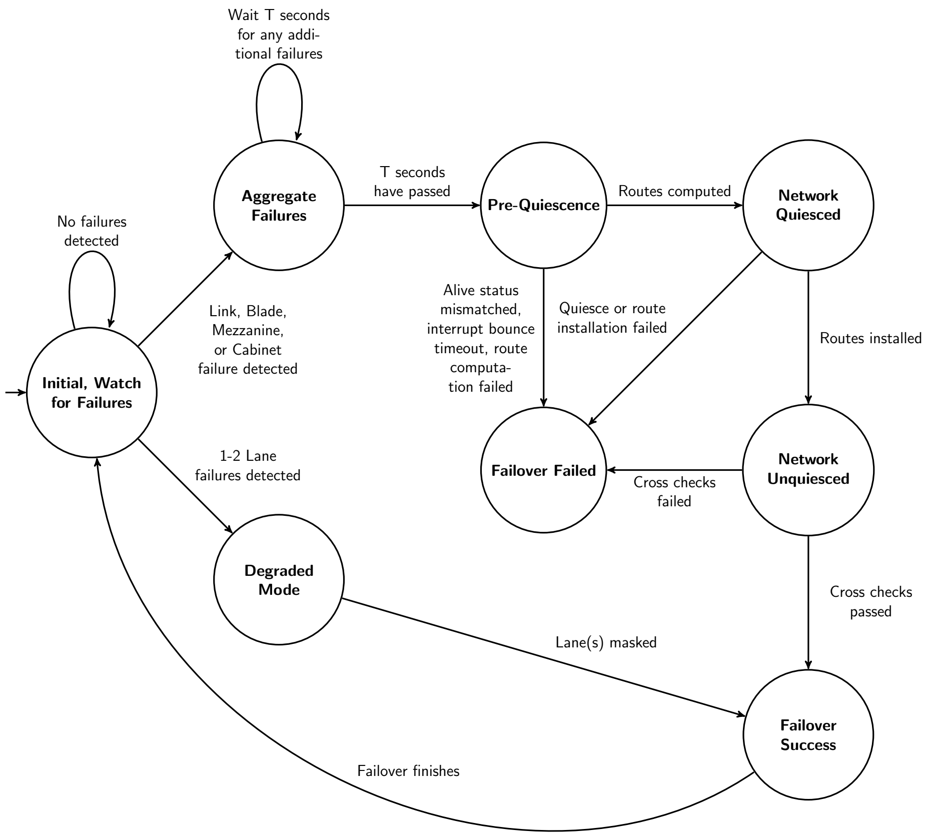

Wait T seconds for any additional failures
Aggregate Failures
T seconds have passed
Pre-Quiescence
Routes computed
Network Quiesced
No failures detected
Initial, Watch for Failures
Link, Blade, Mezzanine, or Cabinet failure detected
Alive status mismatched, interrupt bounce timeout, route computation failed
Quiesce or route installation failed
Routes installed
1-2 Lane failures detected
Failover Failed
Cross checks failed
Network Unquiesced
Degraded Mode
Cross checks passed
Lane(s) masked
Failover Success
Failover finishes

Figure 2.7: State transition diagram for the automatic failover recovery procedure. Adapted from [6].

removed from routing. Each failed LCB (up to 960 in a fully configured system) is marked with an alert flag; each Gemini ASIC on each blade in the failed cabinet is marked with an alert flag.

3. **Route compute.** Computes and stages new routes to the L0s.

4. **Quiesce.** Quiesces the network traffic (i.e. stop all network traffic temporarily) and drains the network traffic.

5. **Switch netwatch.** Makes the netwatch daemon on blades use the newly computed routes.

6. **Down unused links.** Takes down any links that are still active but unused by the new routes.

7. **Route install.** Asserts the new routes in the Gemini ASICs.

8. **Unquiesce.** Allows all network traffic to resume.

9. **Finish.** Performs any final cleanup details to restore the system back to its initial, fault-free state.

This automatic recovery process typically lasts for about 30 seconds. For failure scenarios with power losses to the blade or whole cabinet, the recovery procedure can take up to 60 seconds due the extra timeout time. On much larger systems, this recovery process is expected to take much longer. For example, on Cielo, which had almost 9000 nodes, one recovery took over 600 seconds [15]. Excluding recovery time, the failover processes for link, connection, node, and blade failures are exactly the same.

Quiescing and draining the network traffic is important to avoid network deadlocks, which may occur if there are still packets being routed while new network routes are being installed. Usually, the failover process will avoid disrupting and quiescing the network by instead masking any failed lanes and operating in degraded mode. This, however, cannot be avoided in the case of complete communication loss between two Gemini ASICs and in the case of blade or whole cabinet failures.

Failover procedures can either succeed and restore the network paths, which is called a successful failover. The other outcome is a failed failover, which can leave the whole system in an unusable state, known as a system-wide outage (SWO). Figure 2.7 shows a state diagram of this failover procedure combined with the lane recovery procedure shown in Figure 2.6.

2.7.6 Warm Swap Restoration

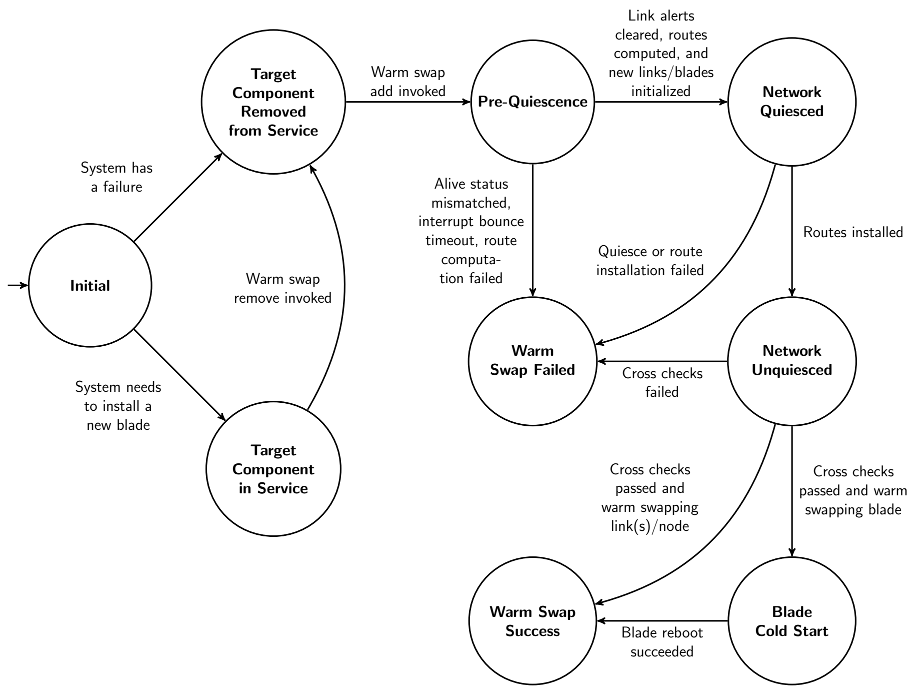
The warm swap procedure is a recovery mechanism to manually add or disable hardware components in a live system, without needing to restart the system or otherwise impact other components of the system. Unlike the previous automatic recovery mechanisms discussed, this process must be manually initiated by a system administrator. In this work, we will use the word *restoration* to refer to the warm swap procedure in order to differentiate from the word *recovery*, which we will use to refer to the failover process.

The series of steps during a warm swap is nearly identical to that of the failover procedure. The main difference is additional stages before the network is quiesced. These new stages include clearing link alerts, testing the rerouting, initialization of new links, and initialization of new blades (if performing a blade warm swap). Every stage following network quiescence is identical to those of the failover process. Figure 2.8 shows the state diagram of a warm swap. This process typically takes about 60 seconds for links or connections and 10 minutes for blades since the warm swap add command performs a cold start on blades.

In this study, after each and every fault injection, the warm swap procedure is invoked by the injector, which is running on the SMW. Warm swapping is necessary to restore the component that was selected for take-down and to return the system and HSN back to an uncongested, fault-free state.

2.8 Test and Development System

In this work, we conducted fault injection experiments on JYC, a single cabinet Cray XE/XK hybrid testbed provided to us by NCSA. NCSA uses JYC as a test and development system on which software and other changes are tested and evaluated before deployment on Blue Waters. JYC shares all the same hardware components (blades, nodes, links, Gemini ASICs, etc.), systems (HSS, Lustre file system, etc.), and software suites and environments



Initial
System has a failure
System needs to install a new blade
Target Component Removed from Service
Target Component in Service
Warm swap remove invoked
Warm swap add invoked
Pre-Quiescence
Link alerts cleared, routes computed, and new links/blades initialized
Network Quiesced
Alive status mismatched, interrupt bounce timeout, route computa- tion failed
Quiesce or route installation failed
Routes installed
Warm Swap Failed
Network Unquiesced
Cross checks failed
Cross checks passed and warm swapping link(s)/node
Cross checks passed and warm swapping blade
Warm Swap Success
Blade Cold Start
Blade reboot succeeded

Figure 2.8: State transition diagram for the warm swap recovery procedure. Adapted from [6].

(Cray Linux Environment, ALPS, TORQUE/Moab, etc.) as those of Blue Waters. The main difference between JYC and Blue Waters is size.

2.8.1 Architecture

Since it is a one-cabinet machine, JYC is constructed from a total of 96 dual-socket AMD Opteron nodes organized across three chassis: 56 XE nodes, 28 XK nodes, and 14 service nodes. Figure 2.9 shows the system map of components on JYC. On this machine, 12 of the 14 service nodes are hosted on Cray XIO blades; the other two service nodes are hosted on a Cray XE6 blade along with two XE nodes. Additionally, the 56 XE nodes are all hosted on 14 Cray XE6 blades and the 28 XK nodes are all hosted on seven Cray XK7 blades. As illustrated in Figure 2.9, Chassis 0 (c0), the bottom chassis, contains XE and service nodes: three blades are entirely service nodes while a fourth blade comprises two more service nodes. No user is allowed to run jobs on any service node. Chassis 1 (c1), the middle chassis, consists of all XE nodes while Chassis 2 (c2), the top chassis, primarily consists of XK nodes (28 out of 32 total). Due to being a one-cabinet machine, JYC's

Gemini interconnect is also reduced by one dimension in the X+ and X- directions as there are no other cabinets to cable to. Thus the number of torus connections is reduced to six: one each in the Y+ and Y- directions and two each in the Z+ and Z- directions.

2.8.2 System Mapping

These compute and network components each have a component name (*cname*). Cabinets have an X and Y position in the physical machine layout. In the case of JYC, its cabinet name is c0-0 since there it is only one cabinet. The next level of components are chassis, which are identified by their vertical position c[0-2] with c0 being the bottom chassis and c2 being the top. The vertical position corresponds to the same Y dimension as in Figure 2.4. The components that make up a chassis are blades, which are identified by their position s[0-7] going from left (s0) to right (s7) in Figure 2.9. This is the same as the Z dimension in Figure 2.4. There are two Gemini ASICs on each blade, identified by their position g[0-1] with g0 as the bottom ASIC and g1 as the top ASIC. On each blade, there are nodes, specifically four nodes per blade or two nodes per Gemini ASIC. Nodes are identified by their position in the Y dimension n[0-3] with n0 and n1 always connected to Gemini g0 and n2 and n3 always connected to Gemini g1. Additionally, each node is assigned a unique decimal value, which serves as its unique node ID (nid000[0-96]) without the need to specify cabinet, chassis, and blade locations. While links are not depicted in Figure 2.9, they nevertheless have their own naming scheme based on physical location as well: l[0-5][0-7]. This positional naming for links comes from the ASIC router's 6x8 tile array as shown in Figure 2.5. This naming is explained in more detail in Figure A.1 in Appendix A.

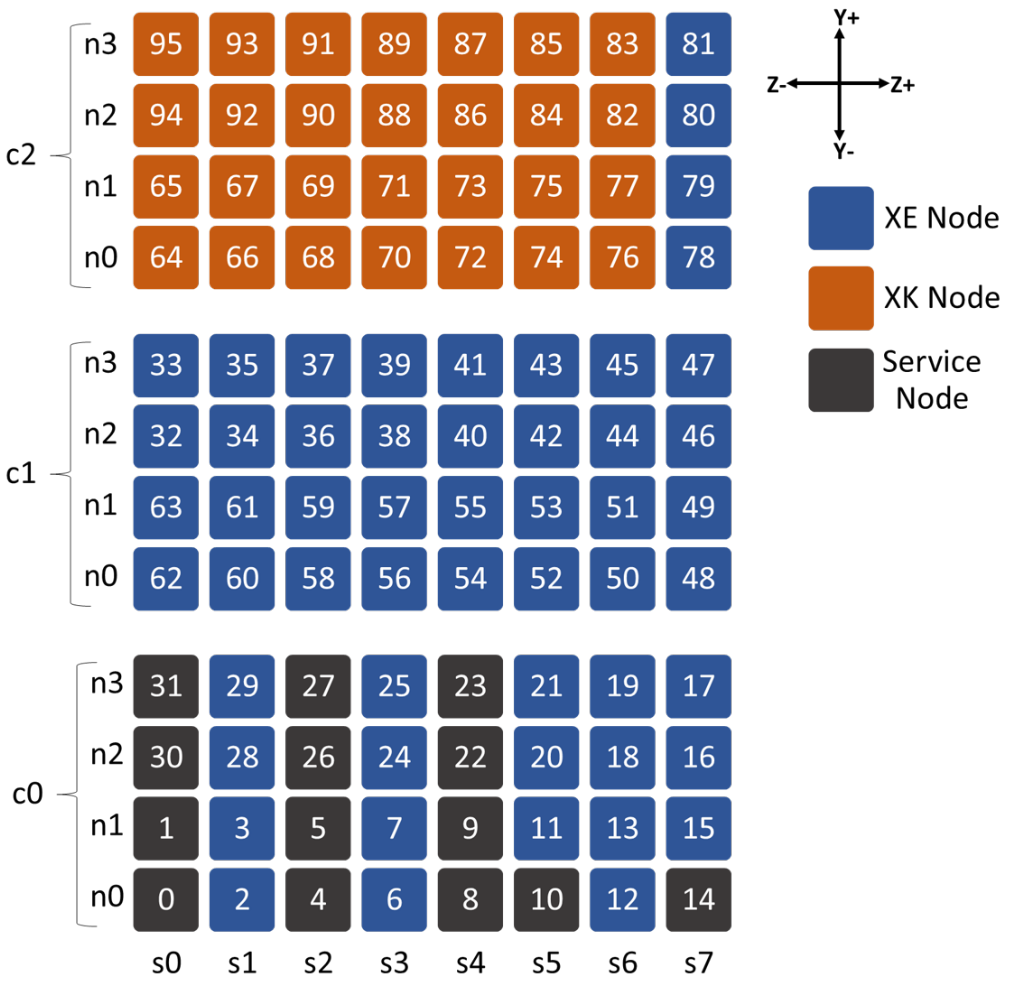


Figure 2.9: System map of components on JYC. Black nodes are service nodes; blue nodes are XE nodes; orange nodes are XK nodes. Chassis are identified as c[0-2]; blades are identified as s[0-7]; and nodes are identified as n[0-3] (component name). Each node is additionally labeled with a unique decimal value, which serves as its node ID.

CHAPTER 3

METHODOLOGY

This chapter presents our fault injection methodology in assessing system resiliency in the presence of faults. Specifically, it details (1) the fault models used to understand system behavior during failures (fault-to-failure scenarios), (2) the benchmark applications chosen to run as workloads during fault injection experiments, (3) the toolkit HPCArrow, designed to execute and log the experiments, and (4) the data collected and processed for analysis and assessment. This approach can be applied to any Cray system, only requiring special consideration for application and execution setup because implementation details are dependent upon each system's configuration suite (e.g. Python versions, compiler versions, Slurm vs. TORQUE job schedulers, etc.).

3.1 Fault Models

We target network and compute components as our fault models. These models involve failures in network links and connections as well as in compute nodes and blades. This study considers both single faults and multiple faults. A single-fault model involves only one fault injection during an experiment, which is defined as the runtime window of a set of applications running simultaneously on the system. A multiple-fault model involves two or more faults. In our experiments, we inject up to two faults, with the second fault deliberately injected during the automatic failover recovery procedure initiated by the first fault being injected.

Production data from Blue Waters showed that the most common failures were single/multiple node failures and link failures [4]. Over one-fourth of the failures observed triggered automatic failovers [4]. These failures also led to system-wide outages (SWOs) in which the whole system is unavailable

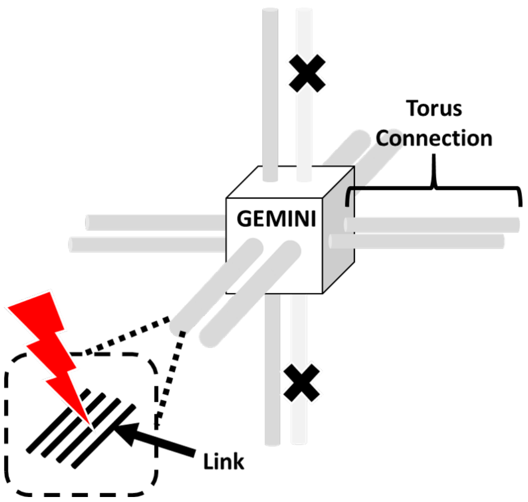


Figure 3.1: Link fault model, targeting a single link in a torus connection of a Gemini ASIC. Two torus connections (lighter colored and marked by black X's) are not used as connections for the HSN, but instead used internally to communicate with the nodes.

(e.g. user cannot log in, cannot access all data blocks of the file system, etc.) [4]. Close to two-thirds of the SWOs are a result of failover procedures failing and affecting re-routing of the Gemini network or access to the Lustre file system [4]. It was also found that network failovers are frequent and that the majority of these triggered failovers were due to lane failures [5]. The next most frequent cause of failovers was link failures [5]. This is also confirmed in studies of other HPC systems, such as Titan [7]. The lane fault model, however, is not included in this study as there is no controlled way (e.g. a Cray administrator command) to trigger lane faults at this time. This is left for future work. Nevertheless, improving failover procedures is important and this study aims to provide a methodology to assess resiliency of recovery procedures.

3.1.1 Single Injection: Link

A link fault, as shown in Figure 3.1, models the failure of a network link that connects two Gemini ASIC routers. We recreate this failure scenario by sending a system administrator command to mark the status of the link component as unavailable or removed from service. In real scenarios, this status flag is typically modified to deactivate problem links. This process effectively masks out any problem links and prevents further use. Traffic that may have been using the problem link is then re-routed to use other links on the same connection. A link failure on one Gemini ASIC also affects the Gemini ASIC on the opposite end. The HSS is responsible for monitoring the health of a Cray system for such failures. When the HSS detects a failure, it feeds this information to a link failover manager on the SMW, thus triggering the automatic failover [24].

The automated failover is expected to mask link failures without causing any major interruption of the system. If the link failure leads to the disconnection of an ASIC, then the failover process needs to compute new routes in order to reroute network traffic around the failed component, quiesce the network (i.e. pause network traffic), install the newly computed network routes on all ASICs, and finally unquiesce the network. As mentioned above, it is possible for this failover process to fail.

This study does not explicitly model single-lane faults (due to a lack of a method to directly cause such a failure). However, because there are three lanes per link, a link failure is the same as three lane failures.

3.1.2 Single Injection: Connection

A connection fault models the failure of a network connection. Recall that a connection, in a torus topology, such as Gemini, consists of either 8 links for each X+, X-, Z+, Z- direction or 4 links for each Y+, Y- direction. Failing a connection means failing or deactivating all the links between two Gemini ASICs. This failure scenario is recreated by modifying the status flags of all the links to be marked as unavailable or removed from service. Note that from now on, what we call a connection refers to all links or all torus connections in one connection direction, as shown in Figure 3.2.

The automated recovery process is the same as the link failover procedure, which involves recalculating and rerouting network paths around the hole created by the connection failure, quiescing the network, installing the new network routes on all ASICs, and finally unquiescing the network. There are some topologies that are unroutable and thus lead to a reroute failure [23], but in general, no major system interruptions are expected.

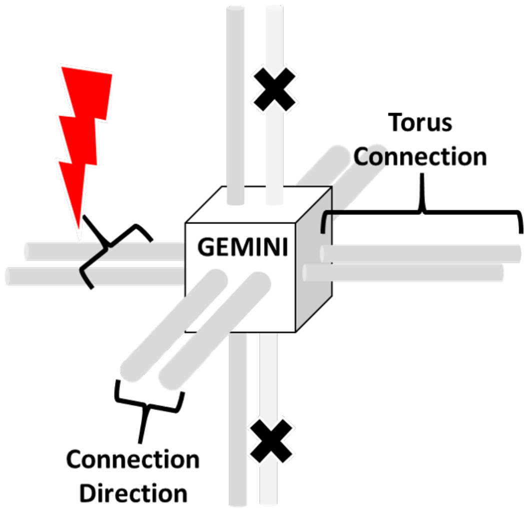


Figure 3.2: Connection fault model, targeting one connection direction of a Gemini ASIC. Two torus connections (lighter colored marked by black X's) are not used as connections for the HSN, but instead used internally to communicate with the nodes. Note the distinction between a torus connection and a connection direction.

3.1.3 Single Injection: Node

A node fault, as shown in Figure 3.3, models the failure of a compute node. In this study, we were asked to not target service nodes due to their importance for critical system services, but we do target XE and XK compute nodes. We recreate this failure scenario by sending a system administrator command that in turn sends a non-maskable interrupt (NMI) to the CPU to cause it to hang. Any application running on this failed node would fail and terminate. The failover process for a node is identical to that for a failed link

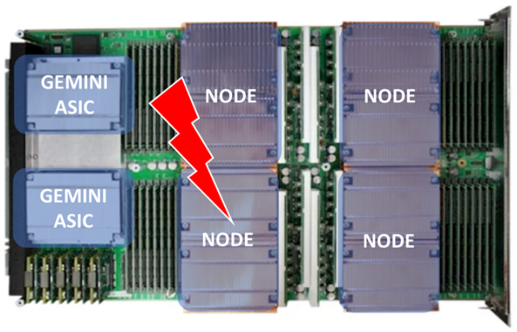


Figure 3.3: Node fault model, targeting one of the four nodes on a blade. The blade shown is a Cray XE6.

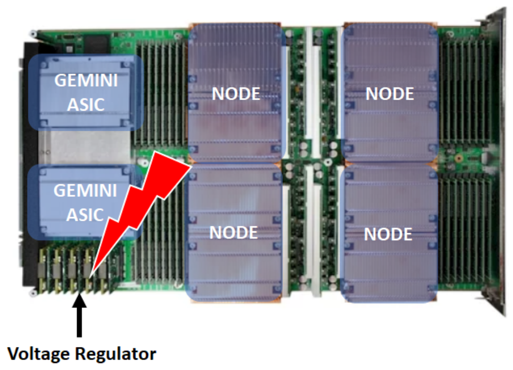


Figure 3.4: Blade fault model, targeting the whole blade and its components by turning of the voltage regulators. The blade shown is a Cray XE6.

or connection.

Additionally, there is also no automated method to reboot a single node, which limits obtaining a statistically significant number of node failure experiments. The only way to reboot a node is through our automated method for rebooting a whole blade or through manual interaction with the SMW. Neither situation is ideal for realistic (a whole blade is never rebooted for a single failed node), repeated experiments given limitations on access to the SMW. For these reasons, there are few node failure experiments in this work. These node failure results are reported together with blade (multiple node) failure results as we did not observe any difference in system or application behavior between the two types of faults.

3.1.4 Single Injection: Blade

A blade fault models the failure of a blade, which consists of four compute nodes, two Gemini ASICs, and six network connection directions or 40 network links in total. When a blade is shut down, all these components are marked as down, which means any application(s) running on the failed blade would also prematurely terminate. We recreate this failure scenario by sending a system administrator command that turns off the voltage regulator of the mezzanine in the blade, as shown in Figure 3.4.

The automated recovery process is expected to reroute around the two failed Gemini ASIC routers just as in the failover procedures for the previously discussed fault models.

3.1.5 Injection During Recovery

Injection during recovery is a model of multiple faults with one or more faults injected during the failover recovery procedure of another single fault. One goal of this study is to recreate the failed failover scenarios observed in Blue Waters field data as well as the fault injections performed on Cielo. In those systems, SWOs and network deadlocks were observed [6],[5],[15].

This model involves multiple faults, chosen from a combination of the previously discussed single injection fault models. For example, a single connection fault is injected as the initial fault to trigger a recovery process. During recovery, a second connection fault is injected.

Behavior of the recovery processes when faults are injected during recovery is not well documented nor understood. Subsequent chapters will discuss results and observations of the system's behavior and the resiliency of failover processes under the stress of multiple faults.

3.2 Benchmark Applications

The main goal of this study is to assess the reliability and resiliency of HPC systems in the presence of faults. One way to evaluate resiliency of HPC systems is through observing the behavior of applications or benchmarks running on the system. Being able to control applications running on the system during experiments and having direct access to the applications' log outputs provides both a real-time monitor of application health as well as a way to observe fault propagation through the rest of the system. This in turn can help provide deeper insights of fault tolerance at the application and system levels. To conduct fault injection experiments in environments that are as realistic as possible, we chose various HPC benchmark applications to generate varying levels of network and computational activity. The list of benchmarks is shown in Table 3.1.

These applications were chosen to cover and evaluate resiliency of a spectrum of HPC programming frameworks, including Charm++, MPI, and PGAS. These frameworks, in the context of fault tolerance, are discussed in more detail in the following sections. These chosen applications cover a wide range of scientific applications, such as particle physics, molecular dynamics, and seismic simulations.

The applications are executed at varying node scales and tuned to run for approximately 30 to 40 minutes, which is sufficient time for one fault injection and its corresponding recovery and restoration processes. For example, the whole blade recovery and restoration procedure can take about 10 minutes to complete. All applications are scaled to run on 64 nodes or less. This in turn limits the applications that we could compile and run simply because certain applications require the resources of more nodes than the testbed can provide.

Beyond the work presented in this study, future work aims to compare reliability and resiliency features and behavior across HPC systems with different interconnect architectures, such as the Aries Dragonfly Interconnect. While discussion of the Aries interconnect and its details are beyond the scope of this thesis, that ongoing work enforced another constraint on the applications that were chosen in the later campaigns of this study. HPC systems and testbeds owned by different organizations are configured with certain programming environments and various versions, a flexibility provided by Cray. This flexibility, however, means that some systems have different compiler and module versions, which further limits the applications that we could compile and run for the comparison fault injection experiments on Aries systems.

3.2.1 Charm++

Charm++ is a parallel programming framework based on the C++ programming language, actively developed and maintained by the Parallel Programming Laboratory at the University of Illinois at Urbana-Champaign. Its key novel concepts include a message-driven execution model and an adaptive runtime system in charge of migratable work and data units [26]. The applications we selected under Charm++ (AMR, Kripke, LeanMD, and NAMD) utilize the user-level Generic Network Interface (uGNI) API, a native low-level interface for Gemini hardware on Cray XE/XK systems, with shared-memory optimization pthreads and Cray HugePages modules [27].

Charm++ maintains fault tolerance only through the checkpoint and restart mechanism, making use of its novelty of migratable objects, but even the developers are skeptical whether this checkpoint/restart scheme is enough for

Table 3.1: Benchmark applications information [25].

Application Name	**Discipline**	**Programming Model**	**Languages**	**Characteristics**
Adaptive Mesh Refinement (AMR)	Numerical Analysis	Charm++ (SMP, HugePages over uGNI)	C++	-
Anelastic Wave Propagation (AWP-ODC)	Seismic	MPI (PGI Environment)	Fortran, C++	Structured Grid, Sparse Matrix, I/O
Kripke	Particle Physics	Charm++ (HugePages over uGNI)	C++	Structured Grid, Dense Matrix
LeanMD	Molecular Dynamics	Charm++ (HugePages over uGNI)	C++	N-Body, FFT
MIMD Lattice Computation (MILC)	Particle Physics	MPI (Intel Environment)	C/C++	Structured Grid, Sparse Matrix
Nanoscale Molecular Dynamics (NAMD)	Molecular Dynamics	Charm++ (SMP over uGNI)	C++	N-Body, FFT
Pseudo-Spectral Direct Numerical Simulations (PS-DNS)	Fluid Dynamics	MPI (Cray Environment)	Fortran	Structured Grid, FFT
Unified Parallel C Fourier Transform (UPC-FT)	Numerical Analysis	PGAS Unified Parallel C	C	FFT

exascale systems where failure rates are expected to increase significantly [28],[29].

3.2.2 Message Passing Interface

The Message Passing Interface (MPI) is an efficient, scalable, and vendor-independent specification for message-passing parallel programming libraries. It is the industry standard for message passing parallel programs running on virtually any parallel computing hardware platform, including HPC platforms. MPI supports point-to-point and collective communication routines [30]. There are multiple implementations of the MPI standard; our chosen MPI applications, AWP-ODC, MILC, and PSDNS cover three implementations: PGI, Intel, and Cray, respectively.

The MPI Standard itself does not consider fault tolerance as a property of the standard itself nor of any MPI implementation on its own. While the developers assert that fault tolerance is a property of MPI programs coupled with MPI implementations, the MPI Standard nevertheless still specifies reliable communication [31]. This means that all MPI implementations must detect and handle network faults (e.g. through retransmission of the message, informing the application of the presence of an error, etc.). The standard also specifies certain error handlers, but the handlers can either be built in or user defined [31]. In general, the standard itself allows for flexibility in implementation when it comes to fault tolerance. Suggestions for writing fault-tolerant programs include checkpointing and use of MPI's intercommunicators to ensure survivability if one party has failed and ceased communication [31].

3.2.3 Partitioned Global Address Space

The Partitioned Global Address Space (PGAS) is yet another parallel programming model, but one that aims to combine the advantages of both distributed-memory models (performance and locality) and shared-memory models (simplicity and programmability) [32]. It provides a global address space, which is shared and accessible by any process yet also partitioned such that portions are local to each processing element, process, or thread [32]. There are various PGAS languages; this work focuses on Unified Parallel C, a programming PGAS-like language descended from the C language [32].

It is known that PGAS applications are susceptible to network failures [5]. This vulnerability stems from requiring atomic memory operations and ordered message delivery [5]. Lost transactions during link failures, such as loss of response, are not tolerated and can lead to a duplicate transaction to be present. Messages that arrive out of order will also disrupt the application's forward progress. DMAPP is a communication library used by UPC-FT. It provides remote memory access (RMA) between processes, but it does not support error recovery in the presence of link failures, leaving it up to applications to handle the error.

3.3 Fault Injection Toolkit: HPCArrow

To investigate the fault-to-failure propagation and the impact of faults on systems and applications, we developed a SWIFI-based tool called HPCArrow. This toolkit systematically executes fault injections on Cray systems with both Gemini and Aries interconnects. Fault injection experiments aim

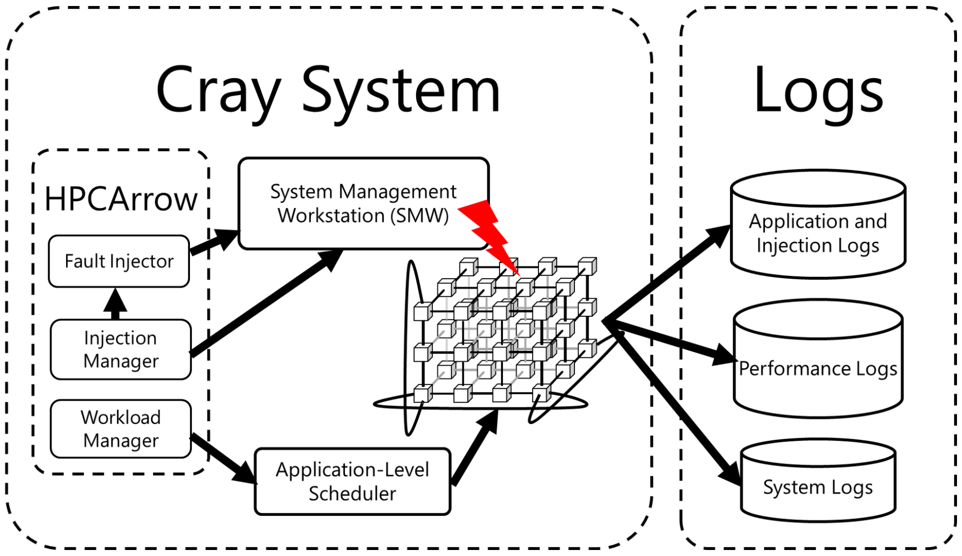


Figure 3.5: Architecture environment of the fault injection experiments, depicting the modules of the HPCArrow toolkit and their interactions within the Cray System, including the final output logs.

to create failure scenarios that emulate actual permanent faults at the hardware component level.

HPCArrow provides the advantage of a controllable and repeatable injection environment. The toolkit can inject one or more faults of any fault type (link, component, node, blade) into either a random or user-specified location at a random or user-specified time. It also supports the specific restoration commands to automatically restore the system back to an unaffected, fault-free state after each injection experiment. Note that restoration is different from the system's automatic failover recovery procedure. Failover is the system's response to mask out or reroute around the failed component(s) as much as possible to minimize disruption of system service; it does not fix the failed component itself. Restoration is a manual procedure for which a system administrator is responsible. Restoration must be manually triggered in order to return the failed component back into service.

This toolkit comprises three main modules, which are described in detail in the next few sections. The workflow of the toolkit in tandem with the target system is illustrated in Figure 3.5.

3.3.1 Workload Manager

Application workloads are configured, launched, and logged by the Workload Manager module to simulate real network and computational activity on a target system. An application can be launched at user-specified locations on the target system and at various user-specified scales. Multiple applications can be launched simultaneously as well.

In this study, the Workload Manager supports as small as two-node applications (smallest allowable size on JYC) up to 64 nodes at intervals of powers of 2 (2, 4, 8, etc.). Due to NCSA restricting our system reservation to the top two chassi, 64 nodes is the largest size available. Regardless of scale, every application is configured for about 30-40 minutes of run time. Every application has its own unique parameters to configure for scaling and run time. See Appendix D for details on these parameters.

The various workload sizes, topology placement, node types, and application-specific runtime parameters are enumerated in YAML configuration files that can be easily modified or extended by the user. Through these configuration files, the Workload Manager module can generate job submission scripts tailored to support various target systems with different resource managers, cluster job workload management packages, or job schedulers. Currently, it supports Moab/TORQUE for JYC/Blue Waters.

3.3.2 Fault Injector

The Fault Injector module is responsible for performing the fault injections by executing the system-specific commands to take down network or compute components. This can be either a random selection of a component (by the tool) or a user-specified component. These selection methods will be discussed in further detail in Chapter 4. In the case of multiple fault injections, this module handles timing the fault injections either based on a user-specified delay or through monitoring system logs to inject faults during recovery procedures. The injector is also responsible for invoking the appropriate restoration procedure corresponding to the fault that was injected.

Fault injection and system restoration are executed via system-specific commands. In this work, those are Cray administrative commands, which are issued from the SMW where this Fault Injector module runs in the background. The injector extracts both the user-specified target component(s) and injection timing delay from the names of currently running jobs. It then looks up the parameters and commands for performing both the injection and restoration phases of the injection experiment and executes those commands. Tables B.1 and B.2 in Appendix B detail the injection and restoration commands, respectively. Throughout each experiment, the injector constantly logs injection and restoration commands as well as standard output and errors and stores these outputs in text files. These are the injector logs.

Because the fault injector calls Cray-specific administrative commands to inject faults, this module can be modified to incorporate other low-level HPC fault injection frameworks that can be triggered by an executable program or shell script.

3.3.3 Injection Manager

The Injection Manager is a daemon-like module responsible for monitoring the target system's ALPS queue to ensure space for applications and that no other unintended jobs are running during an injection experiment. This module also watches for jobs started by pre-specified user(s). The presence of jobs started by a pre-specified user signals the Injection Manager to initiate an injection experiment. If an injection is specified via the job names, the Injection Manager parses the job name and subsequently triggers the injection through the Fault Injector module. At a later pre-specified time, the Injection Manager also collects the appropriate injection logs and system logs that are on the SMW and inaccessible to a normal user. This workflow is illustrated in Figure 3.6.

Due to restrictions of JYC usage for fault injections, the Injection Manager ensures limited user intervention, meaning that we do not have to manually run injection or restoration commands, nor do we have to be logged in on administrator accounts on the SMW. To trigger an injection experiment as a user, the target component type(s) and name(s) are specified in the job names that are submitted to the job scheduler. The injection delay and recovery stage is also specified here as well. The Injection Manager, which is monitoring the job queue from the SMW, parses the job names to determine what type(s) of injection(s) to run and when. Figure C.1 in Appendix C out-

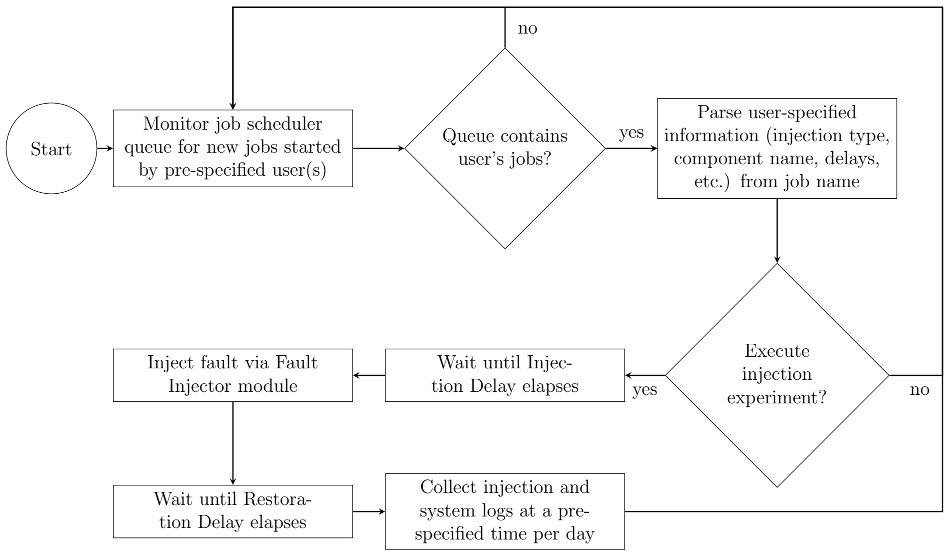

no
Start
Monitor job scheduler queue for new jobs started by pre-specified user(s)
Queue contains user's jobs?
yes
Parse user-specified information (injection type, component name, delays, etc.) from job name
Execute injection experiment?
yes
no
Wait until Injection Delay elapses
Inject fault via Fault Injector module
Wait until Restoration Delay elapses
Collect injection and system logs at a pre-specified time per day

Figure 3.6: Workflow of the Injection Manager module of the HPCArrow toolkit. The Injection Manager continuously runs on the SMW like a daemon and must be started by the system administrator.

lines the parts that the injector parses in order for us to remotely execute an injection experiment without needing to have constant administrator privileges and access to the SMW. A list of seen and injected jobs is kept by the Injection Manager temporarily to ensure that it does not inject more than once on the same set of jobs. In addition, the Injection Manager also ensures that fault injections are only executed during the allocated reservation hours and that all injection, system, and performance logs, which are normally restricted behind system administrator privileges, are made available to a pre-specified user via a log transfer script.

This Injection Manager is an optional module, tailored to our needs and to workaround certain restrictions. The rest of HPCArrow can be run directly on the SMW without running the Injection Manager. Similarly, the log transfer script does not need the Injection Manager to run as it is a simple shell script used to collect all logs.

3.4 Data Collected

To assess the impact of the proposed fault injections on both the system and its running applications, we collect and analyze system-generated logs, monitoring and performance data, application outputs, and injection experiment logs. This data is provided by various collection and aggregation services, including the fault injection toolkit that was discussed in the previous section. The data generated by applications launched on the system without injections or presence of any faults were used as our golden outputs in order to compare against for the later fault injection experiment outputs. These golden logs and outputs were also used to identify the most utilized components on which to inject faults. Analysis of all this data collectively provides a quantifiable way to observe and describe system and application behavior in the presence of faults.

3.4.1 Event Analysis: LogDiver

The system logs that we collect are generated by Cray logging daemons that are always running on the system at various levels, including the SMW and cabinet and blade controllers. They are typically started during system boot, although some may need to be manually turned on by the system administrator. These logs contain information on the network, system, hardware components, and the SMW itself. Table 3.2 describes the specific system logs collected and analyzed for this work.

To parse and identify the events and information necessary to evaluate resiliency, we use a tool called LogDiver.[1] It is a tool designed for analyzing application-level resiliency in extreme-scale environments based on system and hardware error logs on Blue Waters [34]. In this work, we utilize LogDiver to extract network failover recover stages and hardware errors to verify the completion statuses of recoveries and restorations, and to diagnose both abnormal application terminations (crashes and hangs) and recovery failures. To accomplish this, LogDiver filters based on pre-configured regular expressions (regex) that match with pre-specified events of interest in the collected system logs. The regexes were assembled from manual inspection of events in collected logs.

[1]LogDiver was developed by Catello Di Martino and Saurabh Jha at the University of Illinois at Urbana-Champaign (UIUC) based off of data produced by Blue Waters.

Table 3.2: Cray system logs [33] collected and analyzed in this work.

Log Name	**Description**
commands	All commands, excluding xtdiscover, executed on the SMW.
events	HSS-wide events, health and heartbeats, and sequence identifiers; recorded by the event router.
hwerrlog	Hardware error events in the ASIC network chip; the xthwerrlogd daemon monitors for these errors.
netwatch	Link control block (LCB) and router errors; the xtnetwatch daemon monitors the system HSN faults interconnect for these errors.
nlrd	HSN failures, recovery actions taken and phases in event of failures, warm swap requests, and HSN congestion; the network link resiliency daemon (nlrd) monitors blade controllers for these events.
smwmessages	SMW hardware and environmental history.
xtdiscover	Output from running xtdiscover command on the SMW, used to detect Cray system hardware components and to capture changes in hardware configurations in the HSS.

Netwatch Events

The xtnetwatch daemon collects interconnect metadata from the system HSN, logging link control block (LCB) and router errors. Such events, described in Table 3.3, include mode exchanges, transmitting and receiving packets, inactive links, bad send EOP error, send packet length error, and routing table corruptions.

Nlrd Events

The xtnlrd daemon logs interconnect failures, recovery actions taken in response to those failures, and details of each phase of recovery. It also logs warm swap stages and network congestion events. The events of interest, described in Table 3.4, include the recovery stages of the failover protocols, which were discussed in the previous chapter. This information is used to verify the success of failovers and warm swaps or detect abnormal network responses to fault injections.

Table 3.3: Netwatch error events of interest during analysis.

Event	**Description**
Link Inactive	A link has failed and has been marked as inactive.
Bad Send EOP error	On Gemini interconnects, each packet is divided into 24-bit physical units (phits), always with the last phit of a packet serving as the end of packet phit [17]. A corrupted packet will have its end of packet phit marked to be discarded at its destination.
Send Packet Length error	The length of a packet does not match with the expected length at the packet's destination.
Routing Table Corruption error	Every router contains a routing table, which holds the forwarding information for incoming packets. An incoming packet is matched to the routing table entry using its destination. If a routing table is corrupted, packets may reach incorrect destinations, link failures may occur, and network congestion may arise.

Table 3.4: Nlrd network and recovery events of interest during analysis.

Event	**Description**
Link Failed	A link has failed.
ASIC Failed	An ASIC router has failed. Typically seen when a blade fails.
Link Recovery Successful	A link failover finished successfully. This is a generic message even for blade and connection failures since they all involve link failures.
Link Recovery Failed	A failover failed.
Network Quiesced	All traffic is temporarily suspended and any in-flight packets are drained from the network.
Network Unquiesced	All traffic is allowed to resume.
Throttle	The network is experiencing congestion and all blades are instructed to throttle traffic.
Unthrottle	Network congestion has been handled and all blades are instructed to unthrottle network bandwidth.
Warm Swap Successful	A warm swap restoration procedure finished successfully.
Warm Swap Failed	A warm swap restoration procedure failed to restore a component back into service.

Hwerrlog Events

The xthwerrlogd daemon monitors and logs for hardware error events that occur on ASIC network chips. The errors of interest, described in Table 3.5, are related to network deadlocks observed in field data [15]. Netwatch events are also reported in these hardware logs.

Table 3.5: Hardware error events of interest during analysis.

Event	**Description**
LB Lack of Forward Progress	Traffic flow is stopped through a NIC and packets are discarded. This may indicate a severe network issue if this error is generated across the entire network. It is not critical if contained in a small set of routers.
NIF Squashed from Tile Request	Packets are squashed due to failed consistency checks (e.g. ECC, CRC, misroute). This may indicate packet corruption or bad routing.
ORB RAM Scrubbed Upper/Lower Entry	If a network request times out, it is scrubbed or removed from the Output Request Buffer (ORB). ORB can free the upper 64 entries or the lower 64 entries of the ORB RAM as shown in the names. This is a transient hardware error, but if continuously generated in the logs, it could mean a severe problem with the network, such as a deadlock.
ORB Request with No Entry	A response packet is received, but does not correspond to a request entry in the ORB RAM. This is a critical error as it may indicate a routing table corruption.
Receiver 8b10b Error	This indicates a transmission error.
SSID Response RequestTimeout	This error can result from failed HSN components, a failure of the node to which the request was sent, or a transient error that results in a packet being discarded. It could also result from severe network congestion and can be an indication of a network deadlock when it persists over a long period of time.
SSID Response Protocol Error	This is a complex error, sometimes indicating network problems. It can be seen during network re-routing triggered by quiescing. It can also be triggered by process termination interrupting the Gemini low-level protocol.
SSID Detected Misrouted Packet	The destination field in the packet does not match the endpoint at which the packet has been received. This can be caused by a routing problem in the network or a mis-addressed packet (bad/invalid NIC address).
SSID Stale on Response, SSID Stale	A warm swap restoration procedure finished successfully.

3.4.2 Network Performance Counters

Many Cray systems, including Blue Waters, are beginning to utilize a high fidelity, global system monitoring module called OVIS,[2] which aims to detect and diagnose abnormal system behavior or limitations and to provide system-level insight into resource utilization (e.g. how the network is utilized, stressed, or congested). OVIS contains a service called the the Lightweight Distributed Metric Service (LDMS), which collects and transports network performance data [35]. This service logs numerical network performance (throughput in bytes/second) in the X+, X-, Y+, Y-, Z+, and Z- directions for each Gemini on the target system. In this study, LDMS is configured to sample traffic data (in bytes/second) at one-second intervals. Due to low overhead, LDMS can be deployed across an entire HPC platform without any significant detrimental impact on the system itself [35]. However, LDMS is not resilient against network failures (due to needing to aggregate data collected on each node) and node failures (the service itself runs on service nodes). Data is collected by on-node daemons and held in memory on a node until they are overwritten by the next sample. It is pulled from memory by other daemons via RDMA. During node failures, the data collected by on-node daemons are lost if memory is overwritten or lost. During failures in the HSN, i.e. during a network quiescence, data points that would have been collected in that period are dropped.

In this work, we use LDMS network data for guiding component injection selection in our fault injection campaigns on JYC. Based on traffic data from pre-run applications, we select components with the maximum throughput traffic in order to ensure that a single component injection will yield the most impact on the system. We also utilize LDMS network data for diagnosing abnormal application behavior (e.g. observing whether there is traffic during application hangs).

Figure 3.7 shows an example timeplot of LDMS traffic data collected for one normal (i.e. without injections or abnormal system behavior) run of a Charm++ NAMD application on 4 nodes. Based on this plot, we can visually see that the connection direction with the highest traffic is the Z- direction on node nid00060.

[2]The OVIS Project is developed by SNL in collaboration with NCSA and Cray Inc.

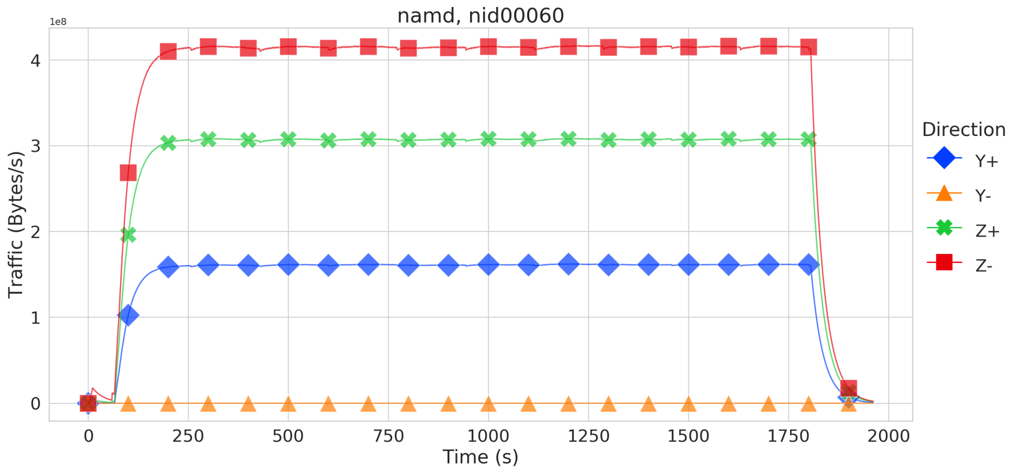


Figure 3.7: Example of LDMS traffic data collected for one run of a NAMD application without injection.

3.4.3 Application Outputs and Injection Logs

Having various applications running on the system is also key to assessing resiliency of application frameworks in the presence of faults. Every workload run information is redirected from stdout and stderr to one or more logs. These application-specific outputs may contain information such as timestamps, computation steps and timings, exit statuses, and any abnormal or critical errors. Information reported by the ALPS is also redirected into these logs. These include network quiescence and throttling events when automatic failure recoveries or manual warm swap procedures are invoked.

The Injection Manager module also outputs it own set of injection logs, collecting book-keeping information regarding injection experiments. This includes timestamps, jobs running on the queue, jobs and/or components selected for injection, and redirected information from stdout and stderr on the SMW.

We use these application and injection logs as another avenue of information to correlate system-level failures and recoveries to fault injection events that occur during workload runs. They also help diagnose the causes of abnormal application behaviors and termination as other studies on HPC failures do not have such detailed information about application behavior and failures.

CHAPTER 4

EXPERIMENTS AND RESULTS

This chapter details fault injection campaigns conducted on Blue Waters' 96-node JYC testbed. A campaign consists of a set of experiments that are thematically unified in injection type (number of injections), fault selection methodology, fault model types, fault injection timing (either seconds after all applications start or during user-specified recovery stages), and applications. These are outlined for each campaign in Table 4.1 and discussed in detail in this chapter.

Across the fault injection campaigns on JYC, a total of 321 experiments were executed, collected, and analyzed. Out of 321, 30 were baseline runs without injections, which were used as the golden outputs or data as comparison reference for later experiments' outputs. The remaining were fault injection experiments. These experiments spanned seven months and took about 9030 node-hours.[1] The following sections detail the experimental setup, which covers application configurations and injection selection methods applied across the campaigns. Each campaign and its results are then presented in separate sections.

4.1 Experimental Setup

4.1.1 Applications and Topology Configurations

As discussed in the previous chapter, applications are used as one perspective to assess system resiliency under the stress of faults. This portion of the experiments necessitates a great deal of setup time, including compilation

[1]A node-hour is a unit of work that an HPC node performs in one hour. More generally, this means that an application ran for a Walltime of t hours on n nodes, e.g. two nodes running 0.5 hour each = one node-hour.

Table 4.1: Summary of fault injection campaigns.

	Fault Injection Campaign I	**Fault Injection Campaign II**	**Fault Injection Campaign III**	**Fault Injection Campaign IV**
Injection Selection Method	Random	Traffic-based	Traffic-based	Traffic-based
Injection Type	Single	Single	Single	Multiple (During Recovery)
Fault Types	Link Connection Node Blade	Link Connection Blade	Link Connection Blade	Link-Connection Connection-Link Blade-Link Link-Blade Connection-Blade
Injection Timing	220 s	220 s	600 s 1200 s	220 s (1st) Recovery Stages (2nd)
Application Set	Set I	Set II	Set II	Set II

and parameter adjustments for scale, node types, and time. See Appendix D for details on these parameters. Due to the duration of failover and warm swap procedures, which can take about 10-15 minutes for a single blade fault, applications were preconfigured to run for about 30 to 40 minutes. Applications are also preconfigured with a `Walltime`, which specifies the maximum amount of time the application can run regardless of completion of computations. This parameter is set to be two hours for all application runs across the four campaigns. Combined, these two constraints dictate the duration of experiments: one experiment run can be as fast as about 33 minutes and another as long as two hours. This upper bound, which is often reached by certain applications and faults, is the limiting factor on running a statistically significant number of repeated experiments.

In total, nine different applications were compiled and executed on JYC. These nine were split and mixed to construct two sets of applications: one used in Campaign I as a proof of concept and the other used in all subsequent campaigns. Applications are varied in workload sizes (i.e. the number of nodes they run on), which are powers of two, ranging from as small as two nodes up to 64 nodes. These limitations are set by JYC scheduler (two nodes) and NCSA restrictions (2 chassis = 64 nodes). Applications are also varied in placement around the system. While fault injections are limited to the top two chassis of JYC to avoid disrupting service nodes as requested by NCSA, applications are still placed in the bottom chassis in some configurations to watch for fault propagation.

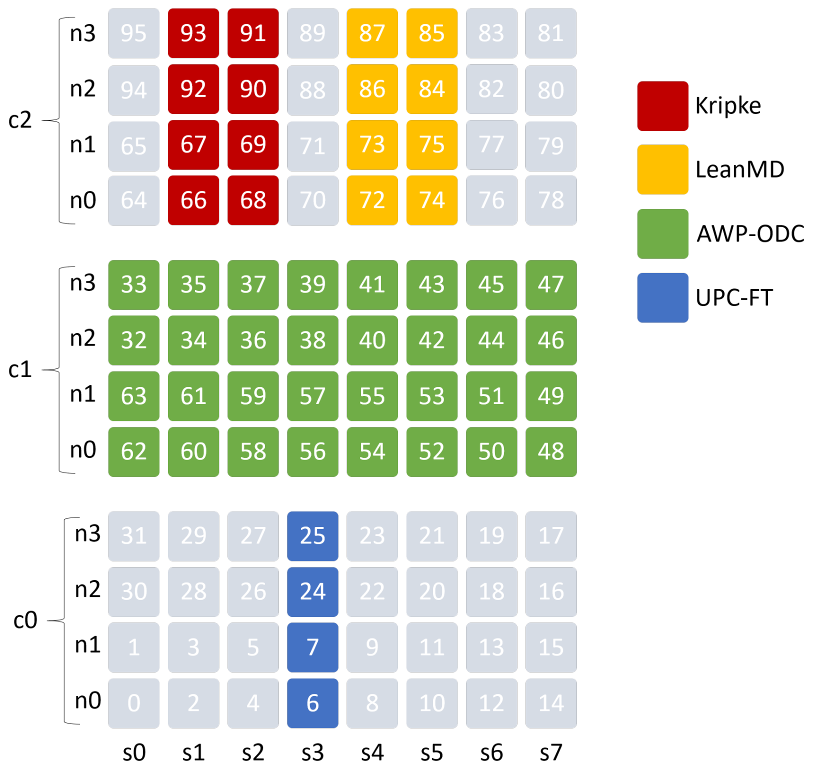


Figure 4.1: Application Configuration 1 on JYC for Fault Injection Campaign 1. Grayed-out nodes are unused nodes.

Application Set I. Campaign I features six applications: AMR (SMP), AMR (HugePages), AWP-ODC, LeanMD (HugePages), Kripke (HugePages), and UPC-FT. Campaign I consisted of five various application topology layouts or configurations, covering various workload sizes, topology placements, and topology density of applications. Figure 4.1 shows an example configuration of how applications were laid out in Configuration 1. Grayed out nodes are unused nodes. Refer to Appendix D for the other four configurations used in Campaign I.

Application Set II. Campaign II also features six applications: AMR (HugePages), LeanMD (HugePages), Kripke (HugePages), MILC, NAMD (SMP), and PSDNS. Note that three of them (MILC, NAMD, and PSDNS) are unique from those executed in Campaign I. From this set of applications, we constructed three various application topology layouts or configurations, covering various workload sizes, topology placements, and topology density of applications. Figure 4.2 shows an example configuration of how applications were laid out in the Dense Configuration, filling up nearly every compute

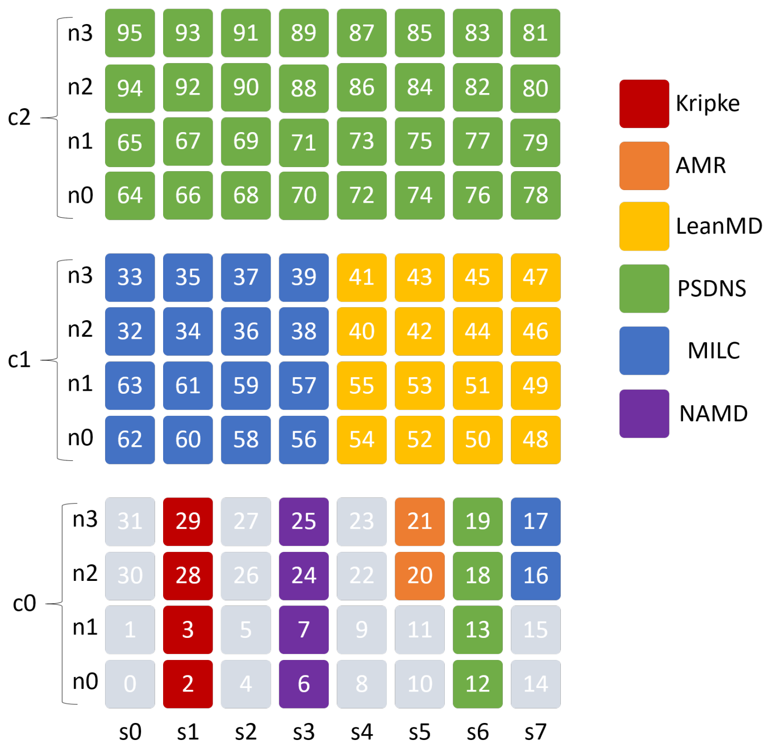


Figure 4.2: Dense Application Configuration on JYC for Fault Injection Campaign 2. Grayed-out nodes are unused nodes.

node of the JYC system (grayed-out nodes are unused nodes or service nodes that cannot be used by users). The change from five different configurations to three is simply due to time constraints and the need to run the same experiments repeatedly. These same applications and configurations used in Campaign II are reused in Campaigns III and IV to maintain consistency and repeatability. Refer to Appendix D for the other two configurations constructed from this set of applications.

4.1.2 Injection Selection Method

Random Selection

Faults in Campaign I were injected on randomly selected components. HPCArrow's Injection Manager monitors the system job queue and assembles a list of jobs currently running. It randomly selects a job from the list and then, depending on the fault type and the workload size, it randomly selects

Table 4.2: Random injection selection method for each fault type, which depends on the number of nodes an application is running on.

# of Nodes	# of Geminis	Fault Type Link	Connection	Blade
2	1	Random outgoing link from Gemini	Connection in random direction from Gemini	Blade containing the Gemini
4	2	One random link of links between the 2 Geminis	Random connection direction of connections from either Gemini	Blade containing the 2 Geminis
8	4	One random link of links between 2 blades where workloads are running	Random connection direction of connections between 2 blades where workloads are running	Random blade of 2 blades where workloads are running
32	16	One random link of links between any 8 blades where workloads are running	Random connection direction of connections between 8 blades where workloads are running	Random blade of 8 blades where workloads are running
64	32	One random link of links between 2 chassis	Random connection direction of connections between 2 chassis	Random blade of 2 chassis' blades (16 total)

a component on which the targeted job is running. Table 4.2 enumerates the selection process for each fault type. For example, if the user specifies a link injection, then a random job is selected. If the targeted job is running on four nodes, then it covers two Geminis' worth of links. A random link from among the links between the two Geminis is selected for take-down or injection.

Traffic-Based Selection

For Campaigns II, III, and IV, the selection process of a target component was based on the maximum utilization of network components over the course of five no-injection runs of an application configuration. This methodology focuses solely on the components that utilize the most network resources. This ensures that we choose the fault injections with maximal impact on the system and on the applications running on it.

As discussed in the previous chapter, LDMS logs traffic throughput (in

Table 4.3: Targeted injection selection method for each fault type, which is only dependent on traffic data, not number of nodes or Geminis.

# of Nodes	**# of Geminis**	**Fault Type**		
		Link	**Connection**	**Blade**
-	-	Link of connection with highest throughput	Connection with highest throughput	Blade attached to connection with highest throughput

bytes/second) in the X+, X-, Y+, Y-, Z+, and Z- directions[2] for each Gemini on the target system. Each of the three workload configurations were run without injections and profiled, using the LDMS data, to determine which applications and corresponding connection directions generated the most traffic on the network. On a time series plot of the LDMS data, connection components with the maximum throughput activity can be visually identified and selected for injection.

For links and blades, the methodology is similar. While LDMS data only tracks traffic at the connection direction granularity, the same traffic data can easily be re-used and extended to determine which links and blades to target. In particular, the links that form the connection with the highest throughput are target candidates; the blades that are attached to the connection with the highest throughput are target candidates. Table 4.3 enumerates the selection process for each fault type. This method is independent of the workload size or number of Geminis, but it must be computed offline from golden outputs. The Injection Manager does not do anything beyond monitor the job queue and trigger the fault injection whose injection type, fault type, and component name are user-specified. Future work should involve real-time monitoring of live LDMS data and real-time component selection based on the workloads running on the HPC system.

To illustrate, we discuss a NAMD Charm++ application running on 16 nodes. Figure 4.3 shows time series plots of the traffic throughput over the course of the workload's execution on each of the 16 nodes (the graph is truncated to 6 nodes for readability). Note that since JYC is a one cabinet system, the plots leave out the X+/- connection directions. The target

[2]LDMS only tracks at the granularity of connections. It does not track utilization of specific links themselves.

component can be identified by visually finding the node with the connection direction that has the highest sustained traffic throughput. In this case, there are two pairs of candidates: nodes nid00032 and nid00033, corresponding to Gemini c0-0c1s0g1, as well as nodes nid00060 and nid00061, corresponding to Gemini c0-0c1s1g0, show the highest sustained throughput in the Z+ and Z directions, respectively. Since they are roughly similar, either Gemini is a suitable target candidate. Here, Gemini c0-0c1s0g1 is arbitrarily chosen, which means the Z+ connection (c0-0c1s0g1,Z+) is targeted for the connection fault injection on this NAMD application. Since the Z+ connection has eight possible links, three links are randomly pre-selected: in this case, we chose c0-0c1s0g1l01, c0-0c1s0g1l10, and c0-0c1s0g1l27. If it were the Y+ direction, which has four links, three links would still randomly be chosen as well. For the node injection, one of the two nodes is arbitrarily chosen. And finally, for the blade injection, the blade that houses the Gemini c0-0c1s0g1 is targeted, which in this case is c0-0c1s0. See Appendix E for all the components that were targeted using this traffic-based selection method.

4.1.3 Experiment Timeline

In a fault injection campaign, running an experiment requires a couple pre-steps. Prior to execution, a user configures and builds an application set through HPCArrow's Workload Manager module. The user specifies the injection type, fault type(s), component name(s), recovery target stage (if needed), and timing delay (if needed) through the applications' job names.

Figure 4.4 illustrates the timeline of an experiment once preconfiguration is completed and the applications begin to run. At time $T = 0$, the user-selected applications are submitted to ALPS to be run on JYC. The Fault Injector module waits until F seconds have elapsed before injecting the first fault. F is composed of two delays: $F = P + D$. For our experiments, P is pre-set by us to be 120 seconds. This is to allow time for all the applications to complete initialization and settle into a steady state flow of computations. The injector then waits an additional D seconds, a delay that is by default 100 seconds or specified by the user in the job name. For all campaigns, except for Campaign III, we leave the delay as its default value. After this, the user-specified fault is injected. After R seconds have elapsed, the failover recovery

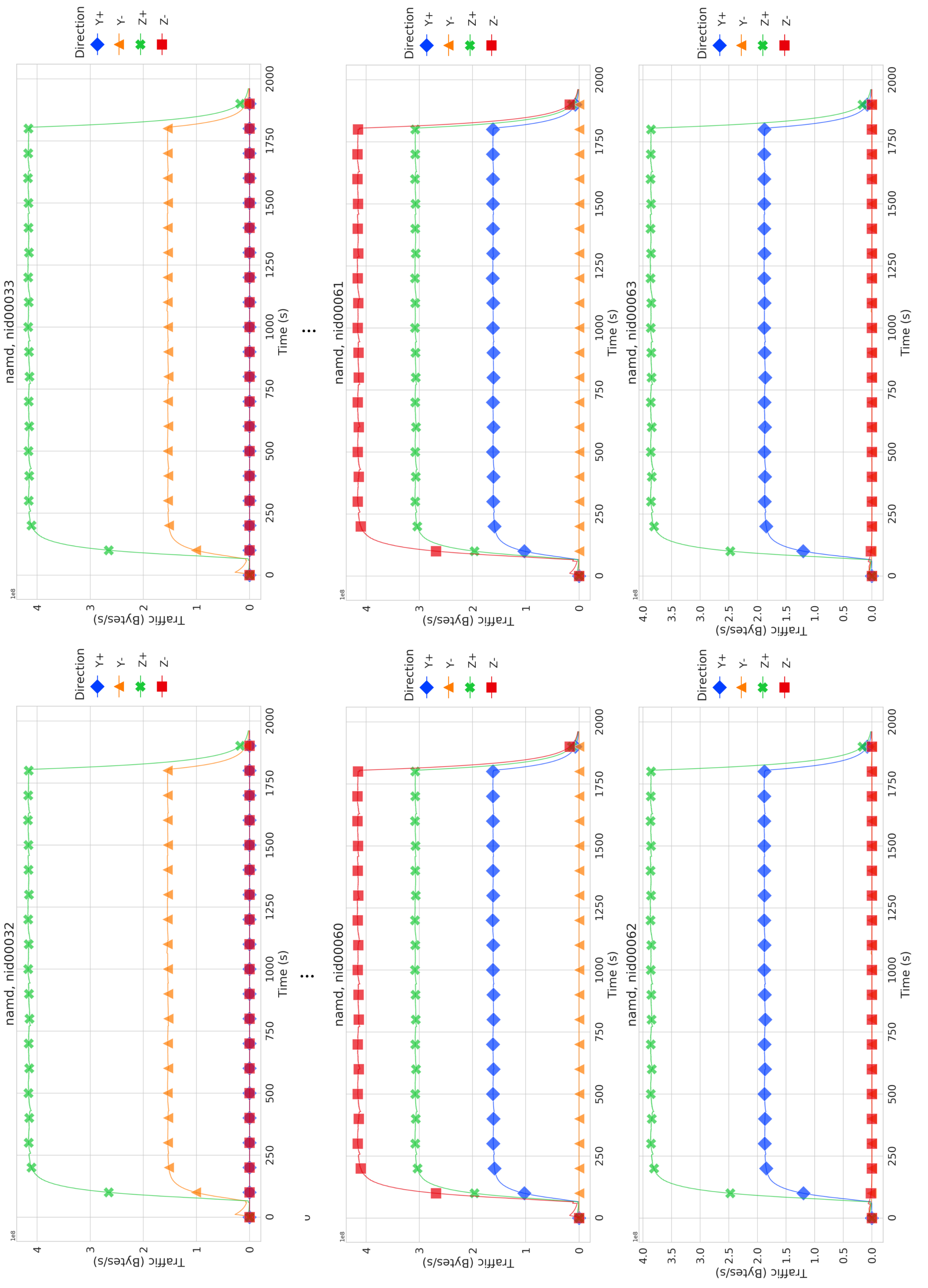


Figure 4.3: LDMS traffic data for every connection direction is plotted for each node on which a NAMD Charm++ application using 16 nodes (truncated to six for readability) runs. The traffic based selection method chooses the highest throughput connection. Such candidates are on nodes nid00032, nid00033, nid00060, and nid00061.

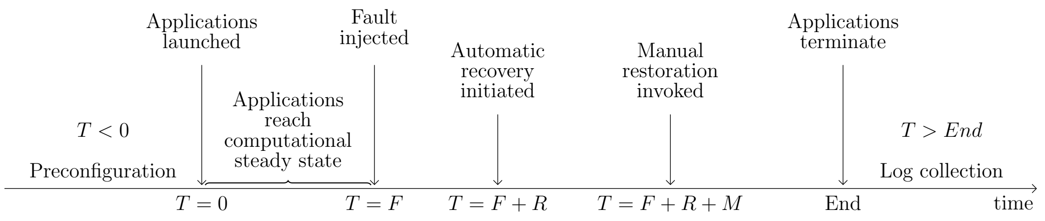


Figure 4.4: A general single injection experiment timeline for Campaigns I-III.

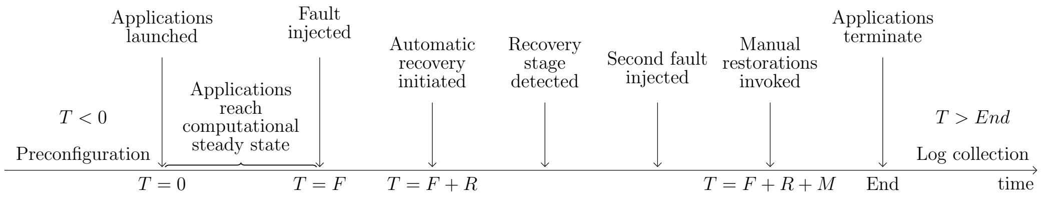


Figure 4.5: A general injection during recovery experiment timeline for Campaign IV.

procedure is automatically triggered by the SMW. R is the time it takes for the SMW to respond to the fault and is not configurable from our end. It is usually instantaneous, taking no more than a second for failover to begin. The failover computes new routes to reroute around the fault while quiescing the HSN. It installs the new routes and then finally unquieces the HSN. An application may crash, hang, or continue as usual during failover. Failover usually takes about 30 seconds. In the case of an injection during recovery experiment, the injector watches for a target recovery stage in the network logs and injects a second fault upon detection during this 30 second window as shown in Figure 4.5. At time $T = F + R + M$, the Fault Injector module invokes the manual restore procedure to re-initialize the failed component and warm-swap it back into service. During the warm swap procedure, new routes are computed, the HSN is quiesced, the new routes are installed, and the HSN is unquiesced again. M is pre-set by us to be 200 seconds, which gives the failover ample time to complete. The experiment ends when all jobs in the application set terminate, usually between 30-40 minutes or up to two hours (in the case of a hung application).

4.2 Fault Injection Campaign I

The purpose of Campaign I[3] was to serve as a proof of concept for our fault injection methodology and to observe JYC's behavior and resiliency to the simplest of failures, single fault injections. Target components were randomly selected as discussed in previous sections. Six different benchmark applications from Application Set I were run during these experiments. The experiences and results gained from this set of experiments helped to refine the methodology and experimental setup for later campaigns.

4.2.1 Summary Results

In Campaign I, there was a total of 63 experiments on JYC. Out of 63, 18 were baseline runs without injections. The remaining 45 were injection experiments: 13 link injections, 16 connection injections, and 16 blade injections.

Recall that the first step of the random selection method is to randomly select an application before randomly selecting a component. Each fault is thus injected into a component that is utilized by an application A. From the perspective of application A, we call this a direct injection. Thus, an indirect injection would be a fault that is injected on a component that application A is not directly utilizing.

In the presence of faults, direct or indirect, applications can behave in various ways. In this work, we focus on an application's run status and run time. An application's run status refers to whether the application completed its computations without impact. Specifically, the outcomes are (1) crash, which is when an application terminates prematurely or abnormally, (2) hang, which is when an application makes no forward progress, (3) silent data corruption, which is when the application runs to completion, but its outputs do not match the golden outputs, or (4) no impact.

Table 4.4 and its corresponding plot in Figure 4.6 show only the application run statuses as a result of direct injections. All applications in this campaign experience no impact as a result of indirect injections. Thus we focus here only on direct injections. See Appendix F for the full summary, including indirect faults.

[3]Development and execution of Campaign I was a joint effort with Lavin Devnani, who also presents the work from Campaign I in his thesis [36].

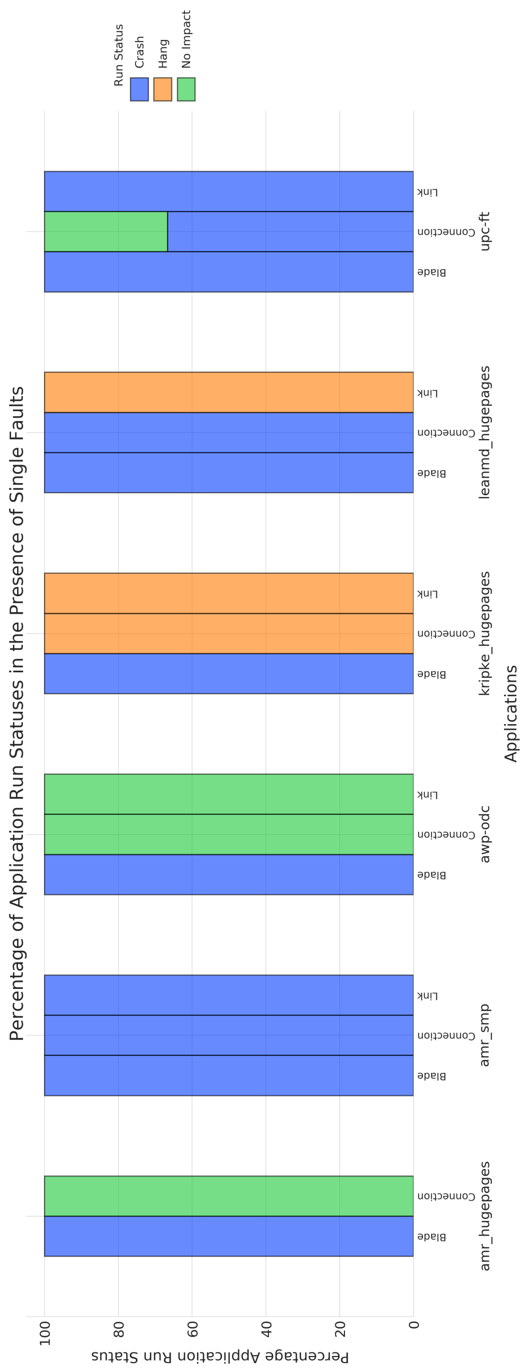


Figure 4.6: Summary percentages of Campaign I's direct injections and application run statuses, grouped by fault type and benchmark. Refer to Table 4.4 for counts.

Table 4.4: Summary of Campaign I, showing the outcomes (Crash, Hang, No Impact) of applications that are directly injected with various fault types (Blade, Connection, Link). See Appendix F for the full summary, including indirect faults.

Benchmark	**Application**	**Fault Type**	**Run Status #(%)**		
			Crash	Hang	No Impact
Charm++ (HugePages)	AMR	Blade	2 (100.0)	0	0
		Connection	0	0	2 (100.0)
	Kripke	Blade	1 (100.0)	0	0
		Connection	0	1 (100.0)	0
		Link	0	2 (100.0)	0
	LeanMD	Blade	3 (100.0)	0	0
		Connection	2 (100.0)	0	0
		Link	0	1 (100.0)	0
Charm++ (SMP)	AMR	Blade	5 (100.0)	0	0
		Connection	5 (100.0)	0	0
		Link	4 (100.0)	0	0
MPI	AWP-ODC	Blade	3 (100.0)	0	0
		Connection	0	0	3 (100.0)
		Link	0	0	3 (100.0)
PGAS	UPC-FT	Blade	2 (100.0)	0	0
		Connection	2 (66.67)	0	1 (33.33)
		Link	3 (100.0)	0	0

4.2.2 Failure Scenarios

In this campaign, we covered single fault types from link to blade. While components are chosen randomly, we still repeat experiments. For each of the five application configurations from Set I, we run each of the fault injection types three times each. In this section, we present case experiments for a single injection on a compute component and a single injection on a network component. For each case, we examine resiliency behaviors at the application, network, and system levels.

Blade and Node Faults

For all blade and node injections, it is expected that an application using the targeted blade or node will crash simply because one or more compute components are not available. These fault injection outcomes are reflected in Table 4.4 and Figure 4.6 in which every direct blade injection always led to an application crash. We present a case blade fault experiment here.

In Experiment 42, blade c0-0c2s5 was targeted. Recall that when a blade is taken out of service, the two Gemini ASICs (c0-0c2s5g0, c0-0c2s5g1) housed by the blade fail as well, causing a cascade of link failures across all outgoing links and links connected at the other ends. Thus 40 link failures were expected to be observed.

This blade fault directly impacted and prematurely terminated the 32-node AWP-ODC application running on the top chassis. All other applications continued to run to completion as normal. The targeted application's outputs reported several instances of `Generic TCP Error` across nodes 74, 75, 84, and 85. The following message was also logged, which is typically generated when a node or blade fails with some hardware error (e.g. memory check error) [37]:

```
[NID 00064] 2018-03-15 21:50:03 Apid 614720 killed. Received node
    event ec_node_failed for nid 74
```

Figures 4.7 and 4.8 illustrate the timeline of events at the experiment level (application and injection timings and durations) and at the network level, respectively. From $t = 0$ to $t = 341.205$, there was a sustained flow of traffic across all eight blades that AWP-ODC was running on as shown by the blue line in Figure 4.8a. At $t = 341.205$, the blade fault was injected, which caused Gemini c0-0c2s5g0 and c0-0c2s5g1 to fail in every connection direction. This caused the application to die as can be observed more clearly in Figure 4.8b when all traffic died after the fault injection. In response, all network traffic was quiesced while the automatic failover rerouted around the failed blade. This failover completed at $t = 391.0$. Traffic for the network resumed across the system for non-impacted applications except for the Geminis that the killed application had been running on. At $t = 543$, the injector initiated the warm swap procedure, which completed at $t = 967.732$ and successfully returned the blade back into service. The spike was likely residual traffic leftover from before the fault.

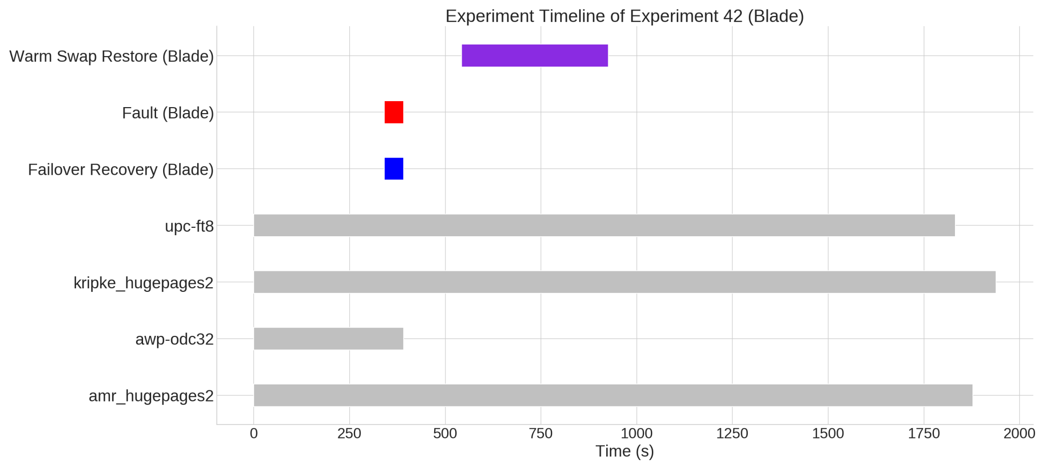
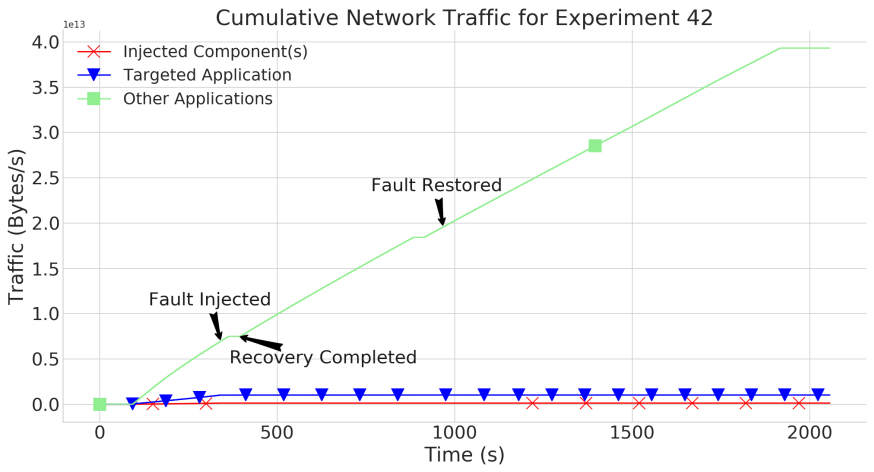


Figure 4.7: Timeline of Experiment 42 with a single blade injection. The digits after each application name indicate the number of nodes the application ran on.

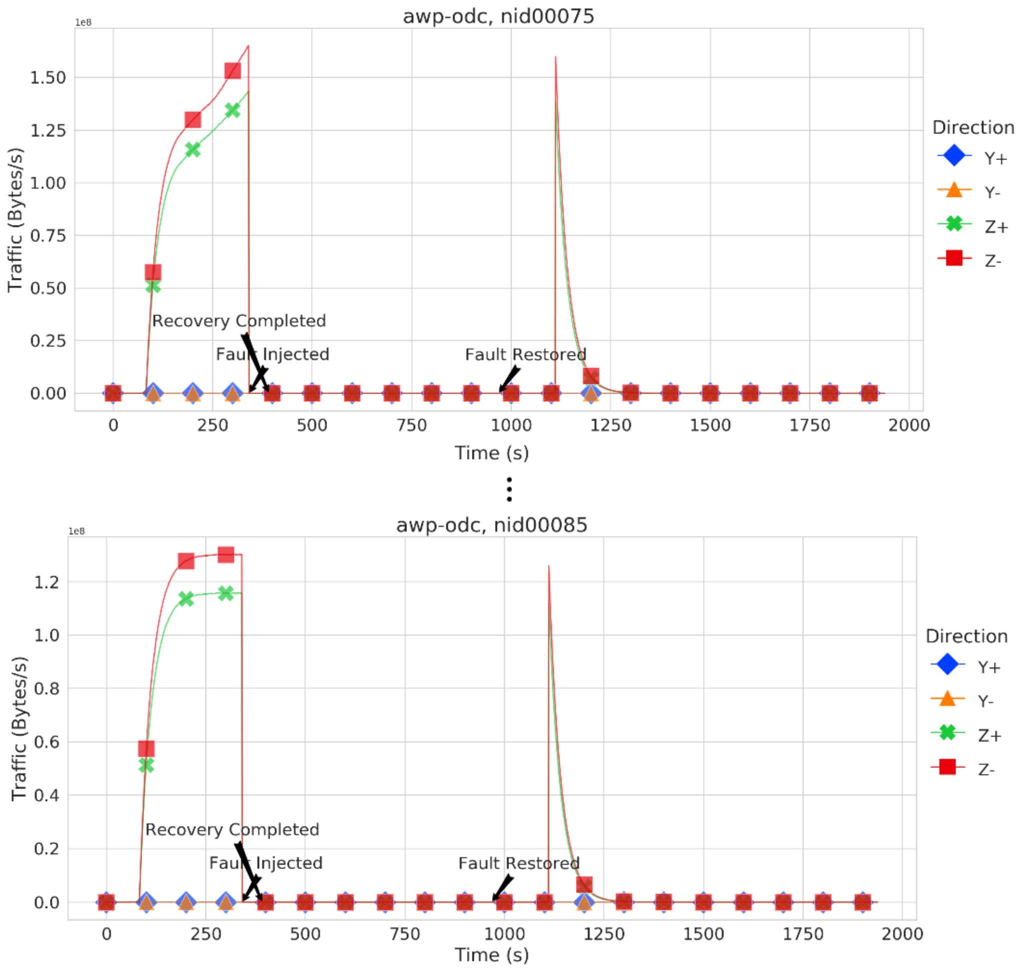
Examining network recovery logs as shown in Table 4.5 confirmed the normal completion of the failover recovery and the warm swap restoration. There were 40 link failures as expected, one successful link recovery (all links are handled in one recovery process), and one successful warm swap. From the hardware error logs, there were a variety of errors, some with an enormous count. The SSID Request Timeouts, SSID Response Protocol, SSID Detected Misrouted Packet, ORB RAM Scrubbed, NIF Squashed Request Packet, and NW (netwatch) errors all indicate network packets being dropped or misrouted intentionally by the routing algorithm due to the blade failure. All of these errors were transient and disappeared once failover successfully completed, leaving no indication of any critical problems in the system and network.

Link and Connection Faults

For single link and single connection faults, outcomes of application run statuses were more varied, especially for Charm++ and PGAS applications. Crashes were observed among AMR (SMP), LeanMD, and UPC-FT. Even more surprising, hangs were observed for link and connection injections into Kripke and LeanMD. In this section, we present a case experiment of a single link injection that results in an application crash. We leave examination and discussion of the hang case for the next section on Campaign II.



(a) Cumulative traffic plot of Experiment 42 with a single blade injection.



(b) Traffic plot of a 32-node AWP-ODC application run in Experiment 42, with focus on two out of the four nodes on the injected blade (left out for readability and space).

Figure 4.8: Traffic plots of Experiment 42 with a single blade injection.

Table 4.5: Summary of system logs for Experiment 42 (single blade fault).

Event	Count
Link Failed	40
Link Recovery Successful	1
NetUnthrottle	2
Quiesced	1
SetThrottleMask	2
Throttle	2
Unquiesced	1
Warm Swap Successful	1
ORB RAM Scrubbed Lower Entry	24
ORB RAM Scrubbed Upper Entry	24
SSID Detected Misrouted Packet	52
SSID Request Timeout	7497
SSID Response Protocol	105
NIF Squashed Request Packet	44
NW Exchange from SUPR to RUN	10
NW Link Went Inactive	40
NW Send EOP Bad	10

In Experiment 46, link c0-0c2s0g0l17 was targeted. Recall that when a link is taken out of service, the link on the other end (c0-0c2s7g0l27) is also taken down. Thus we expected two links to fail in this experiment.

This link fault directly impacted and prematurely terminated the 32-node UPC-FT application running on the top chassis. All other applications continued to run to completion as normal. The targeted application's outputs reported instances of `DMAPP_RC_TRANSACTION_ERROR: Transaction failed` across nodes c0-0c2s0n0 and c0-0c2s7n2, which contain the failed links as shown below:

```
PE 480: ERROR: dmapp_syncid_wait( hdl ): DMAPP_RC_TRANSACTION_ERROR
    : Transaction failed.
PE 15: ERROR: dmapp_syncid_wait( hdl ): DMAPP_RC_TRANSACTION_ERROR:
     Transaction failed.
[NID 00064] Apid 618722: initiated application termination
```

This error means that the network transaction completed with an error state, either a non-recoverable transaction error or a transient error, such as network error, uncorrectable memory error, or resource shortage [38]. In short, the link fault naturally caused transactions to fail. The rest of the log messages record the application crash.

Figures 4.9 and 4.10 illustrate the timeline of events at the experiment level and at the network level, respectively. From $t = 0$ to $t = 440.584$, there was

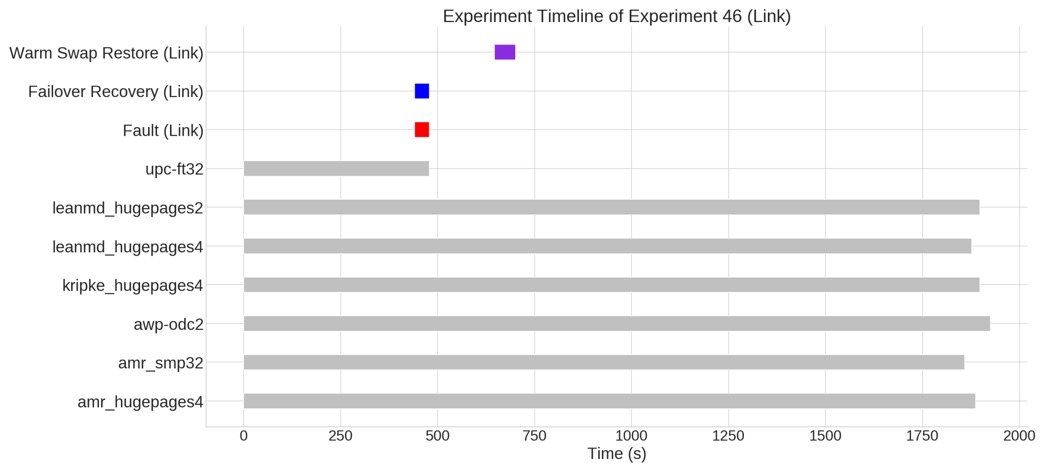
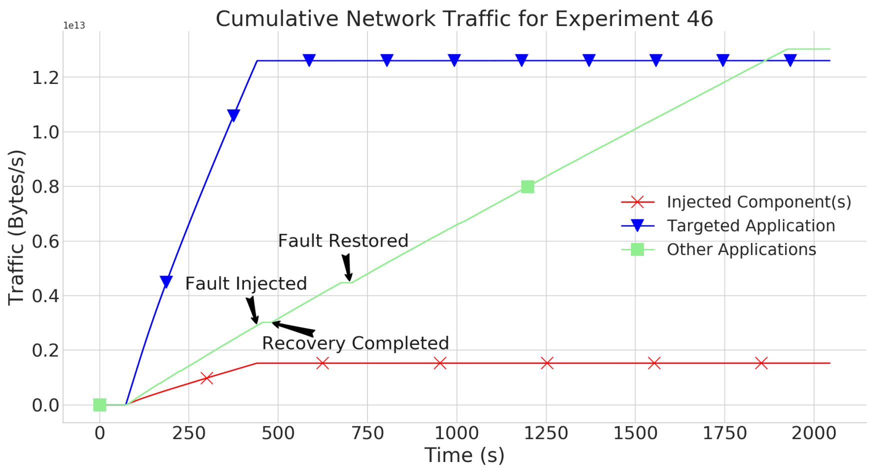


Figure 4.9: Timeline of Experiment 46 with a single link injection.

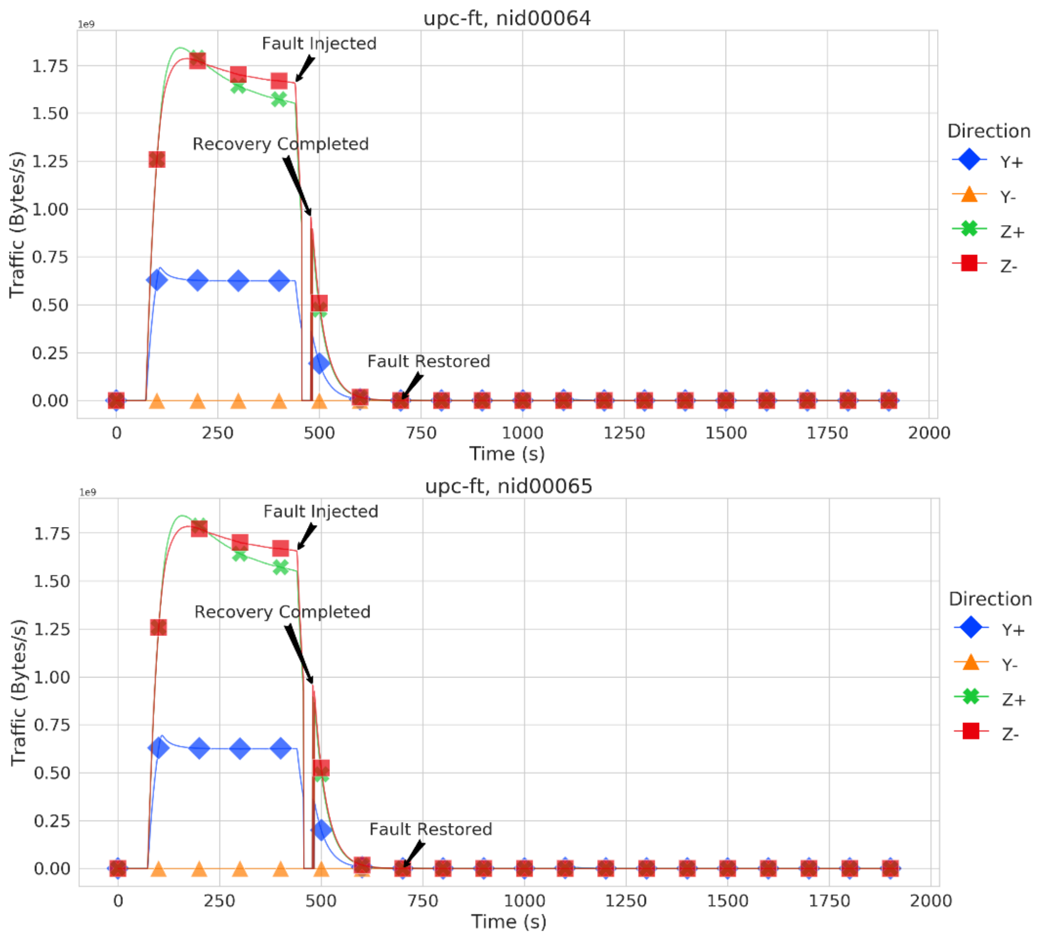
a sustained flow of traffic across all eight blades that UPC-FT was running on as shown by the blue line in Figure 4.10a. At $t = 440.584$, the link fault was injected, which caused the application to die as shown by the red and blue lines in Figure 4.10a[4] flattening. This is more apparent when looking at traffic for the Gemini (two nodes) whose link was injected on in Figure 4.10b. In response to the fault, all network traffic was quiesced while the automatic failover rerouted around the failed link. This failover completed at $t = 479.0$. Traffic for the network resumes across the system for non-impacted applications. The warm swap procedure completed at $t = 700.65$ and successfully returns the blade back into service.

Examining network recovery logs as shown in Table 4.6 confirmed the normal completion of the failover recovery and of the warm swap restoration. There were two link failures as expected, one successful link recovery, and one successful warm swap. From the hardware error logs, we observed errors similar to those in the blade fault experiment discussed above. However, since there were only two link failures, the frequency of these events are much less in this case. Overall, the network and system recovered without any residual critical errors.

[4]It may seem odd that the cumulative traffic plot shows that the single UPC-FT application had more overall traffic than the rest of the applications. This is due to many nodes not having the LDMS data collector enabled by the system administrator at the time.



(a) Cumulative traffic plot of Experiment 46 with a single link injection.



(b) Traffic plot of a 32-node UPC-FT application run in Experiment 46, with focus on one Gemini (two nodes shown) whose link was targeted for injection.

Figure 4.10: Traffic plots of Experiment 46 with a single link injection.

Table 4.6: Summary of system logs for Experiment 46 (single link fault).

Event	**Count**
Link Failed	2
Link Recovery Successful	1
NetUnthrottle	2
Quiesced	1
SetThrottleMask	2
Throttle	2
Unquiesced	1
Warm Swap Successful	1
ORB RAM Scrubbed Lower Entry	9
ORB RAM Scrubbed Upper Entry	9
SSID Request Timeout	69
SSID Response Protocol	1
NW Exchange from SUPR to RUN	1
NW Link Went Inactive	2

4.3 Fault Injection Campaign II

Campaign I was the proof of concept for our fault injection approach and for our injector tool HPCArrow. However, the randomness of experiments and the limited number of experiments provided little sense of predictability in terms of application and system behavior, especially in the presence of link and connection faults. Our research project was also expanding to consider other more modern HPC systems, such as the Aries interconnect. Thus, there are two main reasons for this second fault injection campaign.

The first reason is related to the modification of the injection selection methodology from random (Campaign I) to network traffic-based (Campaign II and onward). This provided two benefits. (1) Fault injection on the components with the highest throughput traffic ensures that such faults will cause the most impact on the system. (2) This traffic-based selection method led to further development of HPCArrow's capabilities to allow for user-specified target components. This in turn created a better mechanism for reproducible experiments as we can now rerun the exact same scenarios repeatedly. Unlike in Campaign I, where each "repeated" experiment can still vary randomly in the targeted application and exact component, in Campaign II, each injection is exactly repeated five times, down to the targeted application and component.

The second reason is the need for a more direct comparison between systems with Gemini and Aries interconnects as part of future work of this project. Maintaining as many parameters (e.g. applications, methodology, etc.) constant across systems is necessary for comparing resiliency of Gemini versus Aries interconnects. This necessity for comparison as well as for applications that generate high traffic on the HSN led to the construction of Application Set II. Applications from this set are ones that we could compile, scale, and run on both the Gemini and Aries testbeds given to us by NCSA and SNL, respectively.

4.3.1 Summary Results

There was a total of 157 experiments on JYC. Out of 157, 30 were baseline runs without injections. The remaining 127 were injection experiments: 52 link injections, 38 connection injections, and 37 blade injections.

Table 4.7 and its corresponding plot in Figure 4.11 summarize these experiments and results. Once again, they show only the application run statuses as a result of direct injections. The general observation that indirect injections have negligible impact on applications remained mostly true, except for a few cases of indirect injections causing unusual behavior. However, there is reason to believe that other factors led to these unique cases, such as secondary failures that occurred naturally in the system. Therefore, their statistics are left out of the summary table (see Appendix G for the full table summary), but we still discuss these cases in a separate section below. Additionally, all failovers and warm swaps executed in response to fault injections completed successfully. There is one case in which a warm swap on a blade failed, which is also discussed below.

4.3.2 Failure Scenarios

As in Campaign I, Figure 4.11 also shows that a direct blade injection always caused the targeted application to fail. What also stands out is that any connection injection on Charm++ applications (AMR, Kripke, LeanMD, and NAMD) always caused either a crash or hang scenario. MPI applications (MILC and PSDNS) appear to be the most resilient of the bunch, with a few exceptions for MILC, which experienced crashes due to link injections. We discuss several interesting cases and examine resiliency behaviors at the

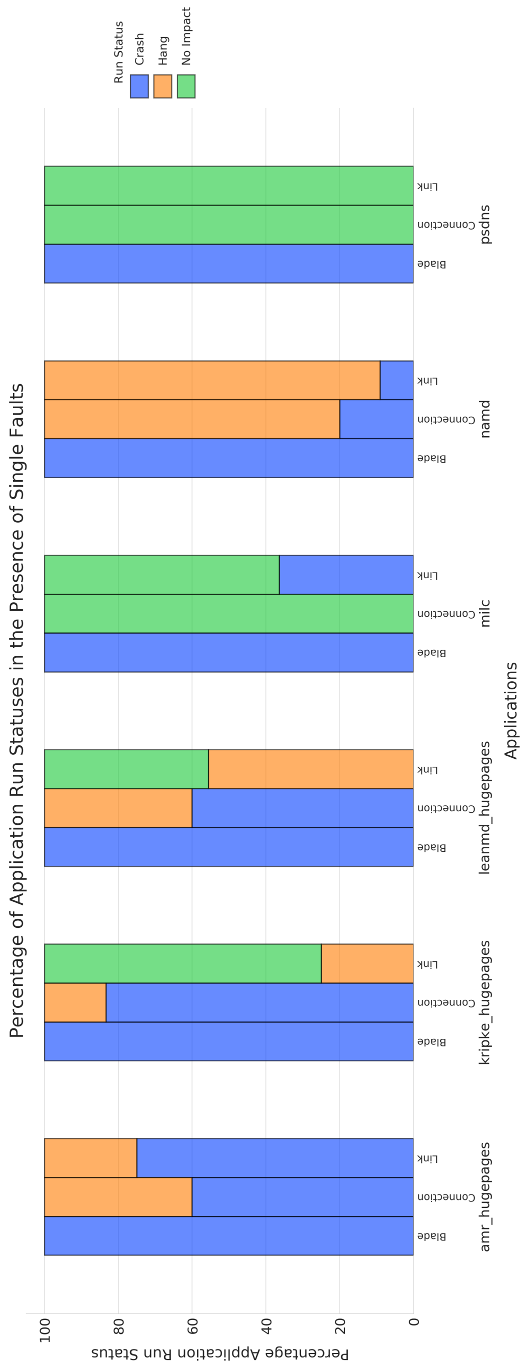


Figure 4.11: Summary percentages of Campaign II's direct injections and application run statuses, grouped by fault type and benchmark. Refer to Table 4.7 for counts.

Table 4.7: Summary of Campaign II, showing the outcomes (Crash, Hang, No Impact) of applications that are directly injected with various fault types (Blade, Connection, Link). See Appendix G for the full summary, including indirect faults.

Benchmark	**Application**	**Fault Type**	**Run Status #(%)**		
			Crash	Hang	No Impact
Charm++ (HugePages)	AMR	Blade	5 (100.0)	0	0
		Connection	6 (60.0)	4 (40.0)	0
		Link	3 (75.0)	1 (25.0)	0
	Kripke	Blade	5 (100.0)	0	0
		Connection	5 (83.33)	1 (16.67)	0
		Link	0	2 (25.0)	6 (75.0)
	LeanMD	Blade	5 (100.0)	0	0
		Connection	3 (60.0)	2 (40.0)	0
		Link	0	5 (55.56)	4 (44.44)
Charm++ (SMP)	NAMD	Blade	5 (100.0)	0	0
		Connection	1 (20.0)	4 (80.0)	0
		Link	1 (9.09)	10 (90.91)	0
MPI	MILC	Blade	5 (100.0)	0	0
		Connection	0	0	7 (100.0)
		Link	4 (36.36)	0	7 (63.64)
	PSDNS	Blade	12 (100.0)	0	0
		Connection	0	0	4 (100.0)
		Link	0	0	10 (100.0)

application, network, and system levels.

MPI: Crashes

While MPI applications AWP-ODC from Campaign I and PSDNS from Campaign II both showed strong fault tolerance against link and connection failures, the MILC application was the only MPI application to show susceptibility to link failures. It was expected for MILC to resume once failover unquiesces and finishes, but it instead crashed over one third of the time for the same reason as shown below:

```
su3_rhmd_hisq: dmapp_c_sync.c:298: _dmappi_c_process_cqe_werror:
    Assertion ‘0’ failed.
```

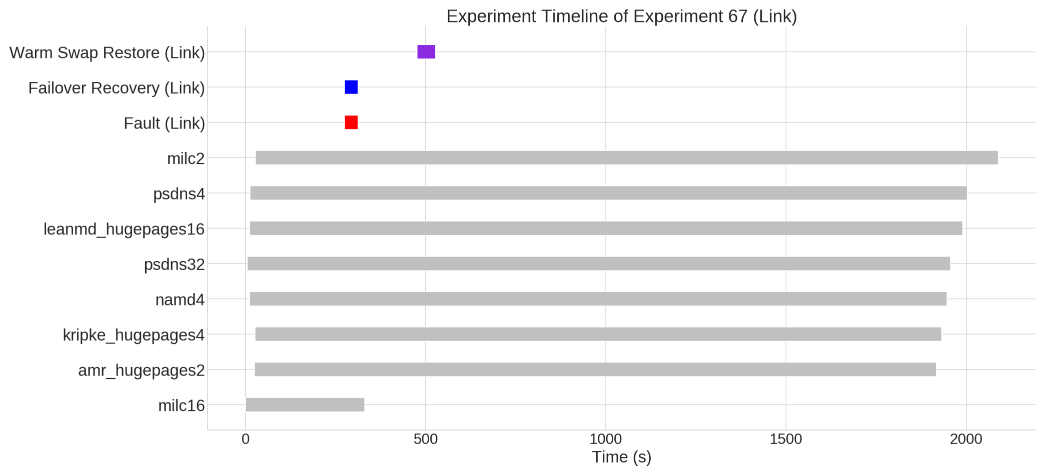


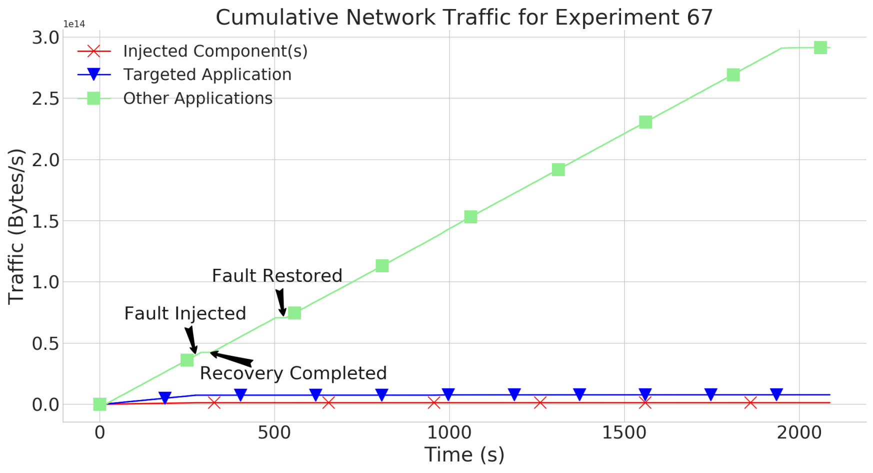
Figure 4.12: Timeline of Experiment 67 with a single link injection.

A rough stack trace produced by Abnormal Termination Processing (ATP), a Cray debugging utility, revealed the application was in the middle of an Allreduce function, preparing reduced results to distribute to all processes. A wait function was invoked to wait for completion of an RMA request and then a test function to test for request completion. It is this test function that failed. Without the source code of the the system messaging layer, it is unclear what the assertion was looking for. We can only speculate that ultimately, an RMA request could not be completed due to the link failure and this caused MILC to trip over this assertion and die.

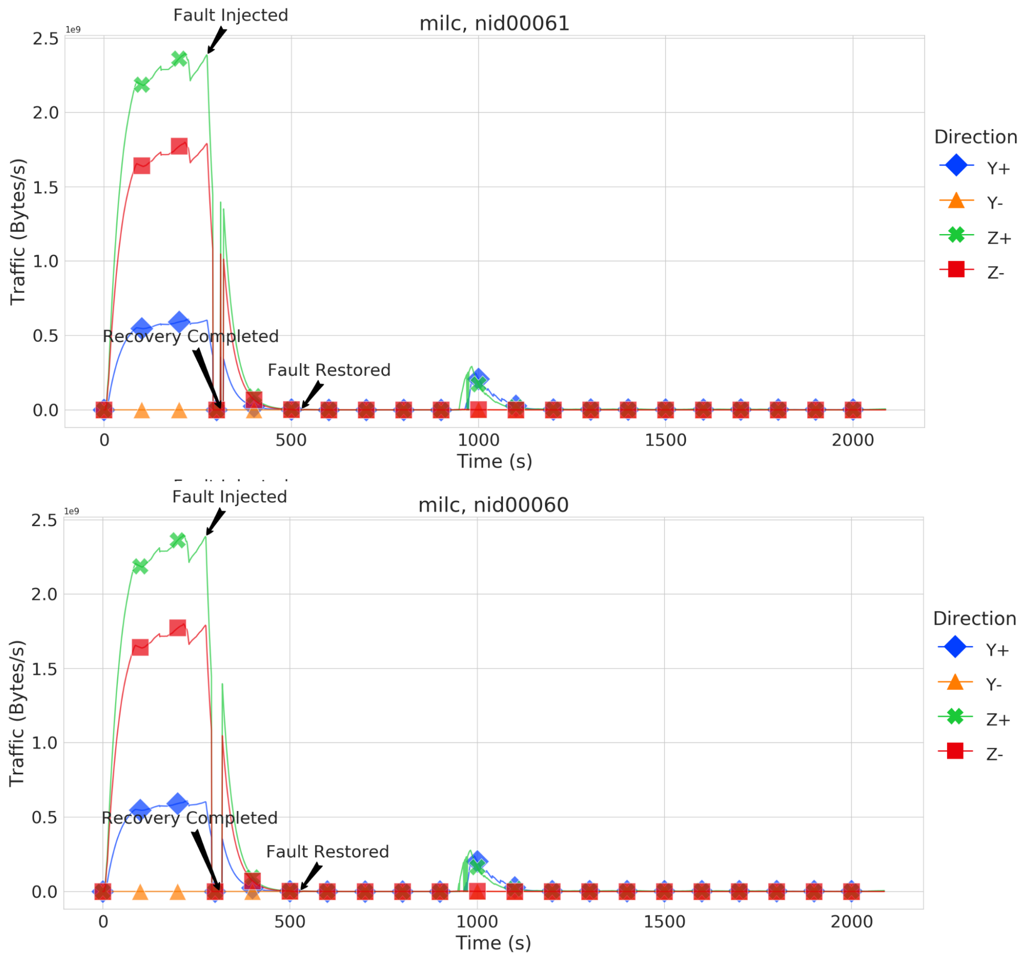
We present Experiment 67 where a 16-node MILC application was targeted by failing link c0-0c1s1g0l00. Figures 4.12 and 4.13 illustrate the timeline of events at the experiment level and at the network level, respectively. We observe traffic patterns similar to those of crashes in 4.13a, where at $t =$ 274.92, the link fault was injected, at $t = 312$, failover completed, and at $t = 526.414$, warm swap completed. The combined network recovery and hardware error logs report is shown in Table 4.8 with nothing out of the ordinary.

Charm++: Hangs vs. Crashes

Campaign I had produced scenarios in which Charm++ applications were observed to hang and make no forward progress following a link or connection fault, despite the scheduler reporting the job as "running." Campaign II



(a) Cumulative traffic plot of Experiment 67 with a single link injection targeting a 16-node MILC application.



(b) Traffic plot of a 16-node MILC application run in Experiment 67, with focus on a Gemini (two nodes shown) whose link was targeted for injection.

Figure 4.13: Traffic plots of Experiment 67 with a single link injection.

Table 4.8: Summary of system logs for Experiment 67 (single link fault).

Event	Count
Link Failed	2
Link Recovery Successful	1
NetUnthrottle	2
Quiesced	1
SetThrottleMask	2
Throttle	2
Unquiesced	1
Warm Swap Successful	1
SSID Request Timeout	4
NW Exchange from SUPR to RUN	5
NW Link Went Inactive	2
Sender Packet Timeout	6
Receiver EOP Bad	3
Receiver CC1 Bad	3
Receiver PIC Error	3

reproduced and corroborated these observations with repeated experiments.

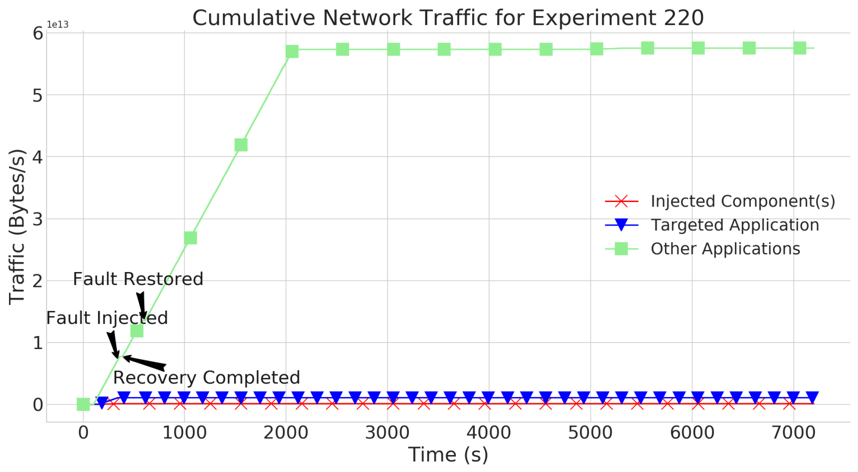
In some hang cases for NAMD, such as in Experiment 220, NAMD did not output any error messages. In other cases, it continuously wrote the following warning to its logs until NAMD was terminated by ALPS for hitting the Walltime:

```
Warning: GNI_PostRdma: ioctl(GNI_IOC_POST_RDMA) returned error -
    Invalid argument at line 161 in file rdma_transfer.c
```

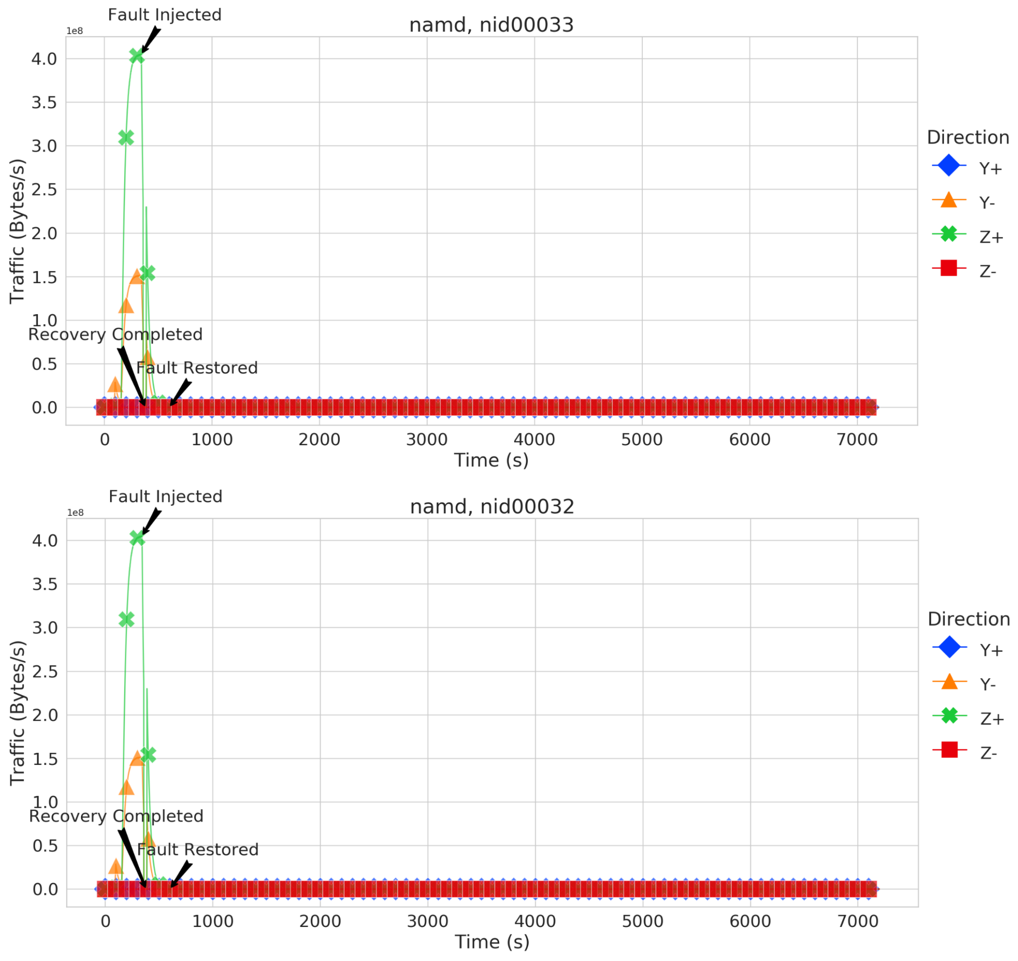
The Charm++ runtime framework logs this warning when an invalid argument is encountered in a RDMA transaction. However, since this was a warning and did not trigger any abnormal terminations in the way that crashes do, the ATP utility did not provide any stack traces. In the one NAMD crash scenario due to a connection fault, the ATP reported a similar error: `[498] registerFromMempool; err=GNI_RC_INVALID_PARAM`.

NAMD traffic as shown in Figure 4.14 died after the link injection (c0-0c1s0g1l01) at $t = 342.004$ and never resumed following failover completion at $t = 380$ or warm swap completion at $t = 603.834$. This pattern mirrored that of crashes. The traffic plots show up to 7200 seconds (two hours) to highlight that the reported running time of NAMD was two hours, as depicted in Figure 4.15, yet there was no traffic over the network for the majority of this time. Hardware error logs are similar to previously discussed examples. For completion, it is still provided in Table 4.9.

During hangs and crashes, Kripke also sometimes reported the same exact



(a) Cumulative traffic plot of Experiment 220 with a single link injection targeting a 16-node NAMD application.



(b) Traffic plot of a 16-node NAMD application run in Experiment 220, with focus on a Gemini (two nodes shown) whose link was targeted for injection.

Figure 4.14: Traffic plots of Experiment 220 with a single link injection.

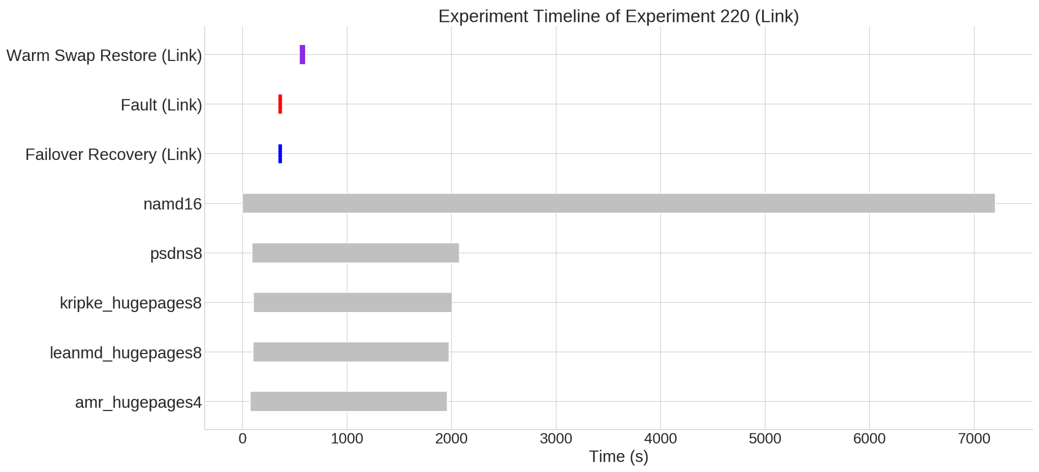


Figure 4.15: Timeline of Experiment 220 with a single link injection.

Table 4.9: Summary of system logs for Experiment 220 (single link fault).

Event	**Count**
Link Failed	2
Link Recovery Successful	1
NetUnthrottle	2
Quiesced	1
SetThrottleMask	2
Throttle	2
Unquiesced	1
Warm Swap Successful	1
ORB RAM Scrubbed Lower Entry	3
ORB RAM Scrubbed Upper Entry	3
SSID Request Timeout	37
NW Exchange from SUPR to RUN	1
NW Link Went Inactive	2

warning as hung NAMD applications. In one scenario, Kripke crashed due a segmentation fault during a memory copy invoked from the Charm++ PUP (Pack/UnPack) library, which is used to pack an array, structure, or object into a memory buffer.

```
PUP::fromMem::bytes(void*, unsigned long, unsigned long, PUP::
    dataType)@pup_util.C:165
__cray_memcpy_INT@0x2042d315
ATP Stack walkback for Rank 74 done
Process died with signal 11: 'Segmentation fault'
```

LeanMD hang instances behaved similarly to NAMD hangs in terms of `GNI_PostRdma` warning messages and incomplete computational step outputs. None of the crashes due to connection faults produced a stack trace to analyze.

AMR (HugePages) hangs produced no output. However, its crash messages varied wildly, such as the memory copy segmentation fault seen for Kripke. AMR also tripped over an assertion as shown below:

```
Reason: Assertion "msg_nbytes > 0" failed in file machine.c line
    2239.
aborting job:
```

This assertion failure was due to a SMSG send failure, meaning a network packet had been dropped in-flight to its destination Gemini. AMR (SMP) crash instances all reported this same error as well.

Across all of our analysis methodologies, there was no indicator to differentiate between a hang or a crash scenario. The only telltale sign that an application had hung was that the ALPS continued to report the application as “running” while no further computations were logged in the application’s outputs. Eventually when the maximum allowed time had elapsed, the ALPS terminated the job. This creates an interesting detection problem in that from the system side, there is no obvious indication of a hung application.

Indirect Injections

Campaign II is the only campaign with failure scenarios in which an application failed even though it was not a direct target of an injection. The only application to exhibit this behavior was PSDNS.

We present Experiment 85, which used the set of applications and placements from the Sparse Configuration. In this experiment, a link injection on c0-0c2s0g1l42 directly targeted a four-node AMR application in the top chassis. While AMR crashed as result, a four-node PSDNS application placed in a different chassis, the middle one, also crashed before completing its computations. The error reported in its logs is simply:

```
lib-4205 : UNRECOVERABLE library error
The program was unable to request more memory space.
```

This error message indicated that during one of PSDNS computational steps, it tried to allocate memory, but was evidently unable to request more. Sometimes this might indicate that the application's problem size is too large to be handled by four nodes. However, this application had been run close to 100 times throughout this campaign, including ten baseline runs and four direct injection runs, without premature terminations. Only five such instances of `UNRECOVERABLE library error` occurred and none were caused by direct injections. So this error was unlikely solely due to the problem size itself.

Since this was an indirect injection, we present only PSDNS's traffic plot as shown in Figure 4.16 and its system report in Table 4.10. While PSDNS may appear to complete within the expected time window of 30-40 minutes, it does indeed crash at around $t = 2087$ according to the ATP stack trace and PSDNS's log outputs. However, the experiment timeline as shown in Figure 4.17 reports the time of link injection at $t = 290.049$, the time of failover completion at $t = 329$, and the time of warm swap completion at $t = 544.498$. All of these events are far removed in time from the time of PSDNS's crash. Similarly, in the other four instances of PSDNS crashing, the application crashed very early in its run (about 80-90 seconds) with the same `UNRECOVERABLE library error`, even before a fault injection had occurred. These four instances were repeated in back-to-back runs of the same link injection experiment in the same night.

It may be possible that the link injections had no causal impact that led to PSDNS's crashes and that there may be some other underlying issue that was not reported to the system. It may also be possible that memory was not properly freed, causing memory across four nodes to run out; it is unclear at this point what might lead to such behavior. Future work should investigate

Table 4.10: Summary of system logs for Experiment 85 (single link fault).

Event	**Count**
Link Failed	2
Link Recovery Successful	1
NetUnthrottle	2
Quiesced	1
SetThrottleMask	2
Throttle	2
Unquiesced	1
Warm Swap Successful	1
ORB RAM Scrubbed Lower Entry	7
ORB RAM Scrubbed Upper Entry	7
SSID Request Timeout	297
NW Exchange from SUPR to RUN	1
NW Link Went Inactive	2

collecting data on memory utilization, similar to traffic data for the network. We should also develop a fault model for memory congestion at the node level.

Warm Swap Failure

Lastly, we present a case in which a warm swap restoration for a blade injection failed to bring a blade back into service. Running applications from the Medium Configuration, Experiment 160 injected a blade fault on blade c0-0c2s5 at $t = 276.492$, targeting an eight-node Kripke application. At $t = 319$, the failover completed and at $t = 858$, the warm swap process terminated with a failure.

From our own injector's logs, the following is reported for the blade warm swap commands:

```
ERROR: xtbounce command to initialize links timed out
ERROR: Timeout during xtbounce link initialization
FAILURE: Warm swap command failed.
```

The warm swap add command had failed due to an xtbounce that was meant to initialize the links on the blade that was being added back into service. The xtbounce timed out instead, causing the whole process to fail. A variant xtbounce command had to be called manually on the SMW by the system administrator in order to successfully warm swap the blade back into service.

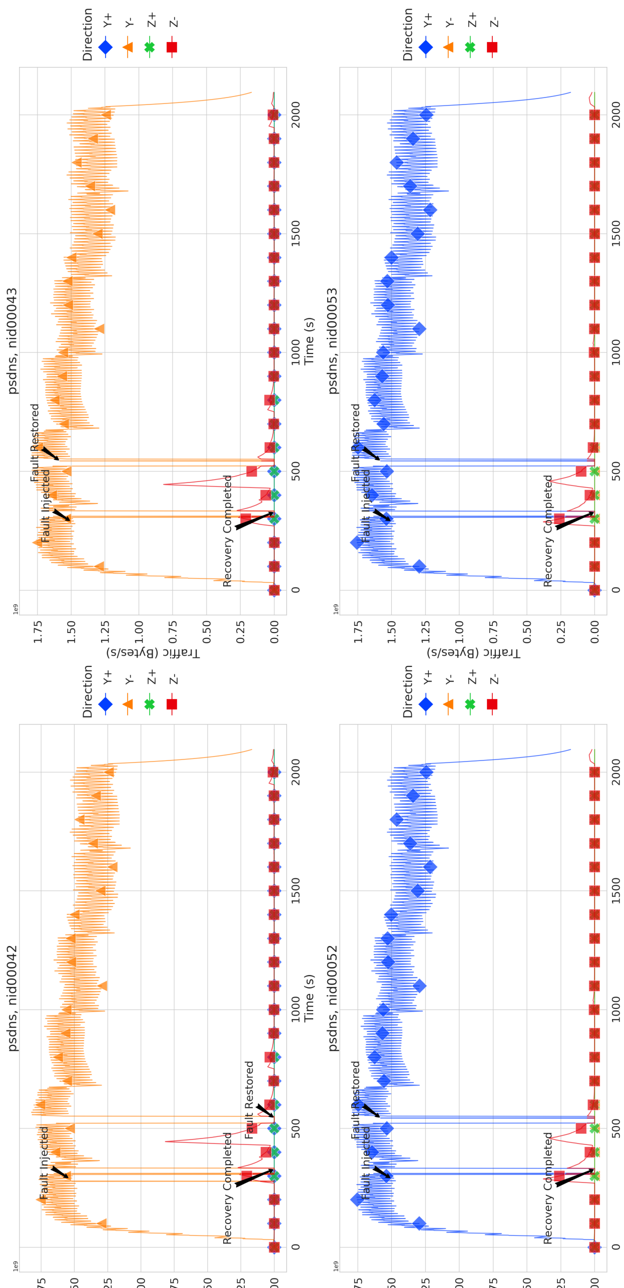


Figure 4.16: Traffic plot of a four-node PSDNS application run in Experiment 85 that was not a target of the single link injection.

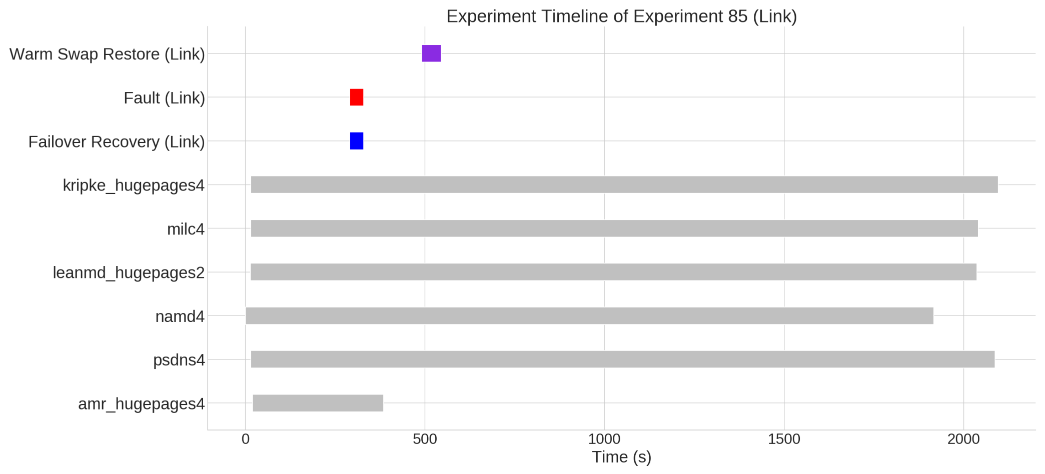


Figure 4.17: Timeline of Experiment 85 with a single link injection.

It is likely another error had interfered with the xtbounce underneath the hood. While it was the warm swap process that was disrupted in this experiment, this particular case reflects the spirit of Campaign IV and its goal to disrupt the failover process with injections during recovery.

Application Run Times

In this last section of Campaign II, we present one final result in Figure 4.18. In the presence of fault injections, applications were observed to have statistically significant increased run times on average compared to applications that ran without any fault injections in the system. This holds true across all applications, despite the crash cases observed. This result demonstrates the performance impact of even a single fault on the system and how critical fault detection and fault tolerance is in HPC systems.

4.4 Fault Injection Campaign III

In this campaign, experiments maintain single fault injections, still using applications from Set II. However, the main difference is varying the time at which faults are injected. All applications have about a 30 minute window. Campaigns I and II only focused on injecting in the first 10 minutes of an application's running time. Campaign III injects during the middle 10 minutes and the last 10 minutes. With regard to results, there is nothing new

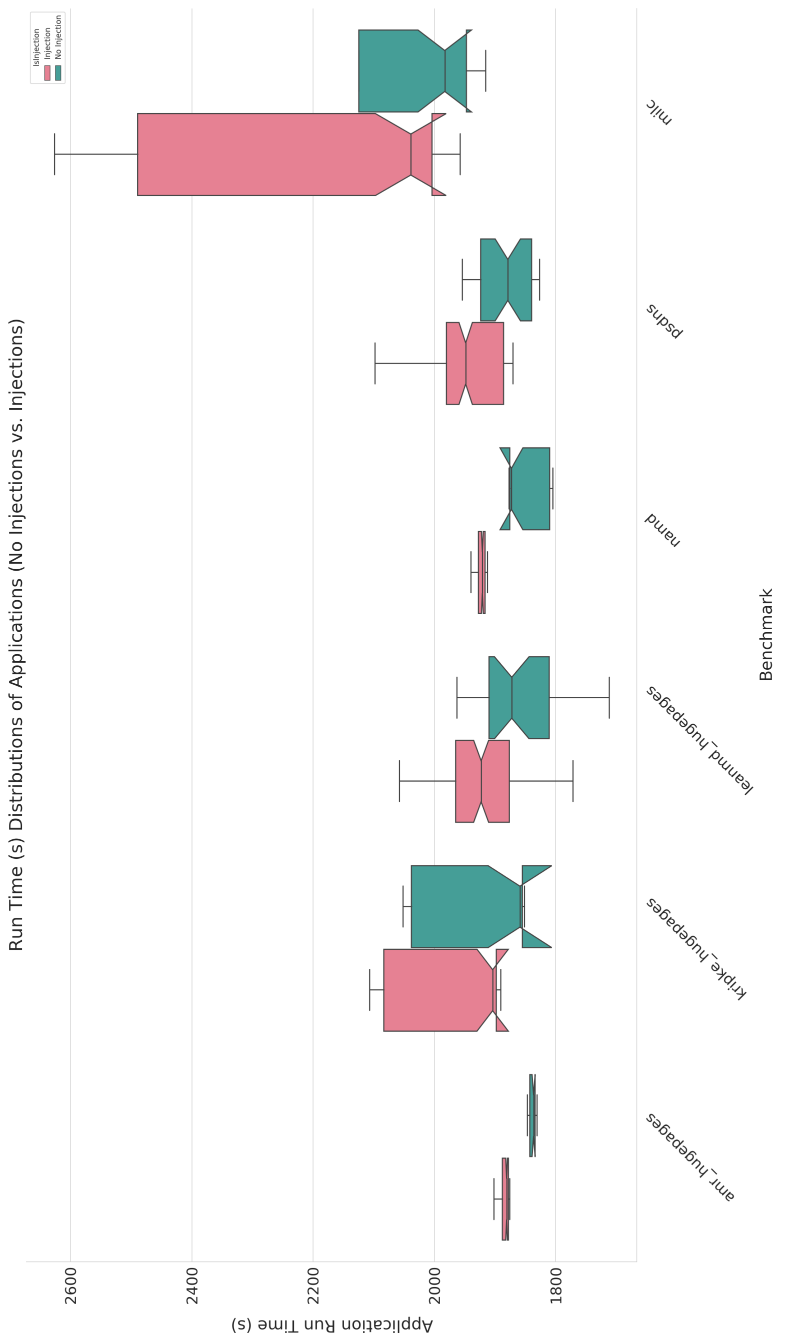


Figure 4.18: Distributions of application run times in Campaign II, comparing applications that ran in the presence of fault injections versus those that ran in a fault-free system.

Table 4.11: Full summary of Campaign III, showing the outcomes of applications that in the presence of (direct and indirect) faults.

Benchmark	**Application**	**Fault Type**	**Run Status #(%)**		
			Crash	Hang	No Impact
Charm++ (HugePages)	AMR	Connection	8 (80.0)	2 (20.0)	0
	Kripke	Connection	0	0	10 (100.0)
		Link	0	0	10 (100.0)
	LeanMD	Blade	0	0	10 (100.0)
		Connection	0	0	10 (100.0)
		Link	0	0	10 (100.0)
Charm++ (SMP)	NAMD	Connection	0	0	10 (100.0)
		Link	2 (20.0)	8 (80.0)	0
MPI	MILC	Blade	10 (100.0)	0	0
		Connection	0	0	10 (100.0)
	PSDNS	Blade	0	0	10 (100.0)
		Connection	0	0	10 (100.0)
		Link	0	0	10 (100.0)

uncovered by varying the fault injection delay that has not already been covered in the previous sections. Thus this section only presents the summary results.

4.4.1 Summary Results

There was a total of 30 experiments on JYC: 10 link injections, 10 connection injections, and 10 blade injections. The link injections targeted the 16-node NAMD application from the Medium Configuration; the connection injections targeted the four-node AMR application from the Sparse Configuration; and finally, the blade injections targeted the 16-node MILC application from the Dense Configuration. Table 4.11 and Figure 4.19 show the full summary of these experiments. Half of the experiments were injected with a 600 second delay while the other half were injected with a 1200 second delay. Figures 4.20a and 4.20b illustrate the timelines of a 600 second delay single blade injection experiment and a 1200 second delay link injection experiment, respectively. Experiment 241 targets a 16-node MILC application while Experiment 240 targets a 16-node NAMD application.

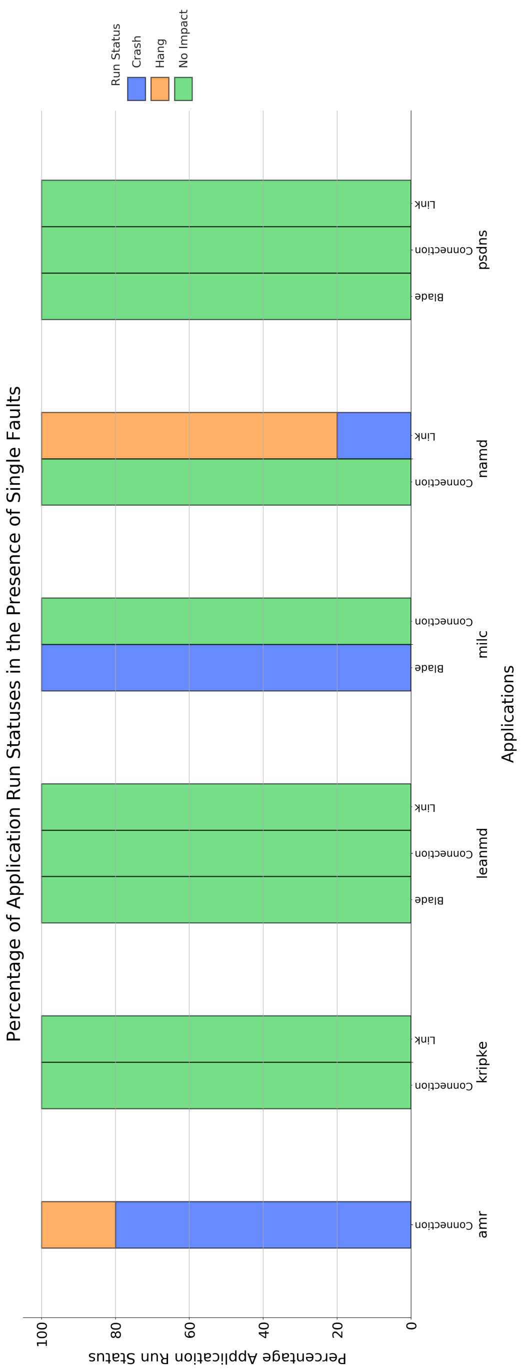
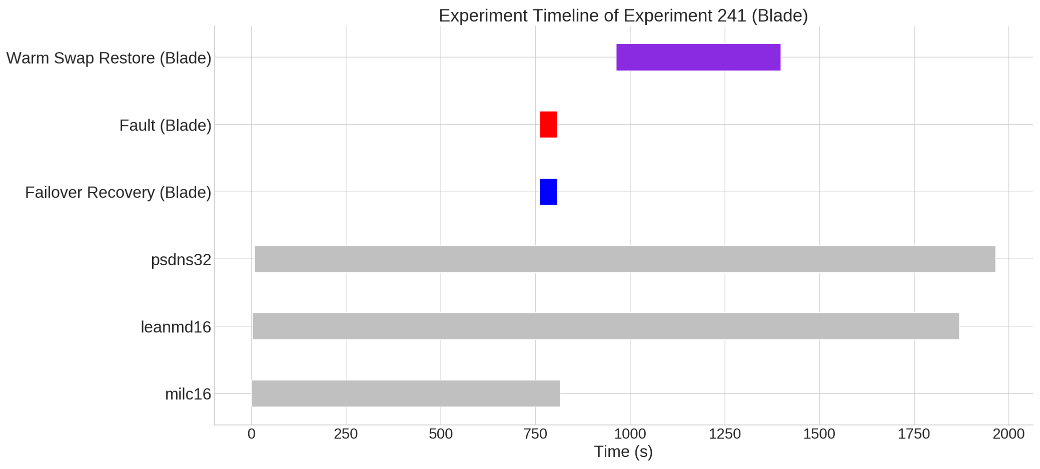
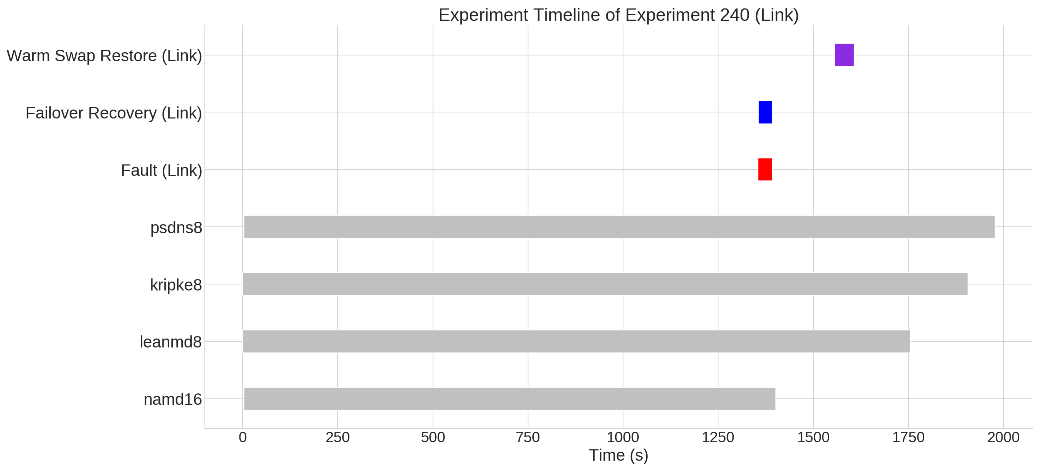


Figure 4.19: Summary percentages of Campaign III's injections (direct and indirect) and application run statuses, grouped by fault type and benchmark. Refer to Table 4.11 for counts.



(a) Timeline of Experiment 241 with a 600 second delayed blade injection.



(b) Timeline of Experiment 240 with a 1200 second delayed link injection.

Figure 4.20: Timeline plots of Campaign III experiments.

4.5 Fault Injection Campaign IV

Campaign IV's focus shifted away from assessment of application fault tolerance to evaluation of system resiliency. One of the main mechanisms for fault tolerance on HPC systems is its failover recovery procedures, which attempts to mitigate any ramifications of faults or failures and to minimize any disruptions of service. To evaluate failovers, multiple fault injections are required: one fault to trigger a failover and at least one other fault to inject during failover. The warm swap procedure is not targeted in this work, although the injection methodology developed here would work the same for injections during warm swap.

The main challenge in this campaign is the tight time window of the failover recovery process due to JYC being a small-scale system. Failover on JYC only lasts for 30 seconds in total while containing about 16 main stages with nine minor stages. This time window varies from system to system, depending upon the scale of the system: the larger the system, the longer the recovery window. For example, on Blue Waters and other large-scale production systems, this can last over 600 seconds [15]. A secondary challenge in this campaign is determining when to inject the second fault. In an ideal case, having the the ability to select any stage of recovery and promptly inject a fault requires real time monitoring of failovers. Network related events and errors are captured by the xtnlrd daemon and reported in the nlrd logs, which are stored on the SMW and rotated daily. We take advantage of the fact that once we inject the first fault, there is always a failover that immediately triggers in response. As soon as the injector triggers a fault, it begins to monitor the most current nlrd log based on date and check for the appropriate output line corresponding to the target stage. Once it detects the stage, it then injects the second fault.

In this section, we present the results of our injection during recovery experiments and illustrate the challenges of injecting on a small-scale system like JYC.

4.5.1 Summary Results

First, we present the summary results of initial injection during recovery experiments in order to determine the feasibility of these experiments on JYC.

In total, 71 injection during recovery experiments were performed and summarized in in Table 4.12. Four injection during recovery experiments are not counted in Table 4.12 because they involved multiple faults during recovery whereas Table 4.12 only shows experiments involving a single fault during recovery. The bulk of the direct injections shown in Table 4.12 involved combinations of link, connection, and blade injections. There are no indirect injections that led to any unusual behaviors and all failovers and warm swaps completed without issue.

The target recovery stages included Alive, Route Compute, Route Install, and Switch Netwatch. These were initially chosen due to the critical nature of each stage and due to scenarios observed in field data. Route Compute involves computing and rerouting around the current faults in the HSN. If this computation is based on an incorrect state of the HSN, then many packets will likely be incorrectly routed. Route Install and Switch Netwatch are similarly critical in that introducing a new fault during these stages will make the installed routes incorrect and obsolete. Lastly, the Alive stage checks which components are alive. Injecting during this stage may corrupt the alive statuses. In short, the goal is to confuse and corrupt the system's view of the network.

We first ran these experiments without information on recovery stage durations and delays, all of which affect the time to recovery stage detection and the time to fault injection. Post-experiment analysis revealed that based on durations and time to detection and injection, certain stages are better candidates to target than others. The next several sections discuss this analysis and present case scenarios on failover behaviors as well as late injections, which we call injection misses.

4.5.2 Duration of Recovery Stages

Using the data collected from Campaign II, which provided 127 data points of completed failover procedures, we measure the windows of time for each recovery stage. These durations are captured and calculated from lines in nlrd logs that announce the current recovery stage. For example:

```
2018-10-25T19:18:56.006931-05:00 SMWTDS 31595 2018-10-25 19:18:55
    SMWTDS 31597 ***** dispatch: current_state quiesce *****
```

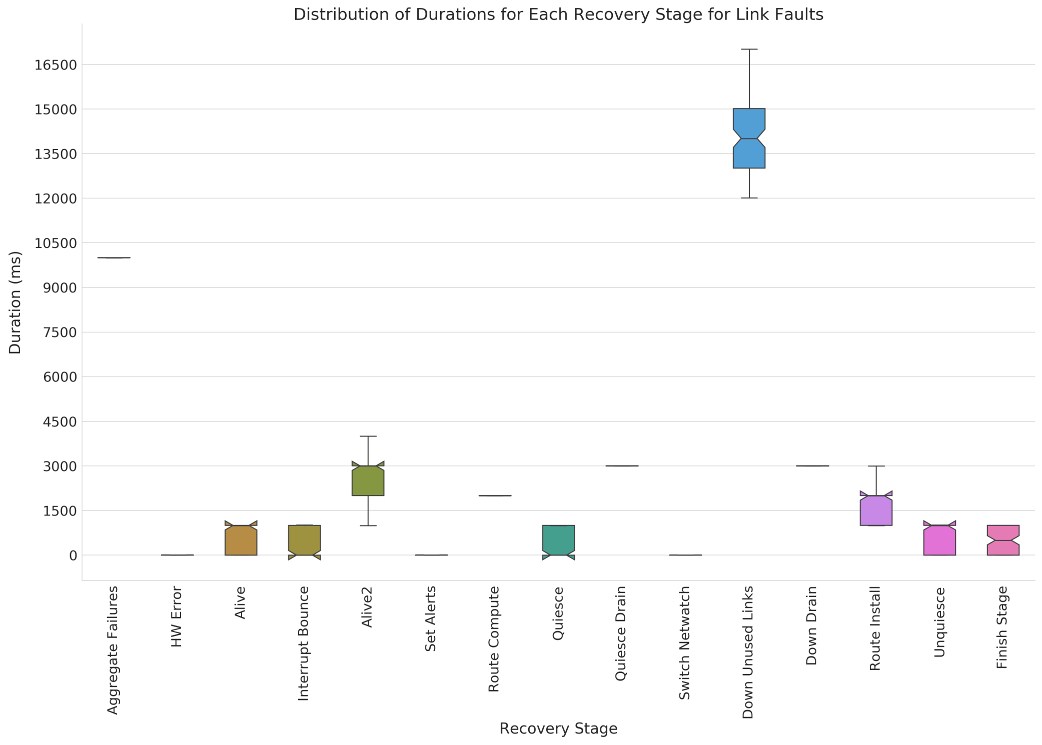
Table 4.12: Summary of Campaign IV, showing the outcomes of applications that are directly injected either by the first fault or the fault during recovery.

App	**Fault Type**		**Recovery Stage**	**Run Status #(%)**		
	1st Fault	2nd Fault		Crash	Hang	No Impact
LeanMD	Blade	Link	route compute	1 (100.0)	0	0
MILC	Connection	Blade	alive	0	0	1 (100.0)
			down unused links	0	0	1 (100.0)
			route compute	0	0	1 (100.0)
			route install	0	0	1 (100.0)
			switch netwatch	0	0	1 (100.0)
	Link	Blade	alive	0	0	1 (100.0)
			down unused links	1 (50.0)	0	1 (50.0)
			route compute	1 (100.0)	0	0
			route install	0	0	1 (100.0)
			switch netwatch	0	0	1 (100.0)
	Connection	Connection	alive	0	0	1 (100.0)
			route compute	0	0	1 (100.0)
			route install	0	0	1 (100.0)
			switch netwatch	0	0	1 (100.0)
	Link	Connection	alive	1 (50.0)	0	1 (50.0)
			route compute	2 (100.0)	0	0
			route install	1 (100.0)	0	0
			switch netwatch	2 (100.0)	0	0
	Connection	Link	alive	0	0	1 (100.0)
			route compute	0	0	2 (100.0)
			route install	0	0	1 (100.0)
			switch netwatch	0	0	1 (100.0)
	Link	Link	route compute	1 (100.0)	0	0
			route install	1 (100.0)	0	0
			switch netwatch	1 (100.0)	0	0
NAMD	Connection	Connection	alive	0	1 (100.0)	0
			route compute	0	2 (100.0)	0
			route install	1 (50.0)	1 (50.0)	0
			switch netwatch	0	2 (100.0)	0
	Link	Connection	alive	1 (50.0)	1 (50.0)	0
			route compute	0	2 (100.0)	0
			route install	0	3 (100.0)	0
			switch netwatch	0	2 (100.0)	0
	Blade	Link	route compute	0	1 (100.0)	0
	Connection	Link	route compute	0	1 (100.0)	0
			route install	0	1 (100.0)	0
			switch netwatch	0	2 (100.0)	0
	Link	Link	alive	0	2 (100.0)	0
			down drain	0	1 (100.0)	0
			route compute	0	3 (100.0)	0
			route install	0	3 (100.0)	0
			switch netwatch	0	2 (100.0)	0
PSDNS	Connection	Blade	alive	1 (100.0)	0	0
			down unused links	1 (100.0)	0	0
			route compute	1 (100.0)	0	0
			route install	1 (100.0)	0	0
			switch netwatch	1 (100.0)	0	0
	Link	Blade	alive	1 (100.0)	0	0
			down unused links	2 (100.0)	0	0
			route compute	1 (100.0)	0	0
			route install	1 (100.0)	0	0
			switch netwatch	1 (100.0)	0	0
	Link	Link	route install	0	0	1 (100.0)
			switch netwatch	0	0	1 (100.0)

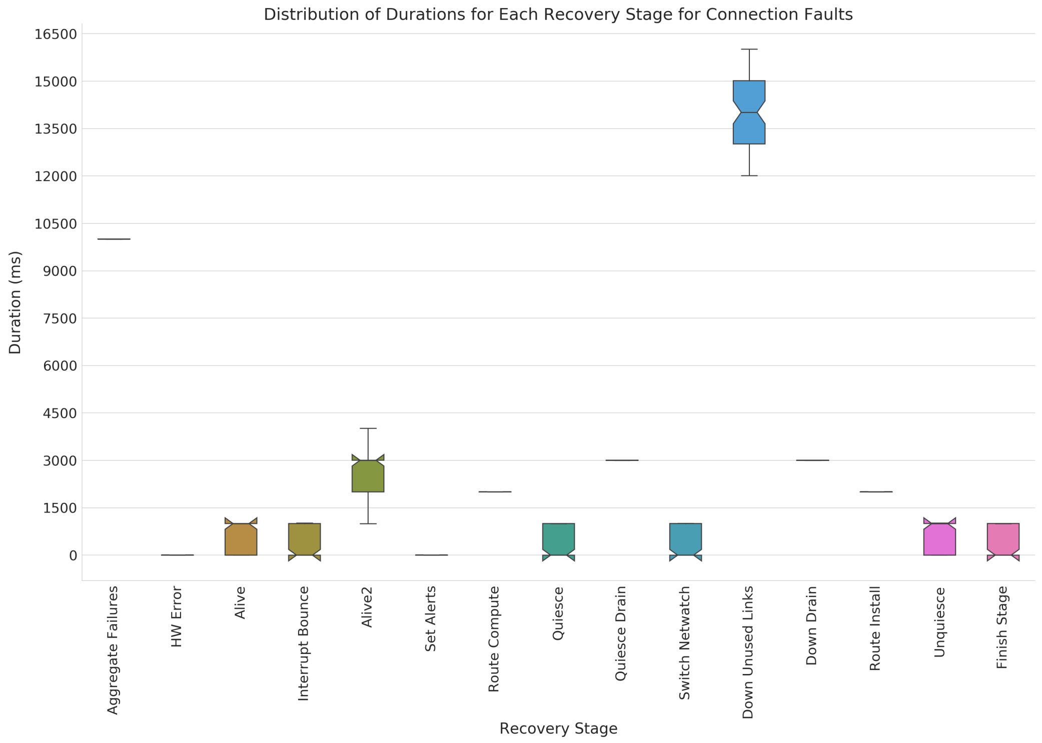
```
2 2018-10-25T19:18:57.007881-05:00 SMWTDS 31595 2018-10-25 19:18:56
    SMWTDS 31597 ***** dispatch: current_state quiesce_drain *****
```

To calculate the duration of a stage, we take the difference of the first timestamps between two log lines. For example, in the log snippet shown above, the duration of the Quiesce stage would be calculated by taking the first timestamp of line 1 and subtracting it from the first timestamp of line 2. Note that there is a second timestamp in the same log lines, which will be discussed in detail later. Here, since we only care about the relative difference and the first time stamp has more precision (microseconds), we use the first timestamp.

Figure 4.21 shows the distribution of durations for the 15 main stages. Each fault type is represented by its own separate graph. The final 16th stage, Initial, is left out as this is the system's natural, fault-free state. Injecting during the Initial stage would be outside the failover recovery period. The main differences across the fault types are the distributions of the Alive stage, which has a larger variation for blades; the Switch Netwatch stage, which has a larger variation for connections; and the Route Install stage, which has practically no variation for connections as compared to the other fault types. The main takeaway across all distributions is that the vast majority of stages are less than three seconds (note that the boxplots are displayed in milliseconds), which does not leave much room for detection and reaction. Certain stages are inherently impossible to inject on due to the short duration, such as Switch Netwatch with a median of 103 microseconds or Set Alerts with a median of 508 microseconds.

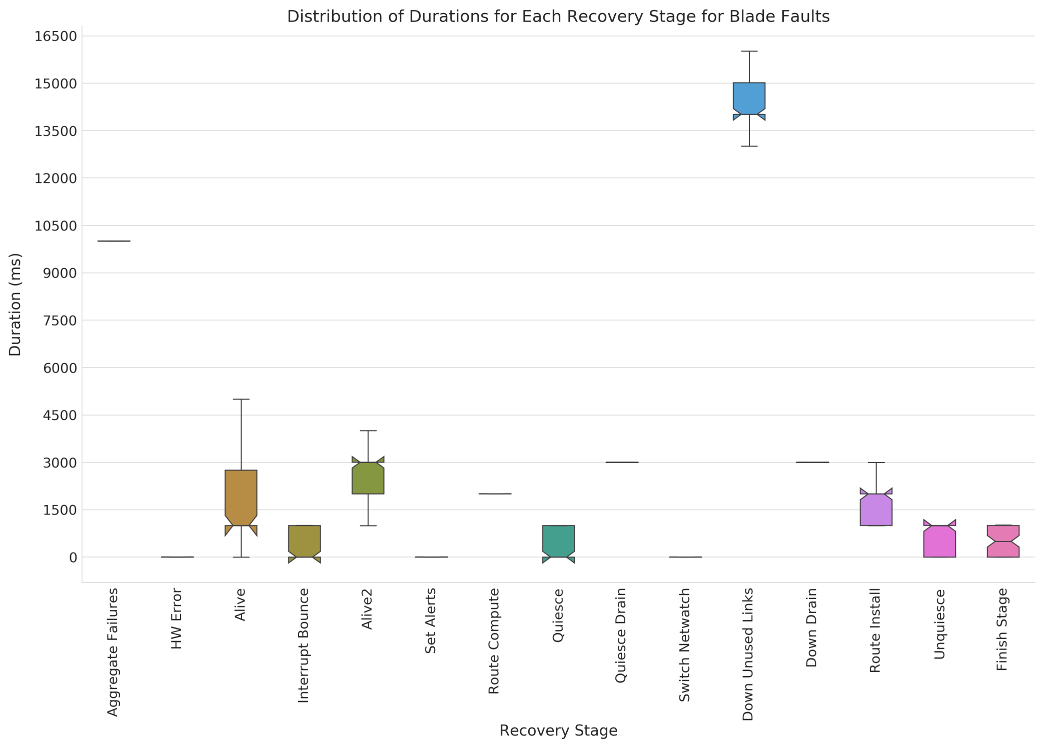


(a) Distributions of recovery stage durations. These were captured during single **link** fault failovers.



(b) Distributions of recovery stage durations. These were captured during single **connection** fault failovers.

Figure 4.21: Distributions of recovery stage durations for each fault type.



(c) Distributions of recovery stage durations. These were captured during single **blade** fault failovers.

Figure 4.21: Continued.

4.5.3 Latency of Injection During Recovery

Given that the windows of recovery stages are brief, verifying how quickly the injector can detect the target recovery stage and react in time to inject is necessary for assessing the injector's capabilities. There are several delays that contribute to this latency between the start of the recovery stage and the injection of the second fault as shown in Figure 4.22. These delays include the time W that it takes for the nlrd daemon to generate and write the target stage's log message, the time D that it takes for the injector to detect the target stage via the nlrd logs, and the time I that it takes for the injector to react and inject the second fault. If the average time to detection plus the average time to injection is beyond the third quartile of a recovery stage duration distribution, then it is highly unlikely to have a successful injection in this stage. Each of these contributors to the latency is discussed in the following sections.

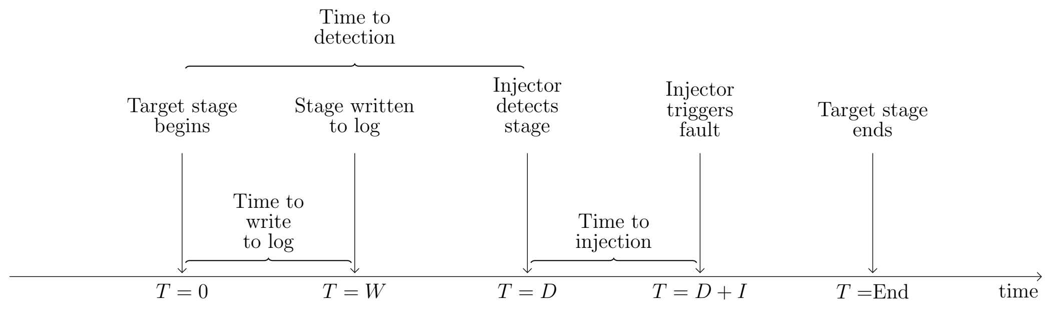


Figure 4.22: Delays that contribute to the latency between occurrence of a target recovery stage and a fault injection.

Delay Between Stage and Log Message

To select a specific stage to target, we require real-time monitoring of all failover procedures. As previously mentioned, network events are stored in the nlrd logs. These logs are the only source of information about the system's network recovery stages that the injector toolkit may access. An example output from an nlrd log is provided as follows:

```
2018-10-25T19:19:20.026178-05:00 SMWTDS 31595 2018-10-25 19:19:19
   SMWTDS 31597 ***** dispatch: current_state finish *****
2018-10-25T19:19:20.026188-05:00 SMWTDS 31595 2018-10-25 19:19:19
   SMWTDS 31597 do_finish: Re-enabling throttle daemon...
2018-10-25T19:19:20.026203-05:00 SMWTDS 31595 2018-10-25 19:19:19
   SMWTDS 31597 INFO: 24 out of 24 L0s are alive
```

Nlrd logs are the only system logs we examine in this work that have two timestamps in a single log line. There is a chain of generating and forwarding of messages that is dictated by the rsyslog, which contains forwarding rules on the SMW that place incoming messages into their final corresponding log files. Messages that are sent to rsyslog are then forwarded and included in its entirety, including the original message's header (e.g. timestamp), as the message body in a new message with a new timestamp. Thus the first, outer timestamp is an artifact of rsyslog while the second, inner timestamp is closer to the event. The challenge that arises here is that the precision of the inner timestamp is at the level of seconds while the outer timestamp's precision is at the level of microseconds. In the log outputs shown above, the event happened approximately at 19:19:19 (hour:minute:second) while the message was placed in its final log location at around 19:19:20.

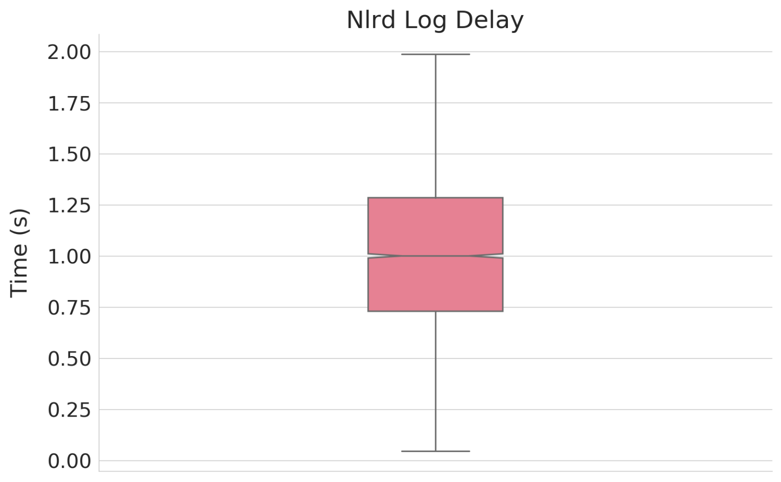


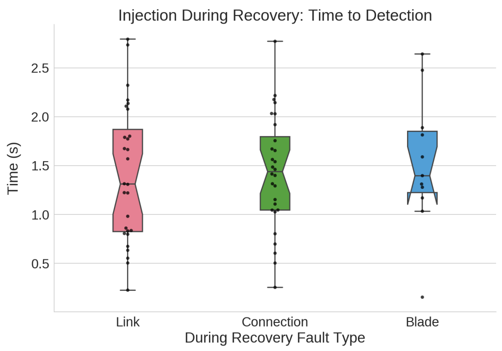
Figure 4.23: Distribution of the delay between recovery stage occurrence and its corresponding log message written to the nlrd log. This is an upper bound estimation. See Appendix H for the summary statistics.

It is difficult to precisely measure the delay caused by this phenomenon, from the occurrence of the event itself, to the event's log message being generated, and finally to the event's log message being forwarded and placed into the final SMW location. To give an upper limit on this delay, we examined over 7000 lines from nlrd logs collected during Campaign II, selecting lines that only report the `current_state` as shown in the example nlrd log messages and calculating the difference between the two reported timestamps. We assume the second, inner timestamp is at zero microseconds and that it is the time of the actual event occurrence. There is no means to measure the delay between the actual event occurrence to its originally generated log message. This upper limit distribution is shown in Figure 4.23.

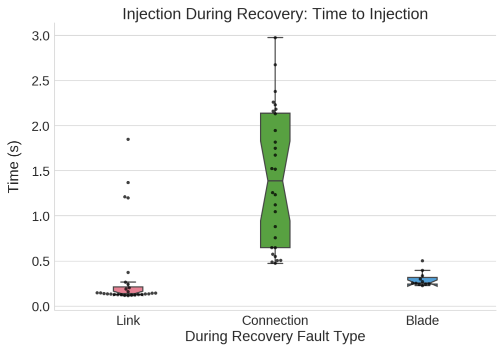
On average, it takes about one second from the event's original log message to it being forwarded and written into its final destination. This, however, can take as long as almost two seconds! Most recovery stages are sub-three seconds, which leaves little time for the injector itself to react.

Delay Between Stage Detection and Fault Injection

To detect the target stage, the stage during which we intend to inject a fault, we implemented a simple log watcher in Python, similar to performing the `tail -F *.log` UNIX command. To avoid any unnecessary overhead, we do not spawn a new thread or process for the watcher. The watcher routine



(a) Distributions of time from the event's original log message to the target stage being detected by the injector. See Appendix H for the summary statistics.



(b) Distributions of time from the target stage being detected to the second fault injection. See Appendix H for the summary statistics.

Figure 4.24: Distributions of time to detection and time to injection.

starts immediately after the first fault is injected and reads lines out of the nlrd log as they are being written. If a target stage is detected, then the injector immediately sends the appropriate fault command, which is prepared even before the first fault is initiated to reduce overhead. If a target stage is never reached, the watcher "times out" after a certain number of iterations while reading the log file. This number of iterations is precomputed beforehand in order to reduce the overheads of using any timeout functionalities or any library that relies on the system time.

Certain latencies, however, cannot be completely avoided as the watcher still must spend time to check every line for the proper substring, the injector's logger must log the time it detected the target stage, and the injector itself must send the fault command to the system. The injector's logger does not log when the command is sent (to avoid unnecessary overhead), but it does record it once the command returns with a status.

We provide another upper bound on the time from the event's original log message to the target stage being detected by the injector and the time from target stage detection to the second fault injection in Figures 4.24a and 4.24b, respectively.

Time to Detection. In Figure 4.24a, the reactions to both fault types examined (link and connection) on average take around 1.43 seconds, but can take over 2.7 seconds at worst. Note that the second, inner timestamp was used to calculate the time to detection. The watcher itself is reasonably fast, about a couple milliseconds on average, in detecting when the final log message is written to the nlrd log.

Time to Injection. In Figure 4.24b, note the stark contrast between injecting a link fault, which on average takes a couple hundred milliseconds, versus a connection fault, which on average takes about 1.4 seconds. This is due to the fact that for connection faults, multiple commands must be sent to fail all the links in a torus connection direction (four for the Y+/Y- directions or eight for the Z+/Z- directions). For link faults (and blade faults), only one command is sent.

Based on these analyses, it becomes evident that in addition to recovery stages that are inherently impossible to inject on, there are also stages with durations that are still too short to accommodate the time to detection and time to injection. Since time to detection takes about 1.4 seconds and time to inject a connection fault takes about another 1.4 seconds, there are few stages for a connection injection to succeed in terms of being timely. It will likely only succeed for Down Unused Links, Down Drain, Alive2, and Quiesce Drain stages. This limits possible injections into critical stages to link and blade injections.

4.5.4 Recovery Behavior and Resiliency to Faults

In this section, we review case experiments to highlight the resiliency behaviors of failover in the presence of multiple faults as well as to illustrate once more the challenge of the small window of recovery on small-scale HPC systems such as JYC. Each case is presented with an experiment timeline of the applications, injections, failover recoveries, and warm swap restorations. An entire experimental timeline ends when the system is restored to a fault-less state and all applications terminate, which typically happens at about 30-40 minutes of run time. Each case is also presented with a more fine-grained time scale graph, focusing in on each experiment's failover recovery timeline and depicting when the two faults are injected, the timings and durations of recovery stages, and the overall failover behavior. The warm swap stages are left out of these recovery timeline graphs as they are not the targets of injections in this campaign.

Double Failovers: Fail and Restart

In experiment 251, two link faults were injected into the system, with the second during the crucial Route Compute stage. The experiment timeline of the injection, recovery, and warm swap events is shown in Figure 4.25. Note that only one warm swap may be invoked at any given time on the system. In this scenario, after the second link fault was successfully injected during the proper stage, the failover recognized the fault immediately, prematurely terminated the current failover, and restarted the failover process. This behavior is shown in Figure 4.26 when the failover jumped to the Finish stage after the Check Route Compute stage. The nlrd logs showed the following messages:

```
2018-10-25T19:18:39.982166-05:00 SMWTDS 31595 2018-10-25 19:18:39
    SMWTDS 31597 ***ERROR***: Link recovery operation failed; error
    11
2018-10-25T19:18:39.982181-05:00 SMWTDS 31595 2018-10-25 19:18:39
    SMWTDS 31597 Error string was: Link resiliency operation aborted
     due to hardware failure during route computation
```

This disproves our previous speculation that if we disturb the route computations, then the system would find itself installing invalid routes and thus be in an invalid state. However, it seems that the system can recognize a failed or corrupted Route Computation stage and is able to restart the failover without further issue. This fail and restart behavior is only observed for the Route Compute stage.

Double Failovers: Back-to-Back

In experiment 270, two link faults were injected into the system, with the second during the Down Unused Links stage, which is the stage with the longest duration. The experiment timeline is shown in Figure 4.27. In this scenario, the first failover completed successfully and the second failover immediately followed, skipping over the Initial stage that normally is present in a completed failover. The system was able to recognize that a separate fault occurred during the Down Unused Links stage and initiated a second failover to handle the second fault immediately after the first failover completed. Injecting during many of the recovery stages, such as Route Install, Quiesce

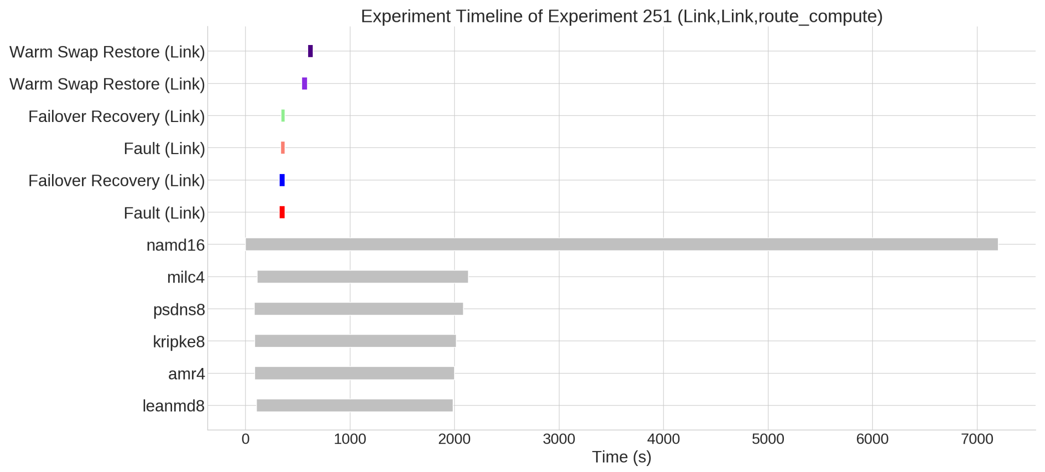


Figure 4.25: Timeline of Experiment 251 with two link injections, depicting when they are injected, when their failover procedures begin and complete, and when the warm swap restorations begin and complete.

Drain, or Down Drain, produces these back-to-back failovers response as shown in Figure 4.28. In other words, this is the most common observed response to injections during recovery.

Single Failover

In experiment 253, two link faults were injected into the system, with the second during the Alive stage, a stage in the early pre-quiescence period of the recovery timeline. The experiment timeline is shown in Figure 4.29. Note that there is only one failover process. In this experiment, after the link fault was successfully injected during the proper stage, the fault was recognized immediately and was added to the list of failed components. The failover process continues as usual and does not require a second failover. This behavior is shown in the experiment's recovery timeline in Figure 4.30.

While the Alive stage may be checking for the status of blades, there is still a background mechanism in the xtnlrd daemon to detect future faults. Additionally, this case demonstrates that the failover does not need to restart since the recovery process is still in its earlier stages. At this point, the failover has not yet used the old list of failed components to compute new routes. Injecting during the Alive2 stage also produces the same failover behavior.

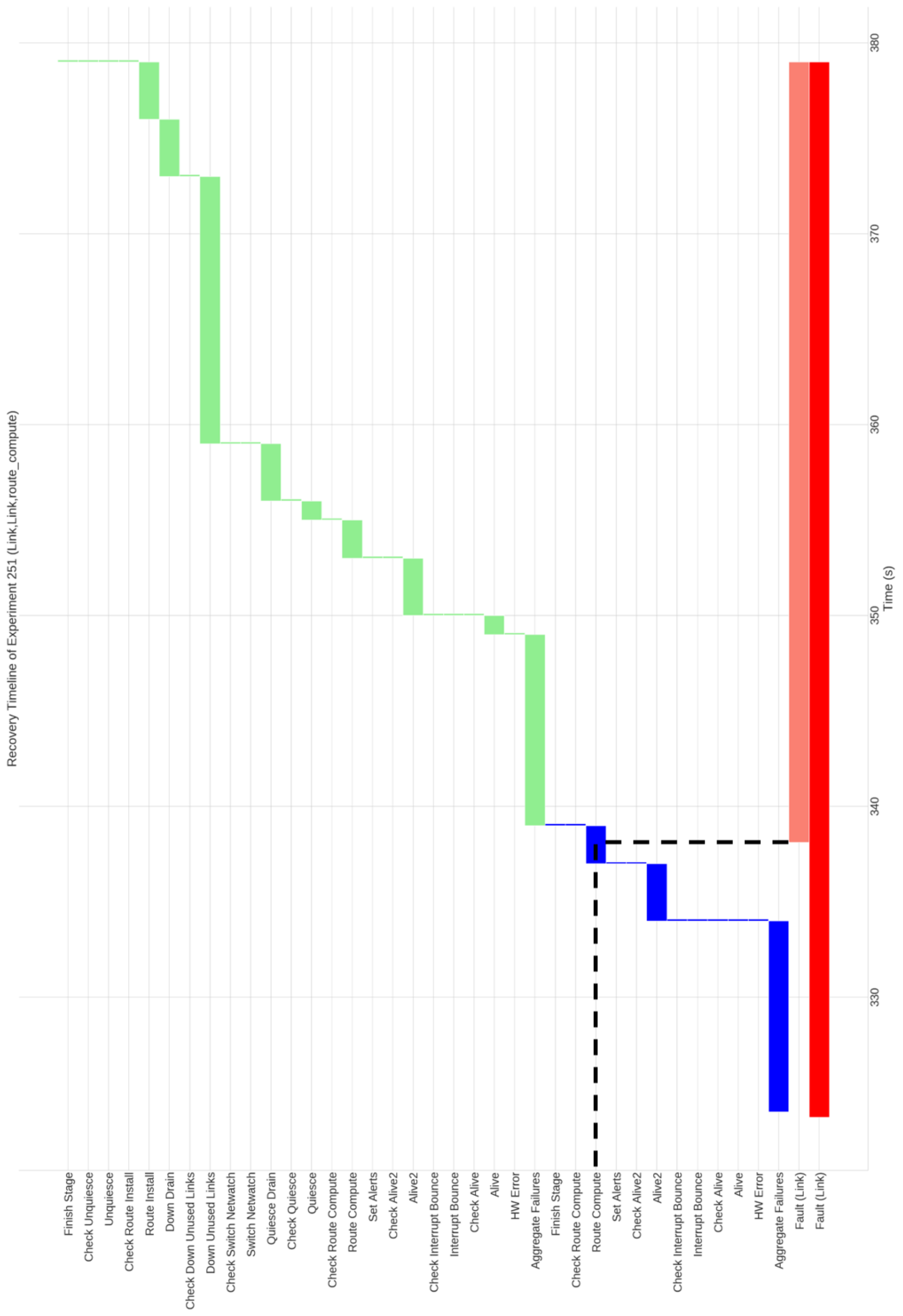


Figure 4.26: Breakdown of Experiment 251's fail and restart recovery timeline, depicting two link injections, with the second during the Route Compute stage. Faults are highlighted in red and stages are highlighted in blue (first failover) and green (second failover).

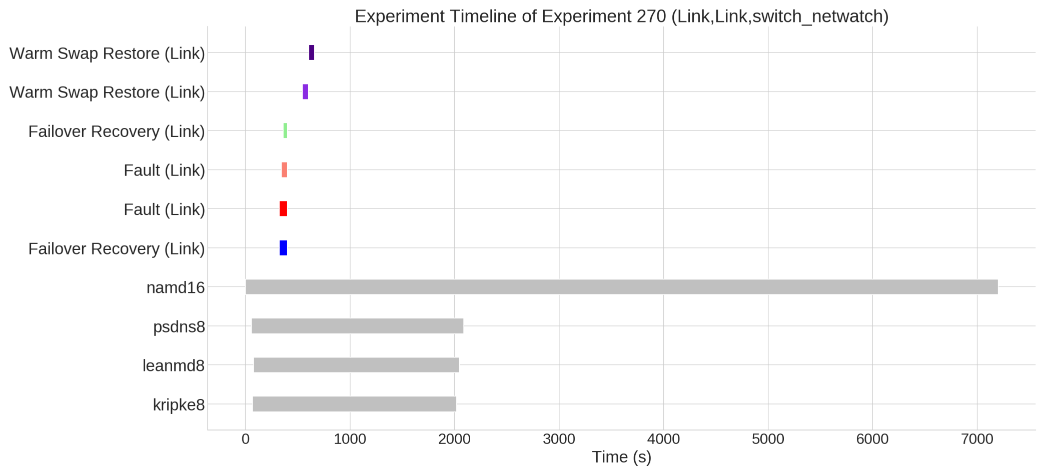


Figure 4.27: Timeline of Experiment 270 with two link injections, depicting when they are injected, when their failover procedures begin and complete, and when the warm swap restorations begin and complete.

Injection Misses

Experiment 276 is presented in this section to illustrate an injection miss, which is a scenario in which the injector fails to inject during the proper target stage. There are no detrimental effects on the system of an injection miss. Either the fault is caught by a subsequent recovery stage or it becomes a second fault after the initial recovery completes. In the latter scenario, a new failover starts in response to the second fault.

Experiment 276 involved an initial connection fault and a second link fault, targeting the Switch Network stage. In this scenario, the injector failed to inject during the Switch Netwatch stage and instead injected during the subsequent Down Unused Links stage as shown in Figure 4.31. Switch Netwatch is one of the near impossible stages to inject during because of its microsecond-level durations. Like in Experiment 270, the first failover process completed successfully, but instead of returning to the Initial stage, the second failover commenced to handle the second link fault. All injections intended for Switch Netwatch ended up being caught in the Down Unused Links stage, which typically lasts over 10 seconds.

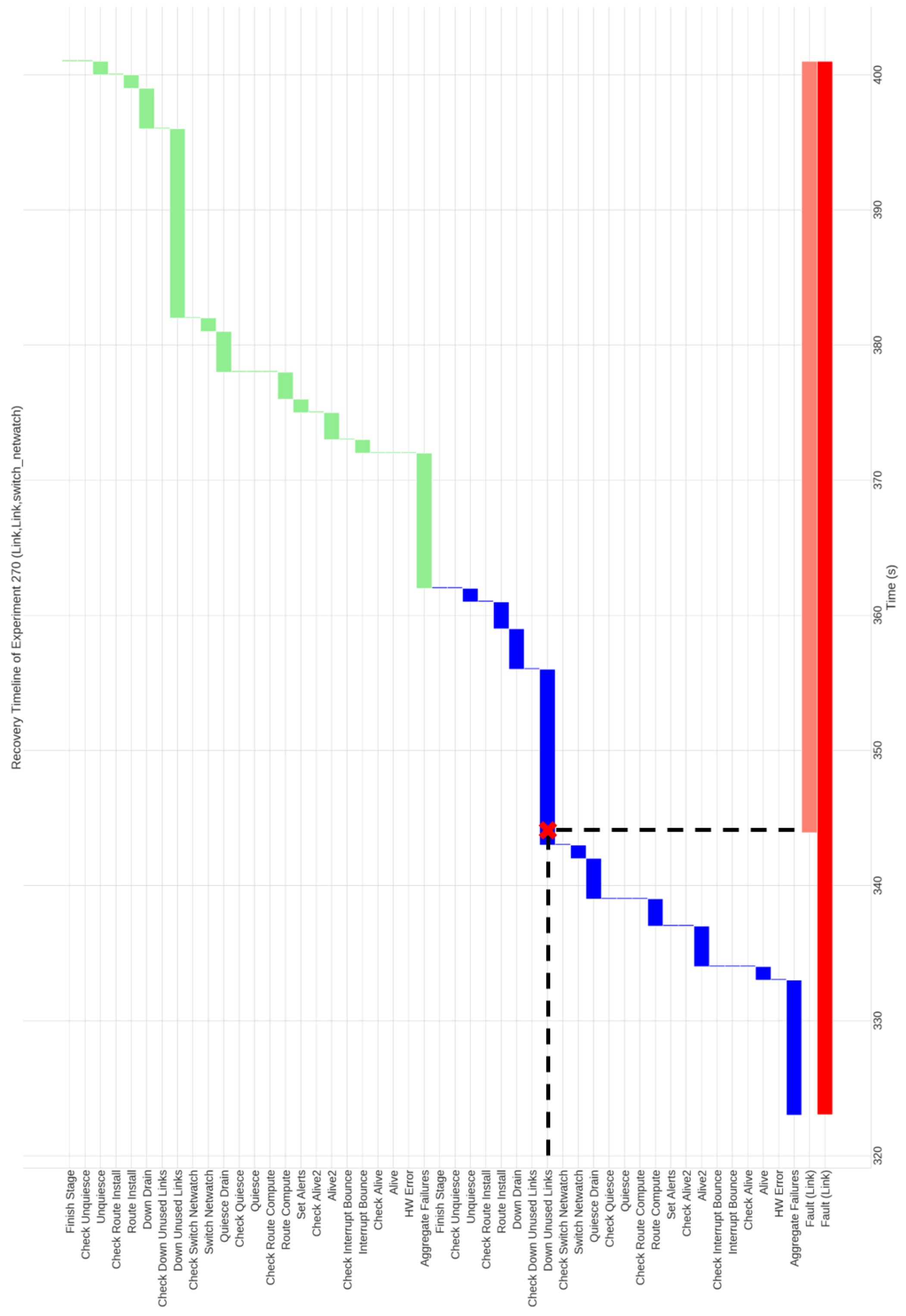


Figure 4.28: Breakdown of Experiment 270's double failover recovery timeline, depicting two link injections, with the second during the Down Unused Links stage. Faults are highlighted in red and stages are highlighted in blue (first failover) and green (second failover).

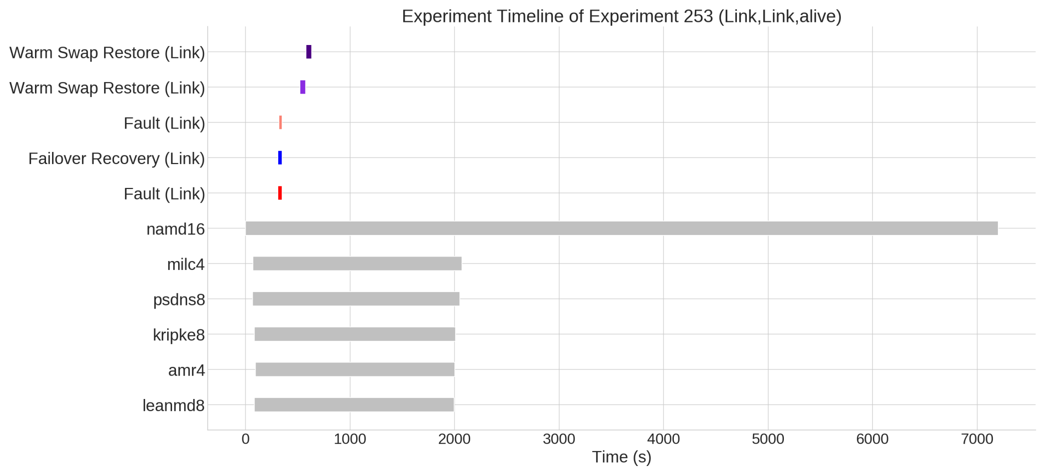


Figure 4.29: Timeline of Experiment 253 with two link faults, depicting when they are injected, when their failover procedures begin and complete, and when the warm swap restorations begin and complete.

Multiple Injections During Recovery

To further demonstrate HPCArrow's capabilities to inject multiple faults, we conducted four experiments involving multiple faults injected during a single failover and during a series of failed failovers (all but the last failover procedure fails). For these experiments, we run the Dense Configuration with faults targeting PSDNS and MILC applications.

Injections into the same recovery stage. In one experiment, we targeted the Alive stage with eight link faults during the same recovery stage of a single recovery procedure. A second experiment injected four connection faults. Ultimately, all failovers succeeded and recovery behavior matched what we observed with only a single injection during recovery.

Injections into recovery procedure initiated in response to failure of proceeding failover. For two experiments, we targeted the Route Compute stage because it is the only stage, as previously shown, that is immediately terminated by the system upon detection of additional faults and automatically restarted as a new failover. Instead of injecting multiple faults all at once, we injected one link fault for each Route Compute stage of a restarted failover (in response to the proceeding failed failover). One experiment involved a total of 16 link faults (the other experiment involved eight faults), each fault injected during the Route Compute stage and causing the current failover to terminate and restart. In total, 15 failovers were

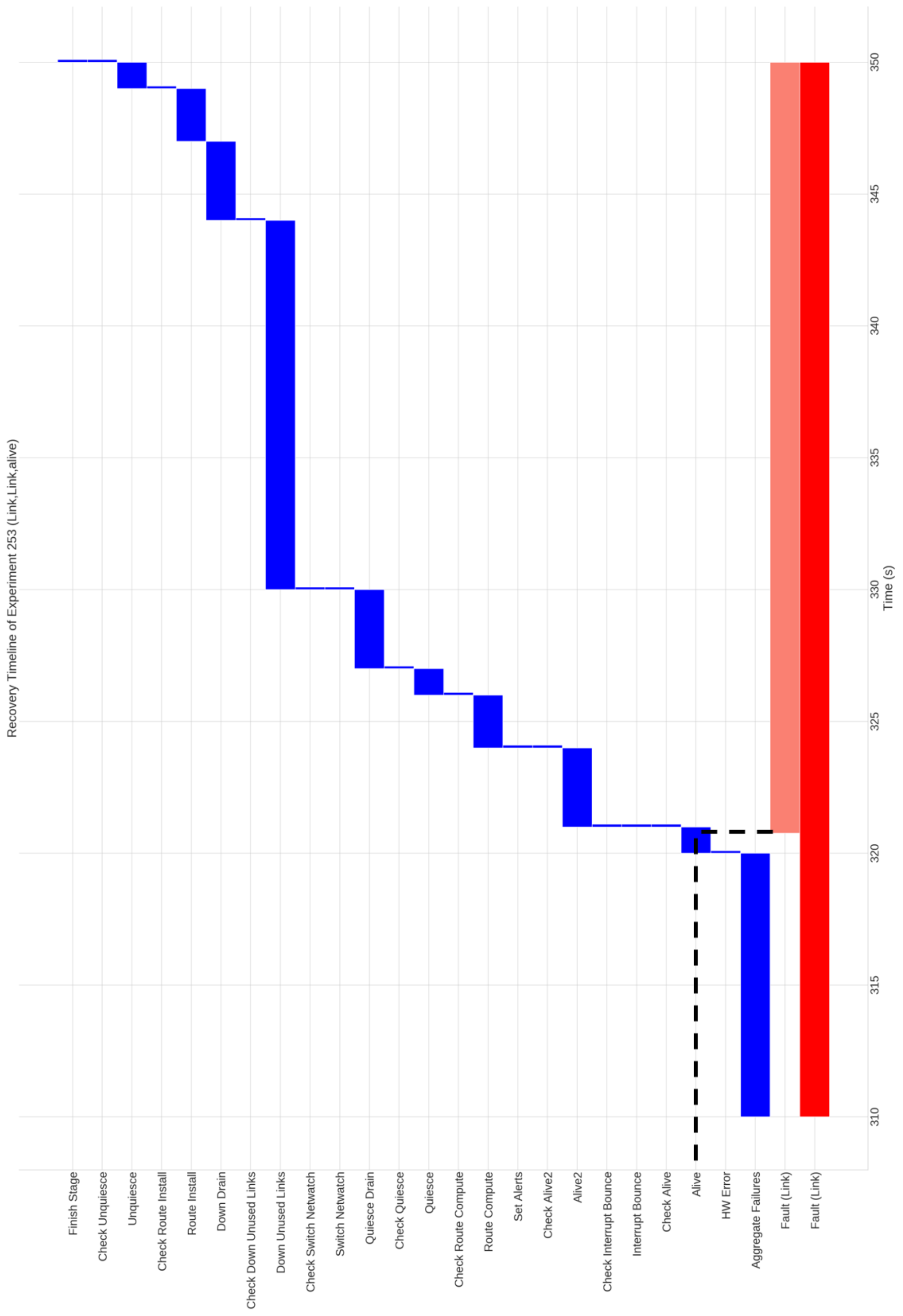


Figure 4.30: Breakdown of Experiment 253's recovery timeline, depicting two link faults, with the second during the Alive stage. Faults are highlighted in red and stages are highlighted in blue (first failover).

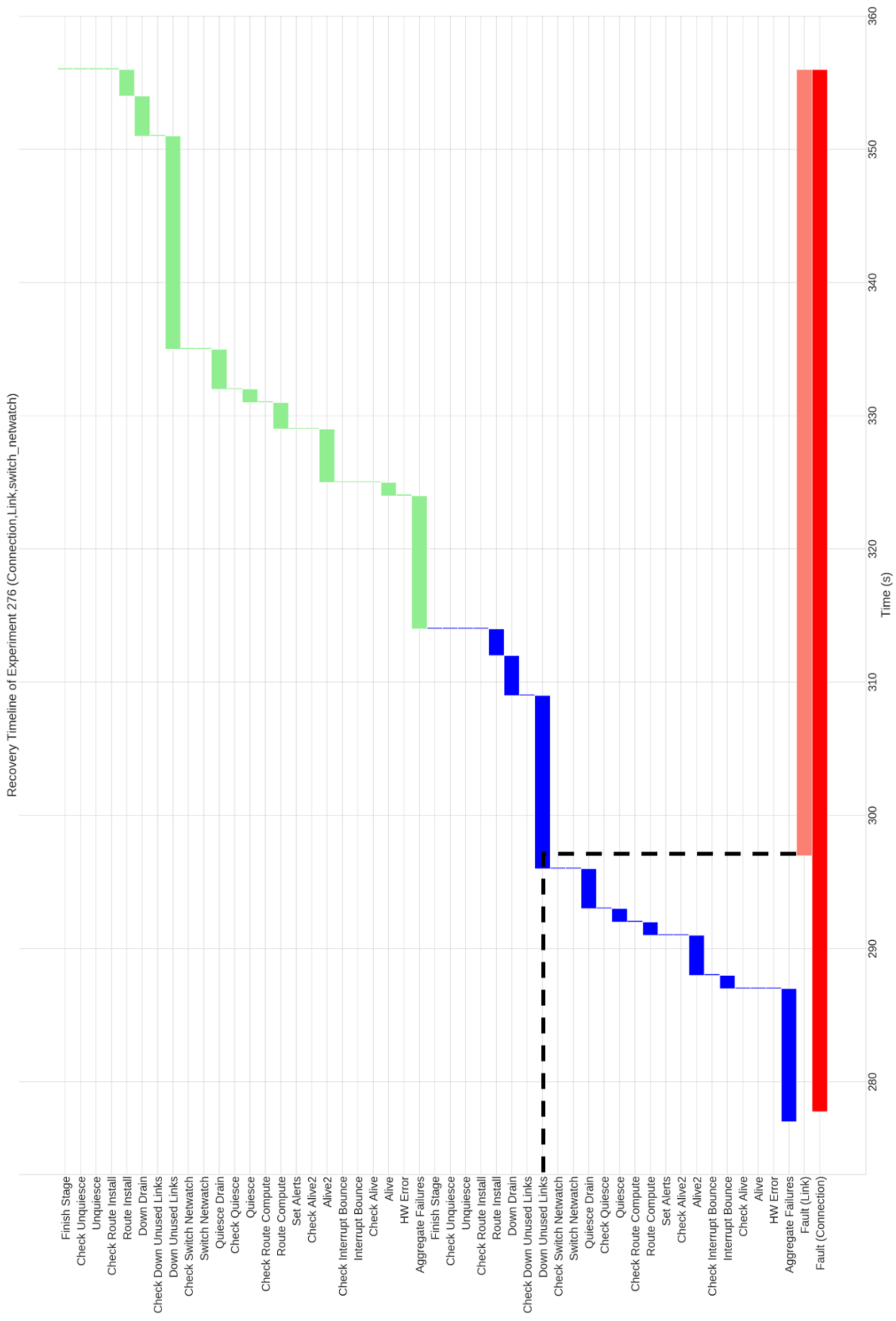


Figure 4.31: Breakdown of experiment 276's recovery timeline, depicting two link faults, with the second failing to inject in the Switch Netwatch stage. Faults are highlighted in red and stages are highlighted in blue (first failover) and green (second failover).

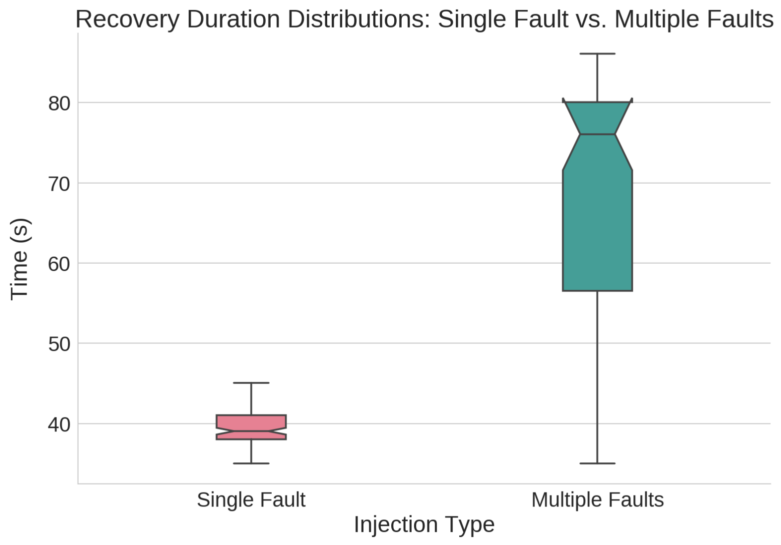


Figure 4.32: Distributions of the recovery durations of single fault experiments (Campaign II) versus multiple faults during recovery experiments (Campaign IV). Refer to Appendix H for the summary statistics.

terminated and restarted. The 16th and final failover ultimately succeeded and the system did not suffer any permanent problems.

Increased Failover Recovery Time

By restarting the failover or by starting a second failover, the recovery time of should inevitably increase. An increased time in failover recovery means an increase in the probability of the occurrence of additional failures, which may eventually lead to catastrophic failures in the system. According to [5], if the recovery time exceeds 300 seconds, the probability of additional failures raises to 0.8.

As shown in Figure 4.32, failovers that occurred during the single fault injection experiments of Campaign II had recovery durations of about 39 seconds. Failovers that occurred during the injection during recovery experiments of Campaign IV had recovery durations of about 76 seconds, which is close to double the durations in Campaign II. There is a clear increase of recovery time as expected, which in turn increases the probability of additional failures occurring.

CHAPTER 5

FUTURE WORK AND CONCLUSION

In this chapter, we summarize the results and observations of all four fault injection campaigns. We provide discussions on these observations, suggest future work in this space, and conclude this thesis.

5.1 HPCArrow

We developed a SWIFI toolkit called HPCArrow as a means of studying fault tolerance and resiliency on HPC Cray systems by systematically executing fault injections on network and compute components. The advantages of this tool are its abilities to provide a controllable injection environment and perform experiments that are automated and repeatable.

HPCArrow's Fault Injector module can inject link, connection, node, and blade faults by running on the SMW and issuing Cray commands. It performs single or multiple injections at random or at user-specified locations and times. In terms of multiple injections, all injections following the first fault can be triggered either during recovery or after recovery. For injections during recovery, this module can monitor network events in real time and conduct single or multiple injections during recovery, targeting a user-specified recovery stage. The injector is also responsible for performing warm swaps to return injected components back into service and restore the system to a fault-less state.

The Workload Manager module can compile, pre-configure, and submit applications to be launched on the system, all based on user-configurable YAML files. Currently, HPCArrow supports Moab/TORQUE as the resource and workload managers. It also supports Slurm [39] in preparation for fault injections on Aries systems.

Should the tool be needed by users without root access on the SMW, the

Injection Manager module allows remote triggering of injections, up to two injections at this time. The Injection Manager is also responsible for launching the log transfer script that gathers all system, hardware, and injection logs and transfers them to a location accessible to our user accounts.

HPCArrow's capabilities are successfully demonstrated through four different types of fault injection campaigns on JYC. Our tool provides insight into application-level resiliency, system-level recovery behaviors, and fault-to-failure propagation paths. It has also been verified to work on several Cray XC systems that use the Aries interconnect.

5.2 Application-Level Resiliency

Across four fault injection campaigns, we ran nine different applications, covering various HPC programming frameworks. These include Charm++, MPI, and PGAS. Throughout the course of this study, we observed various behaviors at the application level in response to faults injected into the system. The behavior common to all is the increase in application run time during injection experiments as opposed to running on a normal, fault-free system. The more failures, the longer the run time. In addition to having increased run time, applications were also observed to crash or hang. These unexpected behaviors highlight a fault tolerance deficiency among these frameworks in that these applications cannot tolerate simple failures of the network, even if recovery is successful. At the framework level, we summarize their unique observations below.

5.2.1 Charm++

Of the three programming frameworks, Charm++ showed the most susceptibility to faults, particularly to failures in communication paths and packets. Across AMR, Kripke, LeanMD, and NAMD, we observed all of these applications experiencing crashes or hangs. By injecting a single connection fault, we can guarantee failure, either a crash or hang, of any of these Charm++ applications. Even a single link failure can cause any of these applications, aside from AMR (SMP), to hang quite frequently.

5.2.2 MPI

Opposite of Charm++, MPI holds up as the most resilient programming framework observed in this study. In terms of direct link or connection injections, both AWP-ODC and PSDNS applications experienced no adverse effects and were able to resume computational progress once the failover rerouted around the injected faults. There were several instances of PSDNS crashes due to running out of memory space across four nodes. The application had been properly scaled and the errors occurred either long after or some time before the fault injection, so it may be possible that the fault injection was not responsible for directly causing the errors and crashes experienced by PSDNS. This phenomenon was also local in time, occurring repeatedly only during one night of experiments.

MILC, however, was shown to crash only due to link injections with error messages indicating packet drops or problems processing the data in the packet. Surprisingly, MILC does not display any adverse effects due to connection injections, even though connection injections are the equivalent of multiple and successive link faults.

5.2.3 PGAS

Lastly, the single PGAS application, UPC-FT, also showed susceptibility to link and connection failures, often crashing in the presence of a single fault. This, however, is expected as PGAS is known to be vulnerable to failures in the network.

5.2.4 Discussion

Across all of these applications and frameworks, there are varying levels of susceptibility to network-level failures and to disruptions in their communication paths. The common story behind these observed failures is packet drops and loss of these transactions, which then cause applications to fail their assertions, to experience segmentation faults, or to pass around invalid arguments. In these crash scenarios, the onus falls partially on application programmers to write more fault-tolerant programs. An example would be retrying failed transactions for a short time before timing out. On the system side, if the network can be globally quiesced, then a global broadcast that can tell all compute node threads to pause may be a viable option to safeguard applications against network failures.

Application hangs are phenomena that were previously unobserved in measurement-based analysis on production data. This hanging behavior is unique to the Charm++ applications in our experiments. However, it is unclear what conditions can lead to a crash or hang. Application logs, traffic data, and system logs show similar trends and outputs for crash and hang behavior outcomes and offer no indicators to differentiate between the two. The only indication of an application hang is that it is reported as "running" beyond its usual execution run time and eventually hits the two-hour Walltime. In that time, no forward progress is made and the application's computations are left incomplete.

These indications, however, can only be detected through offline analysis and diagnosis. Future work should investigate real-time detection of hanging applications on the system side so that the system can either inform the user or terminate the application itself. A hung application is a waste of system resources, reducing system efficiency, and a waste of the unaware user's time and money. Other avenues of system information to detect application hangs in real time will need to be explored, such as CPU usage during the hung period. Additional information such as memory usage will also be helpful in diagnosing the PSDNS crashes. Application writers should also consider developing more fault-tolerant programs since programming frameworks like MPI leave responsibility of fault tolerance to the programmers.

5.3 System-Level Resiliency

In additional to application-level resiliency, this work also endeavored to evaluate system-level resiliency and its responses to failures in the network and compute hardware. Across all fault injection campaigns and nearly 300 fault injection experiments, all failovers and warm swaps were reported to have completed successfully. While Campaigns I-III provided some understanding of failovers, injection during recovery experiments in Campaign IV were intentionally designed to provide insight into failover behaviors, recreate failure scenarios observed in other studies, and assess overall resiliency.

In all 71 injection during recovery experiments, every failover recovery and warm swap restoration completed successfully. Combinations of link, connection, and blade injections were attempted across various stages, including Route Compute, Route Install, Alive, Switch Netwatch, Down Unused Links, and several more unintended stages due to injection misses. These injection misses were experiments that failed to inject during the proper target stage. These unintended stages included Alive2, Quiesce Drain, Check Route Install, and Down Unused Links.

Based on results, the system's failover process is quite robust, having mechanisms to cancel or fail the currently running failover and restart a new one, such as shown in the Route Compute injections. The failover can also defer the handling of the second fault until after the first failover completes, as shown in most other stage injections. For injections during earlier stages, such as Alive or Alive2, the currently running failover can handle the second failure without needing to start a second failover process as new routes have yet to be computed. These observations point to the likelihood that even in the presence of more than two faults, the failover process will still behave similarly and encounter no further issue on JYC. Additionally, by restarting the failover or starting a second failover, the recovery time does inevitably increase. An increased time in failover recovery means an increase in the probability of the occurrence of additional failures, which may lead to catastrophic failures in the system. According to [5], if the recovery time exceeds 300 seconds, the probability of additional failures raises to 0.8.

The mentioned injection misses were predominantly due to connection injections, due to the brief durations of many recovery stages and the nature of having multiple commands to send, which causes a delay in the time to injection. The small time window of recovery stages, which is a property of a small-scale system, coupled with the latency of the injector to detect target recovery stages and to inject the second fault makes this campaign highly challenging. We suspect that these are the main culprits for why we cannot reproduce the failure scenarios (e.g. network deadlocks) observed in field data from large-scale production systems. While there may be slight optimizations still available in the injector code (e.g. a queue based logging system), the recovery window and the inherent delay of the SMW's logging leave little room for improvement. Any optimization in the injector would likely be negligible. There is simply not enough time for faults to propagate throughout the system and create adverse effects on JYC.

5.4 Future Work

The future of this work has many branches, including further development of HPCArrow to support a variety of software and systems beyond Cray, deeper investigation of hung applications and real-time detection of such phenomenon, more fault models such as memory and network congestion, and more experiments on multiple faults scenarios.

Future work has already begun towards resiliency comparisons between Gemini and Aries interconnects. HPCArrow currently can support fault injections on Aries interconnects and Cray XC machines. This development occurred using SNL's Cray XC testbeds, Voltrino and Mutrino, and fault injection experiments are currently underway. Aries and Gemini are not exactly one-to-one mappings due to architectural differences. For example, there are four types of links on Aries (green, blue, black, and tweak) as opposed to the general link on Gemini. Despite this, the fault models remain generally the same. The same LDMS traffic data and system logs can still be collected since these are still Cray systems. However, again, due to architectural differences in the network fabric, analysis code will need to be altered and tailored to the components that LDMS is measuring. Ultimately, the fault injection approach remains the same and we can retain the automation, control, and reproducible advantages that HPCArrow provides. Expanding to other systems beyond Cray may be possible as long as topology mappings are provided per system and there are software mechanisms to induce failures, recover, measure traffic, and monitor system behaviors and responses to failures. Since HPCArrow is built using Python, it may be as easy as including more system-specific commands. Separate programs or scripts written to trigger faults can also be incorporated into HPCArrow. Applications are a major overhead in porting HPCArrow to other systems as benchmarks will need to be recompiled and reconfigured for each system and their software.

Since this work could not cause failed failovers and reproduce critical system errors, future work in the space of injections during recovery should involve testing and running these experiments on larger scale systems. Our hypothesis is that since small-scale HPC systems have such short windows of recovery, there is not enough time for failures and their effects to propagate throughout the system. Large-scale systems such as Blue Waters have much longer windows of recovery time. A related hypothesis is that there may be a window of vulnerability in the failover procedure that is either nonexistent or very short in duration on small-scale systems, but much longer on large-scale systems. One way to test these hypotheses on a small-scale system like JYC would be to investigate methods to create artificial delays during the recovery stages. Another avenue of investigation is to determine whether such a window of vulnerability exists by examining data generated by large-scale systems and then to quantify a relationship between this window of vulnerability and the system scale.

5.5 Conclusion

In this work, we developed a software-implemented fault injection tool called HPCArrow as well as a fault injection methodology that can be used to assess fault tolerance and resiliency of HPC systems. We demonstrated HPCArrow's capabilities through four fault injection campaigns, covering single injections, time-varying or delayed injections, and injections during recovery. These injections induce failures on network and compute components. This fault injection tool and methodology can be expanded to other systems. It has currently been extended to Cray XC systems and Aries interconnects. In addition to demonstrating HPCArrow, these campaigns also provided insights into application-level resiliency and system-level resiliency. There are notable deficiencies in fault tolerance across various HPC application frameworks, most severely for Charm++ applications. Our experiments revealed a failure phenomenon of application hangs in which forward progress is not made, but jobs are not terminated until the maximum allowed time has elapsed. At the system level, failover procedures are very robust and able to handle both single and multiple faults in the network, having an arsenal of responses to these various scenarios.

REFERENCES

[1] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 129–173, May 2014. [Online]. Available: http://dx.doi.org/10.1177/1094342014522573

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

[3] C. D. Martino, W. Kramer, Z. Kalbarczyk, and R. Iyer, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, pp. 25–36.

[4] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '14. Washington, DC, USA: IEEE Computer Society, 2014. [Online]. Available: https://doi.org/10.1109/DSN.2014.62 pp. 610–621.

[5] S. Jha, V. Formicola, C. D. Martino, M. Dalton, W. T. Kramer, Z. Kalbarczyk, and R. K. Iyer, "Resiliency of HPC interconnects: A case study of interconnect failures and recovery in Blue Waters," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2017.

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

[7] M. Kumar, S. Gupta, T. Patel, M. Wilder, W. Shi, S. Fu, C. Engelmann, and D. Tiwari, "Understanding and analyzing interconnect errors and network congestion on a large scale HPC system," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 107–114.

[8] M. A. Ezell, "Understanding the impact of interconnect failures on system operation," Jan 2013. [Online]. Available: https://cug.org/proceedings/cug2013_proceedings/includes/files/pap140.pdf

[9] K. Chadalavada and R. Sisneros, "Analysis of the Blue Waters file system architecture for application I/O performance," *Cray User Group*, Jan 2013.

[10] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Evaluating and accelerating high-fidelity error injection for HPC," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291716 pp. 45:1–45:13.

[11] J. Calhoun, L. N. Olson, and M. Snir, "FlipIt: An LLVM based fault injector for HPC," in *Euro-Par Workshops*, 2014.

[12] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-SEFI: A fine-grained soft error fault injection tool for profiling application vulnerability," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 1245–1254.

[13] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. M. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing performance variations in HPC applications using machine learning," in *ISC*, 2017.

[14] A. Netti, Z. Kiziltan, O. Babaoglu, A. Srbu, A. Bartolini, and A. Borghesi, "FINJ: A fault injection tool for HPC systems," 08 2018.

[15] V. Formicola, S. Jha, D. Chen, F. Deng, A. Bonnie, M. Mason, J. Brandt, A. Gentile, L. Kaplan, J. Repik, J. Enos, M. Showerman, A. Greiner, Z. Kalbarczyk, R. Iyer, and W. Kramer, "Understanding fault scenarios and impacts through fault injection experiments in Cielo," in *Cray User Group*, May 2017.

[16] "Blue Waters user portal — system summary." [Online]. Available: https://bluewaters.ncsa.illinois.edu/hardware-summary

[17] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini system interconnect," in *2010 18th IEEE Symposium on High Performance Interconnects*, Aug 2010, pp. 83–87.

[18] *Managing System Software for Cray XE and Cray XT Systems*, S239331 ed., Cray, Inc, 2010.

[19] "Blue Waters user portal — running your jobs." [Online]. Available: https://bluewaters.ncsa.illinois.edu/running-your-jobs

[20] "TORQUE resource manager." [Online]. Available: http://www.adaptivecomputing.com/products/torque/

[21] "Moab cloud HPC suite." [Online]. Available: http://www.adaptivecomputing.com/moab-hpc-basic-edition/

[22] M. Karo, R. Lagerstrom, M. Kohnke, and C. Albing, "The application level placement scheduler," *Cray User Group*, May 2006.

[23] *Network Resiliency for Cray XE and Cray XK Systems*, S0032d ed., Cray, Inc, 2014.

[24] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," Tech. Rep. White Paper WP-Aries01-1112, 2012. [Online]. Available: https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf

[25] "Blue Waters user portal — benchmarks." [Online]. Available: https://bluewaters.ncsa.illinois.edu/benchmarks

[26] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, Oct 1993. [Online]. Available: http://doi.acm.org/10.1145/167962.165874

[27] Y. Sun, G. Zheng, L. V. Kal, T. R. Jones, and R. Olson, "A uGNI-based asynchronous message-driven runtime system for cray supercomputers with Gemini interconnect," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 751–762.

[28] "Charm++: Parallel programming framework." [Online]. Available: http://charm.cs.illinois.edu/manuals/html/charm++/21.html

[29] "Parallel views: Newsletter of the parallel programming laboratory." [Online]. Available: http://charm.cs.uiuc.edu/docs/PPLnewsletter_fall2012-forweb.pdf

[30] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, Sept 1996. [Online]. Available: http://dx.doi.org/10.1016/0167-8191(96)00024-5

[31] W. Gropp and E. Lusk, "Fault tolerance in message passing interface programs," *The International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004. [Online]. Available: https://doi.org/10.1177/1094342004046045

[32] G. Almasi, *PGAS (Partitioned Global Address Space) Languages.* Boston, MA: Springer US, 2011, pp. 1539–1545.

[33] "SMW daemons, processes, and logs." [Online]. Available: https://pubs.cray.com/content/S-2565/CLE%206.0.UP07/xctm-series-boot-troubleshooting-guide/smw-daemons-processes-and-logs

[34] 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*, ser. FTXS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2751504.2751511 pp. 11–18.

[35] 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, "The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014. [Online]. Available: https://doi.org/10.1109/SC.2014.18 pp. 154–165.

[36] L. R. Devnani, "Fault injections on mission critical systems," M.S. thesis, University of Illinois at Urbana-Champaign, 2018.

[37] "Blue Waters user portal — FAQ." [Online]. Available: https://bluewaters.ncsa.illinois.edu/faq

[38] *XC Series GNI and DMAPP API User Guide*, S-2446 ed., Cray, Inc, 2017.

[39] "Slurm workload manager." [Online]. Available: https://slurm.schedmd.com/

APPENDIX A

JYC COMPONENT NAMES

Table A.1: JYC component names for each component type. * means this is our naming convention for identifying connections in this work as there is no real cname for connections on the system.

Component Type	**Component Name**	**Description**
Cabinet	c0-0	Cabinet 0 (row 0-column 0)
Chassis	c0-0c2	Chassis 2 in Cabinet 0
Blade	c0-0c0s5	Blade 5 in Chassis 0 of Cabinet 0
Node	c0-0c1s3n3	Node 3 on Blade 3 in Chassis 1 of Cabinet 0
Gemini	c0-0c2s1g0	Gemini 0 on Blade 1 in Chassis 2 of Cabinet 0
Link	c0-0c2s1g1l16	Link 16 (row 1, col 6 in tile array) on Gemini 1 on Blade 1 in Chassis 2 of Cabinet 0
*Connection	c0-0c2s1g1,Z-	Z- Connection (two torus connections) on Gemini 1 on Blade 1 in Chassis 2 of Cabinet 0

APPENDIX B

CRAY INJECTION COMMANDS

The following tables detail the commands executed by HPCArrow's Injection Manager on the SMW. Note that the restoration commands in Table B.2 must be run as the *crayadm* user on the SMW.

Table B.1: Cray fault injection commands executed by HPCArrow.

Fault Type	Command	Additional Information
Link Connection	`xtmemio -w ::{gemini} {0x0006000128 | (link row << 22) | (link col << 19)} 2 0`	{gemini}: Target Gemini's cname. {link row} and {link col}: Row and column (from tile array) of target link For example, a target link "c0-0c1s0g1l27" parameters are: {gemini} = "c0-0c1s0g1" {link row} = 2 {link col} = 7 For connections, the command is repeated for each link in the target connection direction. This would be repeated 4x for Y+/Y- connections or 8x for Z+/Z- connections.
Node	`xtnmi {node}`	{node}: Target node cname. For example, {node} = "c0-0c2s0n3" Note: Must run as crayadm
Blade	`rsh -l root {blade} "/opt/bin/i2c 2:0x60/2=0x02,0x00"`	{blade}: Target blade cname. For example, {blade} = "c0-0c2s0"

Table B.2: Cray warm swap restoration commands executed by HPCArrow.

Fault Type	Command	Additional Information
Link	`xtwarmswap -s {link}, {link end} -p p0`	{link}: Target link cname {link end}: Target link's other end cname
Connection	`xtwarmswap -s {links} -p p0`	{links}: Comma separated list of links to restore from target connection. The other ends of target connection must also be specified.
Node	`xtbootsys --reboot -L CNL0 {node}`	{node}: Target node cname. Note: Requires an interactive terminal session on the SMW.
Blade	Remove: `xtwarmswap --force --remove {blade}` Add: `xtwarmswap --add {blade}` Boot: `xtcli boot CNL0 {blade}`	{blade}: Target blade cname. Remove, add, and boot commands must be executed in sequence.

APPENDIX C

JOB NAME PARTS

Due to restricted access to the SMW, we extended HPCArrow's functionality to allow us to remotely trigger injections as normal JYC users. To achieve this, we remotely communicate with HPCArrow's Injection Manager via the job names of applications that we submit to the job scheduler. All applications of a run are specified with the same trailing string in the job name.

Figure C.1 provides an example job name submitted to JYC and broken down into its parts. The parts are explained below. Note that * means it is required, otherwise the injection does not execute. For more than two injections, this method is not scalable due to limitations on the length of job names.

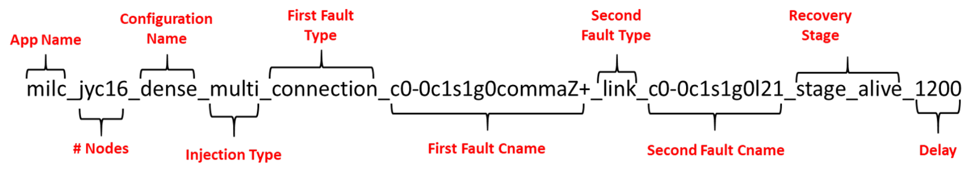


Figure C.1: Breakdown of the parts of an example job name submitted to the ALPS.

App Name*

The name of the application. This is pre-populated by the Workload Generator.

# Nodes*

The number of nodes the application is running on. This is pre-populated by the Workload Generator.

Configuration Name*

The name of the configuration set of applications that was submitted together. This is pre-populated by the Workload Generator.

Injection Type*

Tells the injector to perform single or multiple injection(s). User-specified. If not specified, the injection cancels.

First Fault Type*

Tells the injector what type of component to initially fail. User-specified. If not specified, the injection cancels.

First Fault Cname*

Provides the first fault's cname, which defines the component's physical location in the system. User-specified. If not specified, a random component of the specified fault type is selected.

Second Fault Type

Tells the injector what type of component to fail second if performing multiple injections.

Second Fault Cname

Provides the second fault's cname.

Recovery Stage

Tells the injector to fail the second component during recovery. If not specified and a second fault is still specified, the injector will inject the fault after recovery. Note "stage" must be specified to indicate fault during recovery.

Injection Delay

Tells the injector to fail the first fault after some delay time (in seconds) from the start of the experiment. This delay only applies to the second fault injection if it is not a recovery injection.

APPENDIX D

APPLICATION CONFIGURATIONS

In this work, nine unique applications are compiled and configured to run as workloads on JYC. Campaign I uses a subset of these nine, Application Set I: AWP-ODC, AMR (SMP, HugePages), Kripke (HugePages), LeanMD (HugePages), and UPC-FT. Campaigns II to IV use a different subset, Application Set II: AMR (HugePages), Kripke (HugePages), LeanMD (HugePages), MILC, NAMD (SMP), and PSDNS. For each set, applications are selected, scaled, and placed on the JYC system map. This is a configuration. Campaign I uses five configurations drawn from Set I: Configuration 1 to 5. Campaigns II to IV draw from Set II to create three configurations: Dense, Medium, and Sparse Configurations.

Application Name
: The name of the benchmark. Charm++ applications are distinguished between symmetric multiprocessing (SMP) and HugePages.

Application Size
: The number of nodes that the application is running on.

Node Type
: The node type the application is using: XE, XK, or both.

Processes Per Node (PPN)
: Cray XE nodes support 32 processes; XK nodes support 16. If an application runs on a combination of XE and XK nodes, then the PPN is limited by the node with the smallest number of processes.

Node IDs
: The name identifiers of nodes on which the application runs.

Application Specific Parameters
: Benchmark-specific parameters to adjust for scaling and run time.

Table D.1: Application Set I, Configuration 1 parameters.

App Name	App Size	Node Type	PPN	Node IDs	Parameters
AWP-ODC	32	XE	16	32-63	NX = NY = NX = 712 NPX = NPY = NPZ = 8
Kripke (HugePages)	8	XK	16	66-69,90-93	NITER = 13
LeanMD (HugePages)	8	XK	16	72-75,84-87	steps = 1900
UPC-FT	4	XE	32	72-75,84-87	steps = 1900 NX = 4 NY = 32

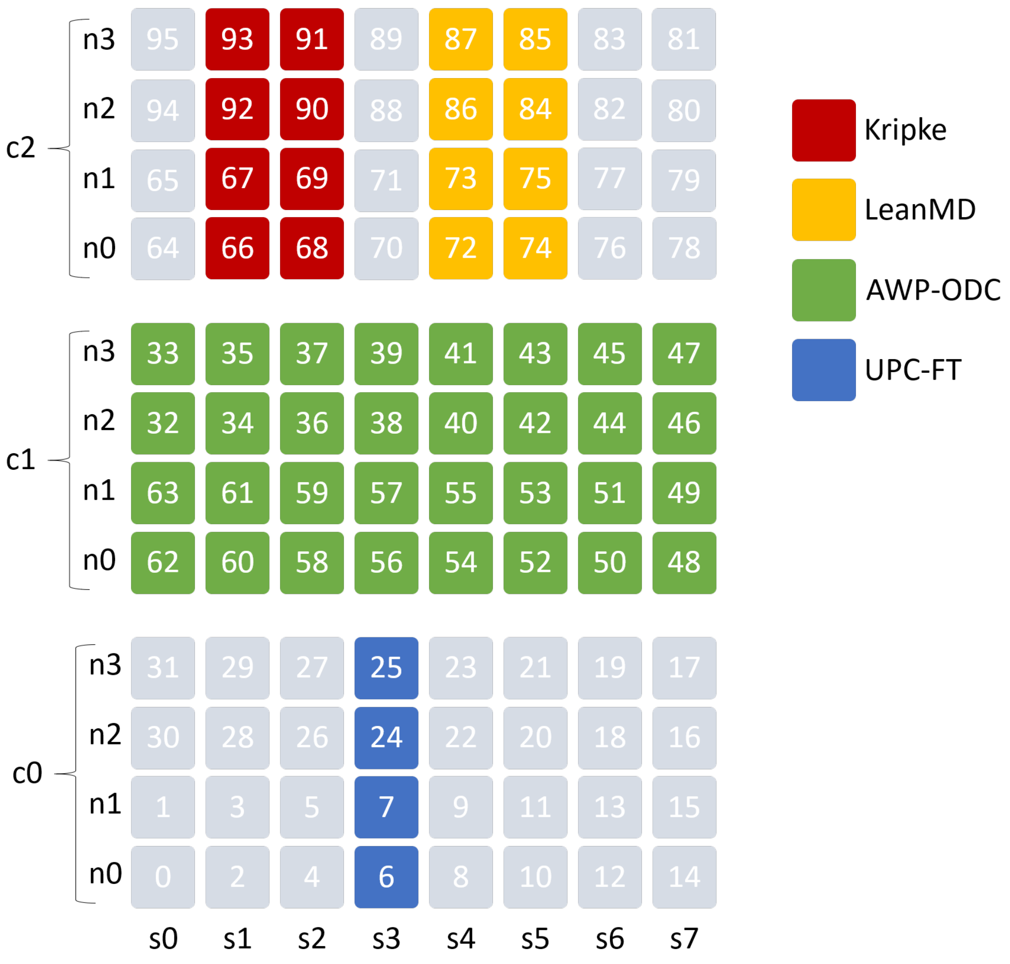


Figure D.1: Application Set I, Configuration 1 JYC system mapping.

Table D.2: Application Set I, Configuration 2 parameters.

App Name	**App Size**	**Node Type**	**PPN**	**Node IDs**	**Parameters**
AMR (HugePages)	4	XE	32	2,3,28,29	ITERATIONS = 175
AMR (SMP)	32	XE	32	32-63	ITERATIONS = 800
AWP-ODC	2	XE	32	32-63	NX = NY = NX = 368 NPX = NPY = NPZ = 4
Kripke (HugePages)	4	XE	32	6,7,24,25	NITER = 13
LeanMD (HugePages)	2	XE	32	11,20	steps = 950
LeanMD (HugePages)	4	XE	32	12,13,18,19	steps = 1775
UPC-FT	32	XE/XK	16	64-95	steps = 76000 NX = 32 NY = 16

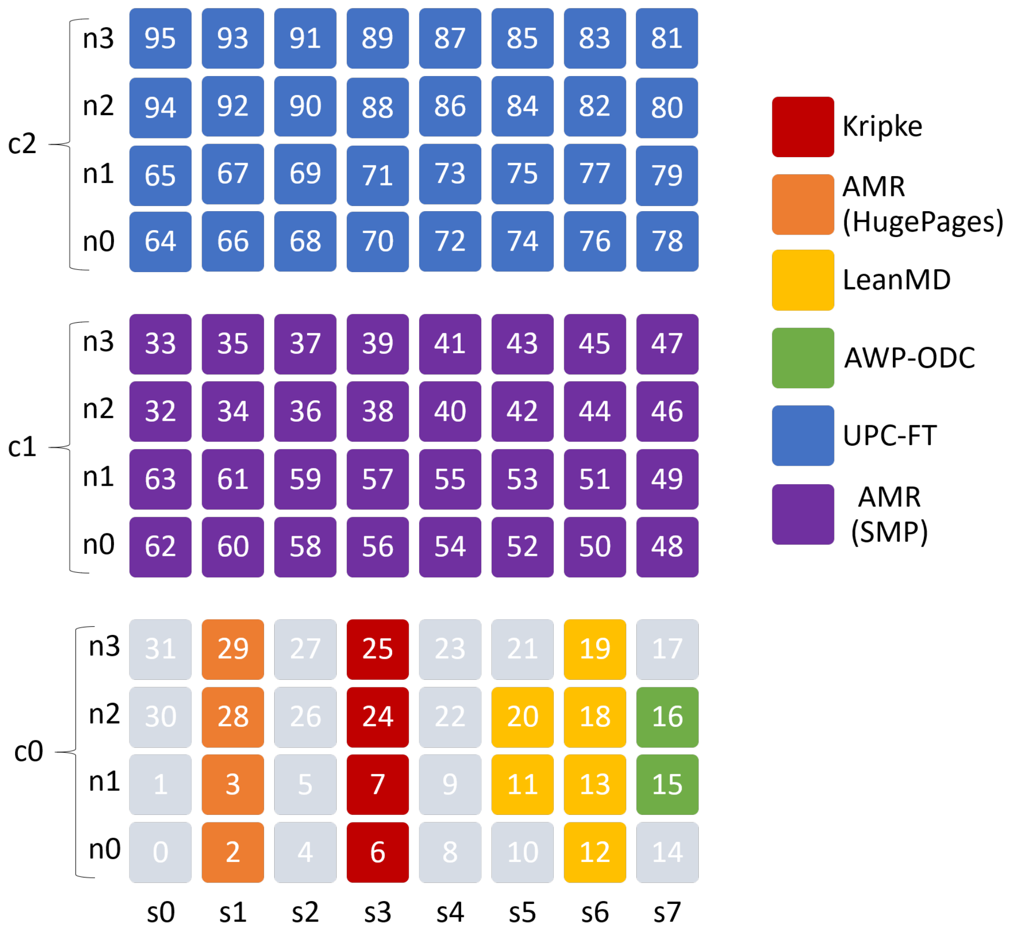


Figure D.2: Application Set I, Configuration 2 JYC system mapping.

Table D.3: Application Set I, Configuration 3 parameters.

App Name	**App Size**	**Node Type**	**PPN**	**Node IDs**	**Parameters**
AMR (HugePages)	2	XE	32	60,61	ITERATIONS = 95
AWP-ODC	4	XK	16	74,75,84,85	NX = NY = NX = 364 NPX = NPY = NPZ = 4
LeanMD (HugePages)	4	XE	32	6,7,24,25	steps = 1775
UPC-FT	2	XE	32	16,17	steps = 12500 NX = 2 NY = 32

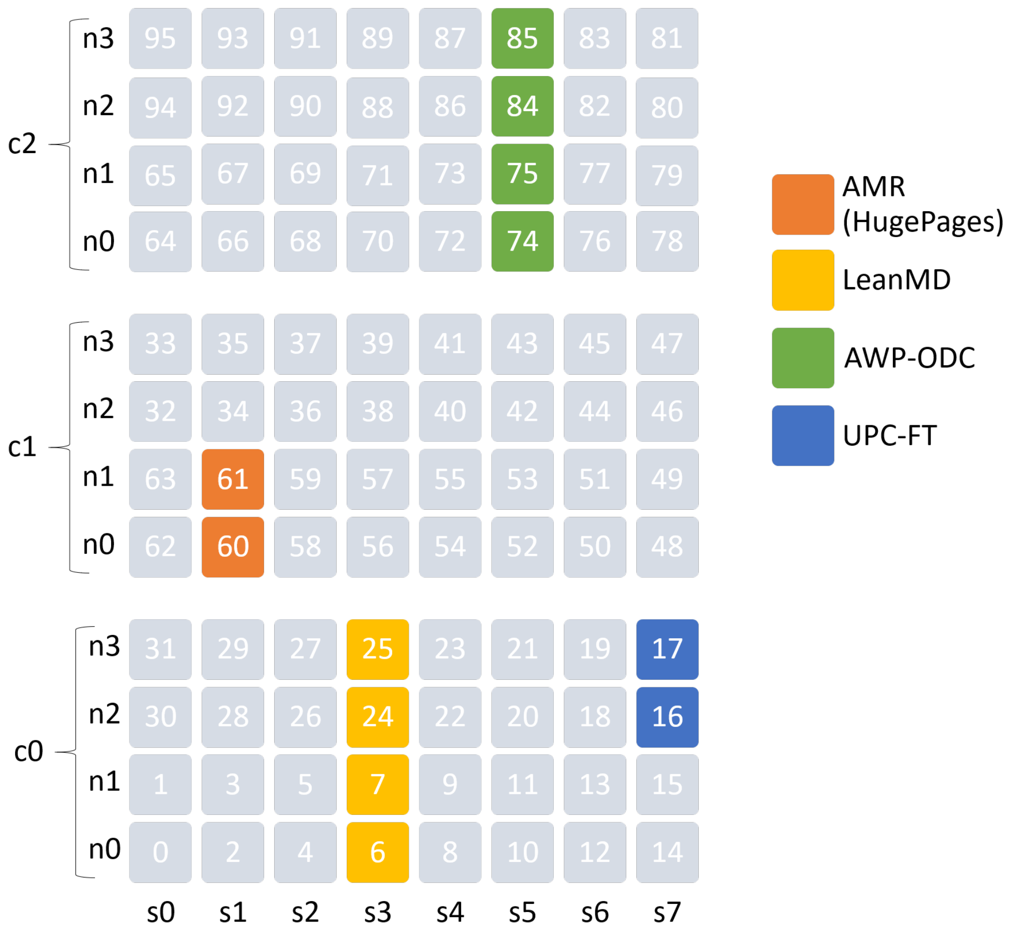


Figure D.3: Application Set I, Configuration 3 JYC system mapping.

Table D.4: Application Set I, Configuration 4 parameters.

App Name	App Size	Node Type	PPN	Node IDs	Parameters
AMR (HugePages)	2	XE	32	16,17	ITERATIONS = 95
AWP-ODC	32	XE/XK	16	64-95	NX = NY = NX = 712 NPX = NPY = NPZ = 8
Kripke (HugePages)	2	XE	32	2,3	NITER = 13
UPC-FT	8	XE	32	34-37,58-61	steps = 32000 NX = 8 NY = 32

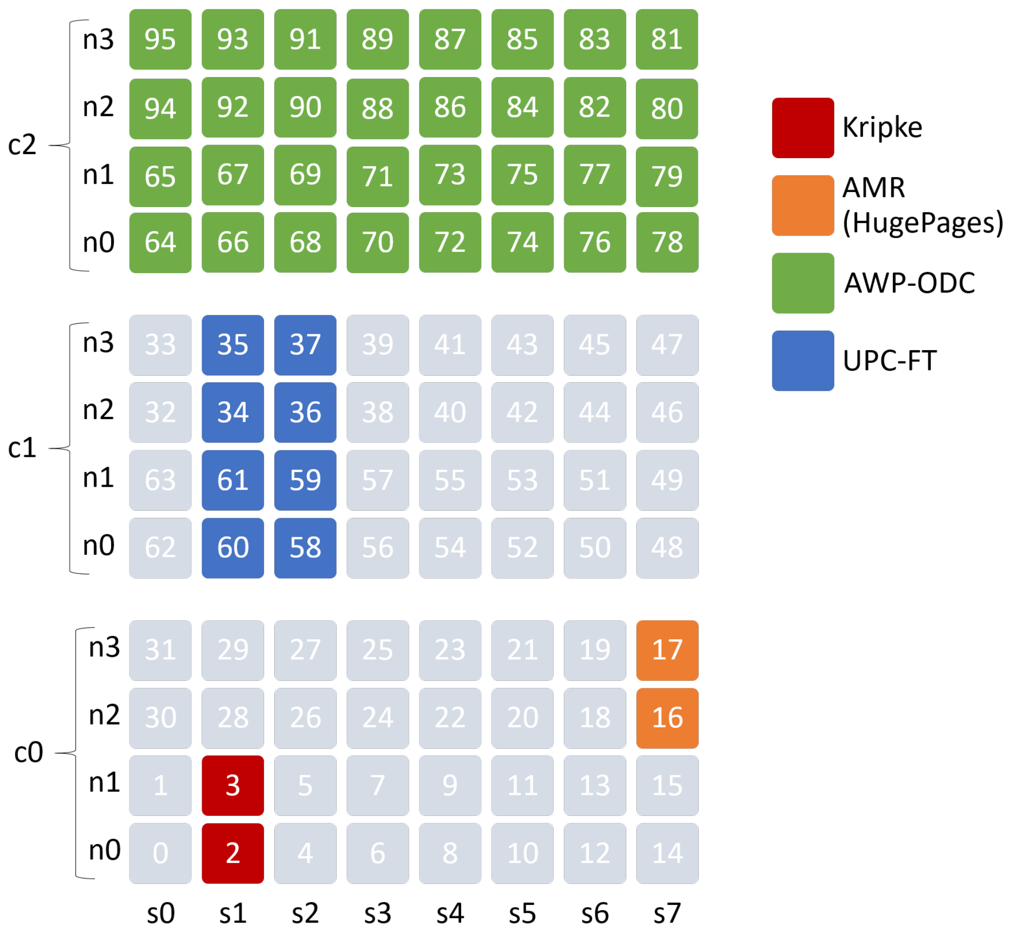


Figure D.4: Application Set I, Configuration 4 JYC system mapping.

Table D.5: Application Set I, Configuration 5 parameters.

App Name	App Size	Node Type	PPN	Node IDs	Parameters
AMR (SMP)	64	XE/XK	16	32-95	ITERATIONS = 800
AWP-ODC	2	XE	32	18,19	NX = NY = NX = 368 NPX = NPY = NPZ = 4
UPC-FT	4	XE	32	6,7,24,25	steps = 21500 NX = 4 NY = 32

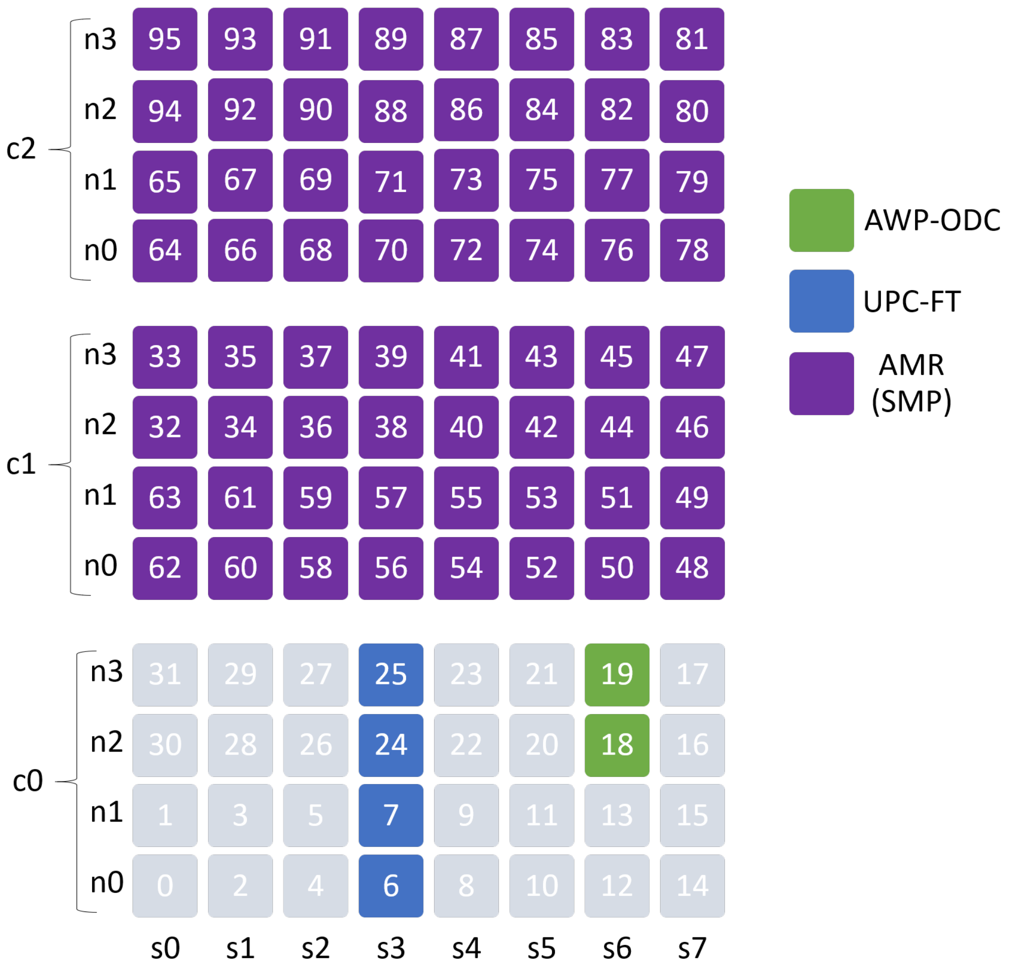


Figure D.5: Application Set I, Configuration 5 JYC system mapping.

Table D.6: Application Set II, Dense Configuration parameters.

App Name	App Size	Node Type	PPN	Node IDs	Parameters
AMR (HugePages)	2	XE	32	20,21	ITERATIONS = 95
Kripke (HugePages)	4	XE	32	6,7,24,25	NITER = 14
LeanMD (HugePages)	16	XE	32	40-55	steps = 7000
MILC	2	XE	32	16,17	NX = NY = NZ = NT = 16 Trajectories = 2
MILC	16	XE	32	32-39,56-63	NX = NY = NZ = NT = 32 Trajectories = 1
NAMD (SMP)	4	XE	32	6,7,24,25	numsteps = 10700
PSDNS	4	XE	32	12,13,18,19	dims: 4 32 nsteps = 46
PSDNS	32	XE/XK	32	64-95	dims:16 64 nsteps = 320

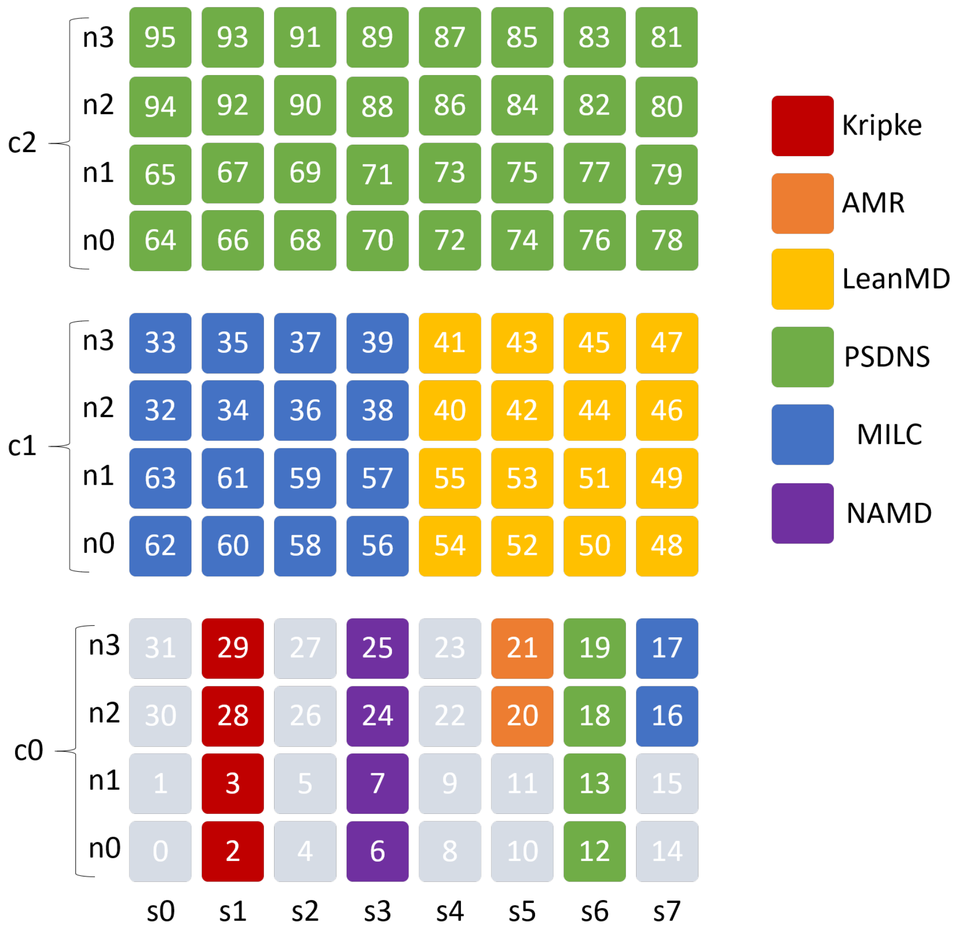


Figure D.6: Application Set II, Dense Configuration JYC system mapping.

Table D.7: Application Set II, Medium Configuration parameters.

App Name	App Size	Node Type	PPN	Node IDs	Parameters
AMR (HugePages)	4	XE	32	12,13,18,19	ITERATIONS = 175
Kripke (HugePages)	8	XK	16	72-75,84-87	NITER = 13
LeanMD (HugePages)	8	XK	16	64-67,92-95	steps = 1900
MILC	4	XE	32	2,3,28,29	NX = NY = NZ = NT = 16 Trajectories = 4
NAMD (SMP)	16	XE	32	32-39,56-63	numsteps = 40500
PSDNS	8	XE	32	44-51	dims: 8 32 nsteps = 100

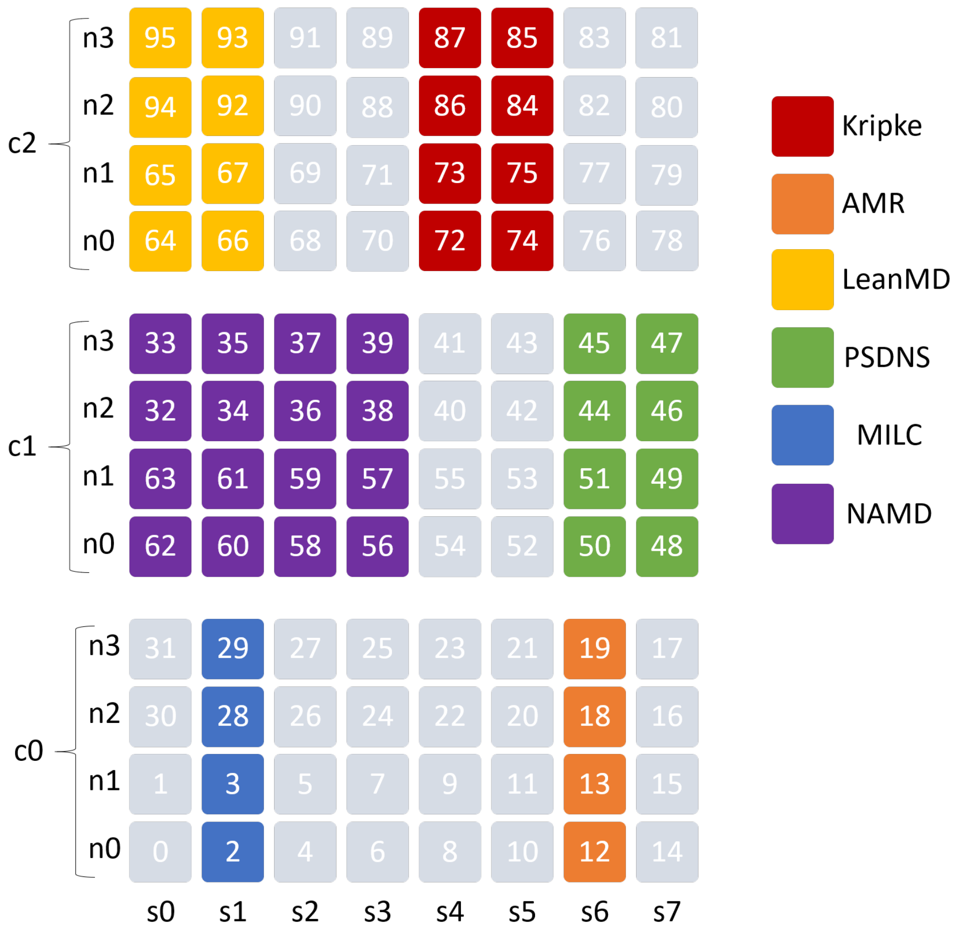


Figure D.7: Application Set II, Medium Configuration JYC system mapping.

Table D.8: Application Set II, Sparse Configuration parameters.

App Name	App Size	Node Type	PPN	Node IDs	Parameters
AMR (HugePages)	4	XK	16	64,65,94,95	ITERATIONS = 175
Kripke (HugePages)	8	XK	16	70,71,88,89	NITER = 14
LeanMD (HugePages)	2	XE	32	48,49	steps = 950
MILC	4	XE	32	78-81	NX = NY = NZ = NT = 16 Trajectories = 4
NAMD (SMP)	4	XE	32	36,37,58,59	numsteps = 10700
PSDNS	4	XE	32	42,43,52,53	dims: 4 32 nsteps = 46

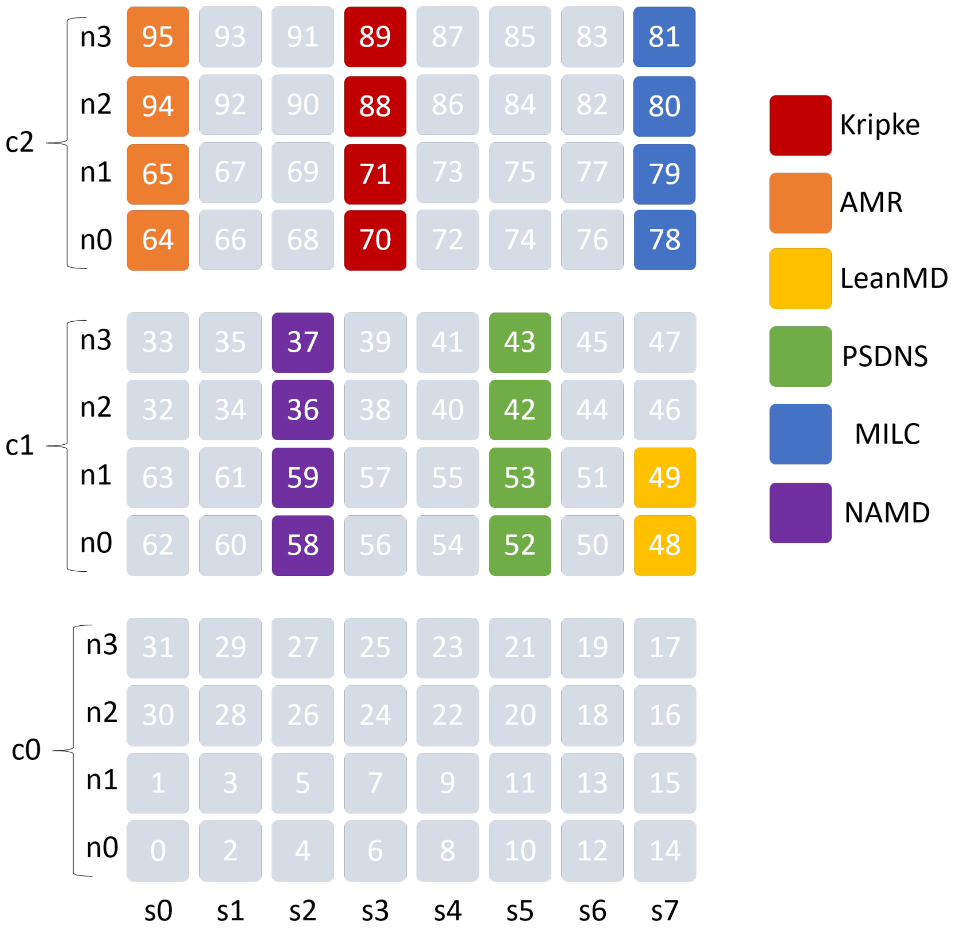


Figure D.8: Application Set II, Sparse Configuration JYC system mapping.

APPENDIX E

FAULT INJECTIONS SELECTED

The following tables list the components that were targeted during Campaigns II to IV based on LDMS traffic data. For each configuration in Application Set II, every application is examined. For each application, the Gemini and corresponding torus connection direction with the highest throughput is selected as the base target component for a connection fault. The link, node, and blade faults simply follow from selection of the target Gemini and connection. Note that for NAMD in the Medium Configuration and MILC in the Dense Configuration, there is a second row. These are the targets for injections during recovery, based on the Gemini targeted in the first fault and the traffic data on that same Gemini.

Table E.1: Target components of the Dense Configuration for each fault type, selected based on the highest throughput connection direction for application runs without any injections.

App Name	**Connection**	**Links**	**Blade**
LeanMD (HugePages)	c0-0c1s6g1,Z-	c0-0c1s6g1l07 c0-0c1s6g1l17 c0-0c1s6g1l22	c0-0c1s6
MILC	c0-0c1s1g0,Z+	c0-0c1s1g0l00 c0-0c1s1g0l25 c0-0c1s1g0l35	c0-0c1s1
	c0-0c1s1g0,Z-	c0-0c1s1g0l21	-
PSDNS	c0-0c2s1g1,Z-	c0-0c2s1g1l16 c0-0c2s1g1l21 c0-0c2s1g1l32	c0-0c2s1

Table E.2: Target components of the Medium Configuration for each fault type, selected based on the highest throughput connection direction for application runs without any injections.

App Name	Connection	Link(s)	Blade
Kripke (HugePages)	c0-0c2s5g0,Z-	c0-0c2s5g0l06 c0-0c2s5g0l07 c0-0c2s5g0l20	c0-0c2s5
LeanMD (HugePages)	c0-0c2s1g1,Z-	c0-0c2s1g1l06 c0-0c2s1g1l20 c0-0c2s1g1l32	c0-0c2s1
NAMD (SMP)	c0-0c1s0g1,Z+	c0-0c1s0g1l01 c0-0c1s0g1l10 c0-0c1s0g1l27	c0-0c1s0
	c0-0c1s0g1,Y-	c0-0c1s0g1l10	-
PSDNS	c0-0c1s6g0,Y+	c0-0c1s6g0l55 c0-0c1s6g0l56 c0-0c1s6g0l57	c0-0c1s6

Table E.3: Target components of the Sparse Configuration for each fault type, selected based on the highest throughput connection direction for application runs without any injections.

App Name	**Connection**	**Links**	**Blade**
AMR (HugePages)	c0-0c2s0g1,Y-	c0-0c2s0g1l42 c0-0c2s0g1l51 c0-0c2s0g1l52	c0-0c2s0
Kripke (HugePages)	c0-0c2s3g1,Y-	c0-0c2s3g1l50 c0-0c2s3g1l51 c0-0c2s3g1l52	c0-0c2s3
LeanMD (HugePages)	c0-0c1s7g0,Y+	c0-0c1s7g0l55 c0-0c1s7g0l56 c0-0c1s7g0l57	c0-0c1s7
NAMD (SMP)	c0-0c1s2g1,Y-	c0-0c1s2g1l42 c0-0c1s2g1l51 c0-0c1s2g1l50	c0-0c1s2
MILC	c0-0c2s7g0,Y+	c0-0c2s7g0l45 c0-0c2s7g0l55 c0-0c2s7g0l57	c0-0c2s7
PSDNS	c0-0c1s5g0,Y+	c0-0c1s5g0l45 c0-0c1s5g0l56 c0-0c1s5g0l57	c0-0c1s5

APPENDIX F

CAMPAIGN I FULL SUMMARY

Table F.1: Full summary of Campaign I, showing the outcomes of applications in the presence of (direct and indirect) faults.

Benchmark	**Application**	**Fault Type**	**Run Status #(%)**		
			Crash	Hang	No Impact
Charm++ (HugePages)	AMR	Blade	2 (22.22)	0	7 (77.78)
		Connection	0	0	9 (100.00)
		Link	0	0	9 (100.00)
		No Injection	0	0	9 (100.00)
	Kripke	Blade	1 (10.00)	0	9 (90.00)
		Connection	0	1 (10.00)	9 (90.00)
		Link	0	2 (20.00)	8 (80.00)
		No Injection	0	0	9 (100.00)
	LeanMD	Blade	3 (23.08)	0	10 (76.92)
		Connection	2 (15.38)	0	11 (84.62)
		Link	0	1 (7.69)	12 (92.31)
		No Injection	0	0	12 (100.00)
Charm++ (SMP)	AMR	Blade	5 (83.33)	0	1 (16.67)
		Connection	5 (83.33)	0	1 (16.67)
		Link	4 (66.67)	0	2 (33.33)
		No Injection	0	0	6 (100.00)
MPI	AWP-ODC	Blade	3 (18.75)	0	13 (81.25)
		Connection	0	0	16 (100.00)
		Link	0	0	16 (100.00)
		No Injection	0	0	15 (100.00)
PGAS	UPC-FT	Blade	2 (12.50)	0	14 (87.50)
		Connection	2 (12.50)	0	14 (87.50)
		Link	3 (18.75)	0	13 (81.25)
		No Injection	0	0	15 (100.00)

APPENDIX G

CAMPAIGN II FULL SUMMARY

Table G.1: Full summary of Campaign II, showing the outcomes of applications in the presence of (direct and indirect) faults.

Benchmark	**Application**	**Fault Type**	**Run Status #(%)**		
			Crash	Hang	No Impact
Charm++ (HugePages)	AMR	Blade	5 (15.62)	0	27 (84.38)
		Connection	6 (15.79)	4 (10.53)	28 (73.68)
		Link	3 (5.77)	1 (1.92)	48 (92.31)
		No Injection	0	0	30 (100.0)
	Kripke	Blade	5 (13.51)	0	32 (86.49)
		Connection	5 (13.16)	1 (2.63)	32 (84.21)
		Link	0	2 (3.85)	50 (96.15)
		No Injection	0	0	30 (100.0)
	LeanMD	Blade	5 (13.51)	0	32 (86.49)
		Connection	3 (7.89)	2 (5.26)	33 (86.84)
		Link	0	5 (9.62)	47 (90.38)
		No Injection	0	0	30 (100.0)
Charm++ (SMP)	NAMD	Blade	5 (13.51)	0	32 (86.49)
		Connection	1 (2.63)	4 (10.53)	33 (86.84)
		Link	1 (1.92)	10 (19.23)	41 (78.85)
		No Injection	0	0	30 (100.0)
MPI	MILC	Blade	5 (9.26)	0	49 (90.74)
		Connection	0	0	55 (100.0)
		Link	4 (6.45)	0	58 (93.55)
		No Injection	0	0	40 (100.0)
	PSDNS	Blade	13 (22.03)	0	46 (77.97)
		Connection	0	0	55 (100.0)
		Link	5 (7.58)	0	61 (92.42)
		No Injection	0	0	40 (100.0)

APPENDIX H

CAMPAIGN IV DISTRIBUTIONS SUMMARY

Table H.1: Summary statistics for the Nlrd Log Delay, Time to Detection (link vs. connection), Time to Injection (link vs. connection), Recovery Durations (single fault vs. multiple faults) reported in Campaign IV. All numbers are reported in seconds except for Count.

Distribution Name	Injection or Fault Type	Count	Mean	Std	Min	25%	50%	75%	Max
Nlrd Log Delay	-	7072	1.004627	0.400815	0.046389	0.730470	1.000891	1.285352	1.987961
Time to Detection	Blade	11	0.297091	0.084562	0.227000	0.244000	0.255000	0.318000	0.502000
	Connection	28	1.429964	0.584833	0.252000	1.043750	1.436500	1.795000	2.771000
	Link	28	1.405214	0.699859	0.224000	0.823500	1.310000	1.869250	2.793000
Time to Injection	Blade	11	0.297091	0.084562	0.227000	0.244000	0.255000	0.318000	0.502000
	Connection	28	1.425679	0.761596	0.475000	0.647750	1.387500	2.139250	2.975000
	Link	28	0.336893	0.459619	0.116000	0.129250	0.141000	0.212500	1.849000
Recovery Durations	Single	127	39.883707	2.648896	35.040480	38.029916	39.044034	41.035962	51.047644
	Multiple	56	66.665411	15.441623	35.027203	55.065860	76.079558	78.078578	84.081822