Reducing Waste in Extreme Scale Systems through Introspective Analysis

Leonardo Bautista-Gomez*, Ana Gainaru†‡, Swann Perarnau*, Devesh Tiwari§,
Saurabh Gupta§, Christian Engelmann§, Franck Cappello* and Marc Snir*†
*Argonne National Laboratory
†University of Illinois at Urbana-Champaign
‡National Center for Supercomputing and Applications
§Oak Ridge National Laboratory

***Abstract*—Resilience is an important challenge for extreme-scale supercomputers. Today, failures in supercomputers are assumed to be uniformly distributed in time. However, recent studies show that failures in high-performance computing systems are partially correlated in time, generating periods of higher failure density. Our study of the failure logs of multiple supercomputers show that periods of higher failure density occur up to three times more than average. We design a monitoring system that listens to hardware events and forwards important events to the runtime to detect those regime changes. We implement a runtime capable of receiving notifications and adapt dynamically. In addition, we build an analytical model to predict the gains that such dynamic approach could achieve. We demonstrate that in some systems, our approach can reduce the wasted time by over 30%.**

***Keywords*-Supercomputers, Fault Tolerance, Resilience, Silent Data Corruption, Soft Errors, Introspective Systems.**

I. Introduction

High-performance computing (HPC) is changing the way scientists make discoveries. Future-generation supercomputers will achieve unprecedented computational performance, thanks to their larger number of components and enhanced parallelism. However, more components and more system complexity also bring higher failure rates[7], [27]. To deal with failures, HPC systems implement checkpoint/restart (CR). Scientific applications are checkpointed to reliable storage and restarted from the last checkpoint upon a failure. The frequency at which a system should checkpoint is an optimization problem. Checkpointing too frequently will produce unnecessary waste, and checkpointing not frequently enough will produce long re-executions. The optimal checkpoint interval is the subject of numerous research works [32], [11]. It is generally found by using two parameters: the mean time between failure (MTBF) and the cost of checkpointing. These approaches assume that failures are exponentially distributed in time. Several recent studies [16], [25] have shown that failures in HPC systems are correlated in time and produce periods with a high failure density. We call those periods of time *degraded regime*, as opposed to the *normal regime*. Using the same fault tolerance policies (e.g., the same checkpoint interval) during different regimes is inefficient. This new knowledge brings new opportunities to improve efficiency through dynamic adaptation of the fault tolerance runtime. Unfortunately, current systems are unable to recognize in real time when the system is functioning in degraded mode. Moreover, current runtime systems are incapable of dynamically adapting to changes in the system. Also, it is uncertain how much benefit can we obtain using this failure regime knowledge.

In this paper, we study the failure logs of multiple HPC systems, including Titan, Blue Waters and Tsubame2.5. We confirm that those systems have periods of high failure density, up to three times higher than the standard failure density of those machines. Then, we analyze different types of failures and discover that some are clear markers that could be used to recognize when the system enters one of those degraded regimes. Accordingly, we develop a distributed monitoring system that listens to hardware events, analyzes them, filters them, and forward the important ones to the runtime. We enhance FTI [3], a multilevel checkpointing library, to be able to receive notifications and dynamically adapt during runtime, in order to reduce wasted time. Moreover, we develop an analytical model that shows that for some systems, wasted time can be reduced by over 30% using our monitoring system and the dynamic runtime adaptation. This capacity of self-analysis (i.e., introspection) and dynamic adaptation will become more important as systems become more dense and failures more correlated.

The rest of this paper is organized as follows. Section II studies the failure regimes. Section III explains our monitoring system and presents our dynamic runtime. Section IV introduces our analytical model, Section V discusses related work, and Section VI concludes the paper.

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. This material is also based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Contract DE-AC02- 06CH11357. The work also used the resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE.

II. Failure Regimes in HPC Systems

We analyzed several HPC systems of different generations and characteristics in order to observe the behavior of the failure events. For this purpose, we gathered failure data for Mercury, a previous-generation cluster at the National Center for Supercomputing Applications (NCSA) [1]; for several systems from the Los Alamos National Laboratory (LANL) [27]; for Tsubame 2.5, a supercomputer at the GSIC Center at the Tokyo Institute of Technology in Japan, ranked 5th in 2011 [19]; for Blue Waters [23]; and for Titan from Oak Ridge National Laboratory, currently the fastest supercomputer in the USA [19].

A. System characteristics

The Mercury cluster was a production high-performance computing system at the NCSA used for scientific applications as part of TeraGrid over a five-year period The cluster started with 256 compute nodes, half having large amounts of memory (12 GB). Later, an additional 635 compute nodes were added with faster processors. Each compute node consisted of two Itanium processors running at 1.3 or 1.5 GHz with 4 or 12 GB ECC protected memory.

The cluster was operational from January 2004 until March 2010 when it was decommissioned. During this time, the most frequent failures can be divided in six categories: (1) errors in memory that were uncorrectable by ECC; (2) processor cache errors; (3) hardware-reported error in a device on the SCSI bus; (4) NFS-related error indicating unavailability of the network file system for a machine; (5) PBS (Portable Batch System) daemon failure to communicate; and (6) an unexpected node restart caused by unexpected hardware failure indicated by a node's inability to connect to either the gigabit or management networks after a restart. Details about each failure type can be found in [22].

Table I
System characteristics

System	Timeframe	MTBF (h)	Hardware	Software	Network	Environmental	Other
Blue Waters	2012/12/28-2014/02/01	11.2	47.12%	33.69%	11.84%	3.34%	4.01%
Tsubame	2015/01/01-2015/02/28	10.4	67.24%	12.79%	6.56%	7.66%	5.75%
Mercury	2005/01/01-2009/12/26	16.0	52.38%	30.66%	10.28%	2.66%	4.02%
LANL all	1996/06/01-2005/06/01	23.0	61.58%	23.02%	1.8%	1.55%	12.05%

LANL has collected data for 22 of their supercomputing clusters, for five years. The laboratoy publishes data from system logs and information regarding failures that occurred in the system's lifetime. This data has been intensively analyzed in order to extract data statistics for failure distribution and for root cause analysis [34], [27].

Most of these systems are large clusters of either NUMA (non-uniform memory access) nodes, or two-way and four-way SMP (symmetric multi processing) nodes. In total all systems include 4,750 nodes and 24,101 processors. In general, systems vary widely in size, with the number of nodes ranging from 1 to 1,024 and the number of processors from 4 to 6,152. We analyzed five of these systems individually, as well as investigated the failure characteristics of all systems combined. A failure record for each system contains the time when the failure started, when it was resolved, the system and node affected, the type of workload running on the node, and the root cause.

The Blue Waters system is a Cray XE/XK hybrid machine at the National Center for Supercomputing Applications composed of AMD 6276 "Interlagos" processors and NVIDIA GK110 "Kepler" accelerators all connected by the Cray Gemini torus interconnect. Blue Waters contains over 25,000 computing nodes, reaching a peak performance of 11.6 Petaflops and offering a total system memory of over 1.4 PB. Information about failures is kept in a failure log where Cray system administrators write down the approximate timestamp for each failure and the possible cause. This file has been analyzed and validated by NCSA staff; their observations were published in [13]. Overall, we analyzed failures occurring from September 2013 to February 2014.

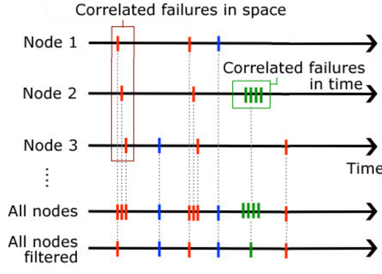
The Tsubame supercomputer was deployed at the GSIC Center in 2006 and has been providing computing and storage resources for research at the Tokyo Institute of Technology in Japan. With 74,358 cores, Tsubame 2.5 can reach a theoretical peak performance of 5.6 petaflops. The machine contains over 12,000 Xeon X5670 6C 2.93 GHz processors connected through a InfiniBand QDR network and a total of close to 75 TB of memory. We analyzed the failures experienced by this system for several months in 2015.

Table II
Systems analysis

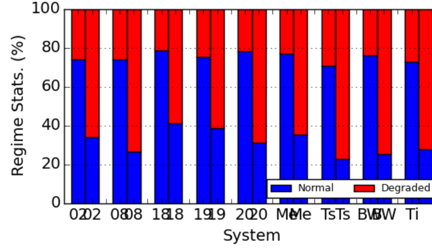
System	LANL02	LANL08	LANL18	LANL19	LANL20	Mercury	Tsubame 2	BlueWater	Titan
Normal reg. px	73.81	74.15	78.36	75.05	78.19	76.69	70.73	76.07	72.52
Normal reg. pf	33.92	26.42	40.84	38.58	31.05	35.10	22.78	25.05	27.77
Normal reg. pf/px	00.46	00.36	00.52	00.51	00.40	00.46	00.32	00.33	00.38
Degraded r. px	26.19	25.85	21.64	24.95	21.81	23.31	29.27	23.93	27.48
Degraded r. pf	66.08	73.58	59.16	61.42	68.95	64.90	77.22	74.95	72.23
Degraded r. pf/px	02.52	02.85	02.73	02.46	03.16	02.78	02.64	03.13	02.63

Titan is a 18,688 nodes machine deployed at ORNL since early 2013. Each node has one AMD Opteron 6274 CPU (with 32 GB of DDR3 memory) and one NVIDIA K20X GPU (with 6 GB of GDDR5 memory). Each blade has two high-speed interconnect Gemini routers, each shared by two nodes. NVIDIA K20X GPU has a total of 14 streaming multiprocessors, each streaming multiprocessor has 192 CUDA cores. Overall, each GPU has peak performance of over 1.30 Tflops (double precision). Titan supercomputer is able to perform at a theoretical peak of 27 petaFLOPS [6]. We analyzed system failure for the Titan supercomputer for Jun 2013 to Feb 2015. We note that it is very complex to breakdown the failure root cause for some of the failures without introducing inaccuracy. Therefore, we omit the particular breakdown of failures in Table I without affecting the conclusion of this study. Nevertheless, overall we observed the failure regimes and those results are included in Table II, more details can be found here [18], [14], [17], [12].

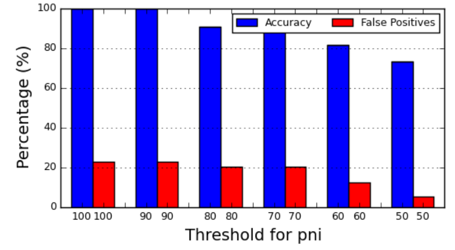
For all systems, we used the failure categorization given



(a) Failures correlation



(b) Regime characteristics



(c) Accurate regime detections VS false positives

Figure 1. Analysis of failures in petascale systems

by the system administrators in each center, and we group each failure in one of the following categories: hardware, software, network, environment, and unknown (i.e., failures cause could not be determined). Table I shows the timeframe analyzed for each system and their failure characteristics.

B. Regimes

Several studies analyzing HPC systems have concluded that the failures follow an exponential or Weibull distribution; thus all the models used so far assume that successive intervals between failures are independent. We created a simple, yet effective algorithm to test this hypothesis.

The algorithm follows the following four steps:

- We extract the standard MTBF by dividing the time length for each system to the number of failures that occurred in the given time frame. For this purpose, a filtering mechanism is applied first in order to isolate individual failures from their cascading occurrences
- We divide the time frame of each system in segments of their corresponding MTBF length. If failures are indeed independent to each other and uniformily distributed, each segment will show at most one failure.
- We count the number of failures in each segment, after which we aggregate this information for all segments that present the same number of failures. We use x_i to denote the number of segments with i failures with i=0,1,or any number > 1. The segments with 0 or 1 failure define the normal regime, while those with more than one failure indicate a degraded regime.
- We define f_i as $f_i = x_i * i$, which gives the total number of failures for each type of segment. This information will allow us to compute the percentage of failures that occur in degraded regimes.

The first step of our algorithm assumes a filtering method that will correctly match multiple failures indicated in failure logs to one individual failure. We processed each message type by choosing different thresholds that allowed us to isolate unique failures from redundant ones, in both space and time. For example, some memory failures can appear multiple times during the execution of an application because of repeated access to the corrupted component when a memory module failed. At the same time, transient correctable errors in these components can generate isolated single messages but not affect an application, so they are of no interest for our study. Figure 1(a) presents scenarios where failures need to be filtered across different nodes or in time. We use the method from [20] to remove redundant failures of the same type both in space and time for all our systems, before applying our algorithm. For the systems that offer failure logs written by system administrator, filtering in time is not necessary since notifications were gathered into a single row.

C. Results

Table II presents the results obtained when applying our algorithm on the systems presented in Section II-A. We define the percentage of failures that occur in the normal/degraded regimes as $pf = \frac{f_i * 100}{\sum f_i}$ and the percentage of segments that represent the normal/degraded regimes as $px = \frac{x_i * 100}{\sum x_i}$. The ratio between pf and px defines the multiplier to the standard MTBF that gives the MTBF of the current regime. For systems whose failures follow an exponential distribution, the normal regime pf and px should both be close to 100%.

As seen in the table, all the analyzed systems present degraded regimes. For the Blue Waters system, the degraded regime is present in 24% of the system's lifespan, during which time 75% of the failures occur. In other words during degraded regimes, Blue Waters experiences an MTBF 3 times shorter than the standard MTBF. Similarly, all systems show a degraded regime that takes 20-30% of each system's timeframe and that presents between 60 to 78% of all failures affecting the system. We can no longer assume exponential distribution of failures in HPC systems, and one can expect degraded regimes with 2.5 to over 3 times more failures.

Moreover, during the normal regimes the frequency of failures is much lower than expected with only 25%-40% failures occurring in more than three quarters of the entire lifespan of a system. This means that for most of the system's lifetime the standard MTBF represents only a fraction of the

actual MTBF. For example, for Blue Waters, the MTBF for the normal regime is around three times higher than what is considered for computing the optimal checkpoint interval.

Figure 1(b) presents a concise view of the regimes identified in the previous table. The first column for each system presents the percentage of time spent in normal and degraded regimes while the second presents the percentage of failures occurring in each regime. It is quite visible that all systems present almost 75% of the failures in around 25% of the time. The figure also makes it easier to compare the results between systems, showing that supercomputers today (represented by Tsubame and Blue Waters) have, in general, larger percentages of failures in shorter time intervals. We can expect this trend to increase in future systems.

We analyzed the properties of failures occurring inside degraded regimes. Around two thirds of the regimes have a time span of more than 2 standard MTBFs during which more than 30 failures occur in each. Depending on the system and on each regime, the failures can be fitted by the Weibull and Exponential distributions with different parameters. Due to space limitation we cannot show our detailed analysis. However, our results show that the standard formula for computing the checkpoint interval can be used inside degraded regimes.

D. Regime detection

A degraded regime is defined by an increase in failure frequency. Thus, a regime change from normal to degraded each time the system encounters a failute would hold a false negative rate of 0%. However, since not all failure indicate the beginning of a degraded regime, this method will give false positives. The second column for each system from Figure 1(b) shows the percentage of failures occurring in each regime. Thus, less than 25% of total failures trigger an unnecessary regime change, keping the false positive rate around 50%. However, the regime is switched back to normal after a time frame equal to half of the standard MTBF.

In order to decrease the rate of false positives for this default detection mechanism, we look into more detail at the types of failures that usually occur during degraded or normal regimes. We use the following steps in order to find patterns of failures that can indicate degraded regimes:

- Instead of listing the number of failure per segment we keep a list of failure types that occur on each segment.
- For each failure type i we count the segments during a normal regime where i occurs alone in n_i.
- For each failure type i we count the segments during a degraded regime where i occurs first in d_i.
- We compute the percentage of time a failure type occurs in normal regimes as $\frac{n_i*100}{count_i}$. Since for detection purposes, we only care about the ratio of the number of times the failure occurs in normal regimes to how many times it occurs as a first failure in a degraded regime, we also compute $pn_i = \frac{n_i*100}{n_i+d_i}$. In degraded regime we can ignore the occurrences of this failure type.

We identified several failure types for all systems that can improve detection. Table III shows a couple of failure types examples and their pn_i percentages for two of the analyzed systems. When only considering the failure types that always occur during normal regime (pn_i=100%) we still detect correctly all degraded regimes, but now we decrease the false positive rate to 30-35% depending on the system. This corresponds to between 10-15% of all failures triggering a regime change unnecessarily.

Tsubame 2.5		LANL systems	
Failure type	pn_i	Failure type	pn_i
SysBrd	100%	Kernel	100%
GPU	55 %	Memory	61%
Switch	33 %	Fibre	100%
OtherSW	100.0 %	OS	49%
Disk	66 %	Disk	75%

Table III
FAILURE TYPES OCCURRING IN NORMAL REGIME

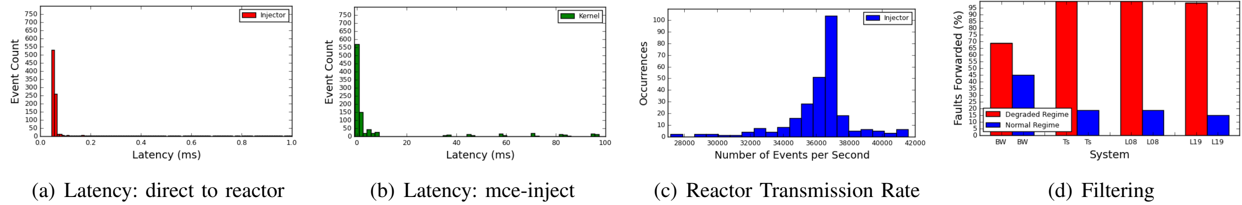
When also including failures that most of the time occur during normal regimes ($pn_i > X$ for X in {75%-100%}) we can play with the ratio between the number of false positives and the accuracy of the degraded regime detection. Figure 1(c) presents the trade-off between these two metrics for the LANL system 20 for different thresholds for pn_i.

III. REAL-TIME MONITORING AND DYNAMIC CHECKPOINTING

A dynamic checkpointing scheme can take advantage of those different failure regimes in an HPC machine, if several components are provided. First, a monitoring system must exist, to discover events from as many parts of the system as possible. Second, a notification system needs to provide this information to the checkpointing runtime. Then, that runtime must be aware that this information is available and must have policies in place to react to those notifications.

While current machines do have a fault monitoring, notification, and management system, those components are considered part of the administration software. Their purpose is to notify system management staff and, to a lesser extent, the job scheduler, so that failed components can be repaired, exchanged, or reconfigured. Moreover, current machine configurations tend to destroy any job encountering a failure, without the application being able to act on it.

Nevertheless, we expect future machines to provide more details on the current status of the system, and we also expect that runtime systems will be capable of using that information to react to failures. As an example, the Argo project [5] is building an operating system and runtime software stack for exascale systems and includes a fault management component linked to a publish/subscribe infrastructure available to application runtime systems. In the meantime, it might be possible to collocate with a user job this kind of service. To this extent, we first describe here a prototype event



(a) Latency: direct to reactor (b) Latency: mce-inject (c) Reactor Transmission Rate (d) Filtering

Figure 2. Validation results: (a) and (b) show latency distributions using different injection paths (respectively directly to the reactor and by the mce-inject tool), for 1000 events injected. (c) shows the distribution of the number of events forwarded in a second by the reactor. (d) presents the ratio of failures that were forwarded by the reactor depending on the probability of its type happening in each regime.

monitoring, notification, and filtering system, to validate that this approach is reasonable. In particular, we want to test notification latency, overhead of the on-the-fly event analysis, and platform-aware filtering strategies. Then, we describe a dynamic checkpointing scheme that can use this monitoring facilities.

A. *Monitoring Machine Events*

Our monitoring implementation is split into two major components: a monitor and a reactor. The monitor is responsible for polling various sources of information inside a standard Linux node. In particular, we gather events from the *Machine Check Architecture* [10] of recent processors. The reactor receives those events from the monitor and acts on them based on predetermined rules. We also implemented an injector, to test the infrastructure by injecting fake events into the system. These components were prototyped in Python and use ZeroMQ to communicate with each other. We detail here those components.

Monitor: our current monitor scans the system for events originating at several levels. The Machine Check Architecture is a feature of recent processors generating interrupts when a correctable or uncorrectable error occurs in the system. This interrupt is handled by the Linux kernel; if it doesn't create a system panic right away, it is forwarded to a user-level daemon that decodes the information and notifies users. Our monitor polls the corresponding log files for this information, encodes it for consumption by the reactor, and forwards it. Additionally, our monitor retrieves at regular intervals information from the temperature sensors in the system. This information includes the location of the sensor (e.g., fan, CPU), the current reading, and the hardware limits (critical levels). Finally the monitor gather statistics on network interfaces and disks to discover errors or intensive use.

Reactor: our reactor implementation is straightforward. It listens for events, analyzes them, and *reacts* to them by either filtering them or forwarding them to an application runtime. The analysis subsystem is implemented by using two major ideas: an encoding of all events as set of values **(component, event type, data)**, and user provided **platform information**. The main goal of this reactor is to to attach the maximum amount of information to important events before forwarding them while minimizing the noise.

Event Encoding: most of the encoding of our event is done by the monitor. In particular, the component and event type identification is directly related to the source of the event and thus is better applied the first time the event is detected. That is to say, if an event is received several times in a short period of time, only one notification is raised to limit system noise. Nevertheless, we could envision a trend analysis inside the reactor identifying a slow but steady increase in temperature, for example, and act on it by rewriting the encoding of some events.

Platform information: the user can provide to our reactor information about the events occurring on the machine. This information would typically originates from the kind of offline analysis presented in the previous section. For our experiment purposes, the platform information is used to map events to information on their occurrence during normal and degraded regimes.

B. *Validation*

For such infrastructure to work, several key features need to be validated. In particular, we want to verify that major events are detected rapidly, that a cluster of events will not overload the system, and that our notification/filtering system provides enough information to a runtime that it can take useful measures. All these experiments were made on a single Linux system with an 4 cores Intel i7–4600 CPU and 8 GB of RAM, kernel version 4.2.2.

First, we measure the latency of a event traveling through our system. We measure this latency for two different paths: an event being injected directly into the reactor and a machine check error being injected into the kernel, caught by the monitor and forwarded to the reactor. In both cases, the reactor simply annotates the event with a timestamp and prints it out. Figures 2(a) and 2(b) give the resulting distribution of latency for 1,000 events injected. As expected, an event traveling through the kernel and the monitor takes significantly more time than sending the event directly to the reactor. Nevertheless, those latencies are all largely below one second, a very good latency in the context of checkpointing runtimes with a resolution in the order of minutes.

Second, we measure the transmission rate of our system: how many events can be caught and analyzed by our system per second. While it is unlikely that a single node

experiences a massive amount of failures in a single second, this experiment can provide some insight for an eventual distribution on this infrastructure across nodes. Indeed, if multiple components in an HPC system fail at the same time, the node gathering those events for analysis must react efficiently to the sudden flow of events. Figure 2(c) presents the distribution of the number of events our reactor can analyze in a second, during the continuous injection of 100 million events per 10 different processes running concurrently. As we can see, our prototype reactor can receive and analyze an average of 36,000 events each second. We can expect that no scenario can produce that many errors in a single node in a single second. From an analysis of the fault trace archive traces, we can also say that even on modern systems, no type of failure triggers that many events at the same time across the system. Moreover, we expect each source to filter its own events for better scalability.

One strategy for such filtering is to ignore events depending on their types and the probability of this type of events to happen in degraded regimes, based on platform information. To evaluate such approach, we create traces matching the systems we analyzed before. Those traces contain a fixed number of segments, each either a normal regime or a degraded regime and in each segment, we randomly generate failures, using the px and pf from Table II. The type of each failure is then randomly chosen to respect the ratio of Table I and the ratio of each event type happening in each regime. The reactor is provided by the platform information the probability that each type of event occurs in a degraded regime. Additionally, each segment of the trace starts by a *precursor event*: an event carrying a random number, modifying the platform information only for the events occurring during the same segment. Finally, we inject those traces in our monitoring system. The reactor is configured to filter all event types that happen more than 60% of the time in normal regime for the system simulated. In this setup, precursor events simulate live reports for the monitor that the system is behaving in a certain way (i.e a normal precursor increase the change that an event is happening during a normal regime, and vice versa).

We then analyse the injected trace and the filtering applied by the reactor to measure how many failures happening in normal regime and in degraded regime have been forwarded to the runtime. Figure 2(d) reports the results, which across most system show a high rate of degraded regime events forwarded and reduced amount of events in normal regimes.

C. Dynamic Checkpointing

We have seen that HPC systems can detect regime changes by using a lightweight notification system. Once that notification is transmitted up through the software stack, the runtime system catches it and adapts to the new regime. In the case of fail-stop failures, adapting to a degraded regime means changing the checkpoint interval. In most HPC applications, however, the checkpoint interval is almost hard-coded in a certain number of outer-loop iterations. Using such an approach, one cannot change the checkpoint interval during runtime.

Algorithm 1 Dynamic Checkpoint Interval

```
procedure FTI_SNAPSHOT
    addLastIterationLengthToList(IL)
    if updateGailIter == currentIter then
        GAIL = compute Global Average Iteration Length
        IterCkptInterval = wallClockCkptInterval/GAIL
        if updateRoof > expDecay*2 then
            expDecay = expDecay*2
        end if
        updateGailIter = currentIter + expDecay
    end if
    if nextCkptIter == currentIter then
        FTI_Checkpoint
        nextCkptIter = currentIter + IterCkptInterval
    else
        received = checkForNewNotifications(noti)
        if received then
            endRegimeIter, IterCkptInterval = decodeNotification(noti)
        end if
    end if
    if endRegimeIter == currentIter then
        IterCkptInterval = wallClockCkptInterval/GAIL
        endRegimeIter = -1
    end if
currentIter = currentIter+1
end procedure
```

FTI circumvents this problem by proposing an interface where the checkpoint function (i.e., FTI_Snapshot) is called at every iteration and a check is done to see whether checkpointing is necessary or not. The user provides the checkpoint interval in wall-clock time (i.e., minutes) in a configuration file and FTI transforms it into numbers of iterations. To do so, FTI constantly measures the time spent between two consecutive calls to FTI_Snapshot (i.e. one iteration) and computes an average, which is later averaged with all the processes of the application and they agreed upon a *global average iteration length* (GAIL). Using GAIL, FTI can guarantee that a number of minutes will be translated into the same number of iteration for all the processes in the application.

Implementing a dynamic checkpointing runtime on top of this structure is relatively simple. The OS will transmit a notification and FTI will decode it, match it with an existing rule and enforce the new checkpoint interval. If a new notification arrives before the end of the expiration time of the just enforced rule, FTI will enforce the parameters of the new notification and reset the expiration time. Algorithm 1 presents the whole algorithm for dynamic checkpoint interval adaptation. The source code of the dynamic version of FTI is available online [24].

IV. Analytical Modeling of Waste Time for HPC Applications

In the previous sections we explained how a combination of online monitoring and dynamic checkpointing could work together to adapt to different failure regimes. In this section we develop a model to predict how much wasted time reduction is achievable for extreme scale systems with different regime characterizations.

Table IV
PARAMETERS USED IN THE MODEL.

Notation	Meaning
T_{waste}^{total}	Total wasted time (ckpt.+restart+re-exec.)
Ex	Total (failure-free) computation time
R	Number of failure regimes
M	Overall MTBF of the system
Ck_i	Time spent in checkpointing in regime i
Rt_i	Time spent in restarting in regime i
Rx_i	Time spent in re-executing in regime i
px_i	Fraction of time spent in regime i
M_i	MTBF in regime i
α_i	Checkpointing interval in regime i
β	Time to write one checkpoint
γ	Time to restart
ϵ	Average fraction of lost work per failure.

A. Analytical Model

The total wasted time is the sum of the time spent in checkpoint, restart overhead and re-execution across all the regions. This can be expressed as following:

$$T_{waste}^{total} = \sum_{i=1}^{R} (Ck_i + Rt_i + Rx_i) \tag{1}$$

Next, we estimate the time spent in checkpoint, restart overhead and re-execution for a given region. The number of checkpoints in i^{th} region can be determined by dividing the time spent in the ith region by the checkpointing interval in the ith region (α_i) minus one. Therefore, the time spent in the checkpointing phase can be expressed as:

$$Ck_i = (\frac{Ex \times px_i}{\alpha_i}) \times \beta \tag{2}$$

To estimate the time spent in restart phase and re-execute phase, we need to calculate the number of failures in the ith region. Let's call the combination of one compute interval (α_i) and checkpointing (β) as one pair. One will need to finish $\frac{Ex \times px_i}{\alpha_i}$ such pairs to complete the required computation in the ith region. Let's denote these number of pairs as by P_i.

The number of failures in the ith region can be expressed as the difference between the total number of trials needed to complete P_i pairs without encountering a failure and the number of times the pairs completed successfully (P_i). Recall that each pair is a combination of compute and checkpointing activity, ($\alpha_i + \beta$). Therefore, the number of failures in the ith region can be expressed as:

$$f_i = \frac{P_i}{1 - Pr(t < (\alpha_i + \beta))} - P_i \tag{3}$$

For an exponential distribution, the probability of failure before time t is given by $Pr(X \leq t) = 1 - \mathrm{e}^{-\frac{t}{M}}$. Therefore, the above expression can be rewritten as:

$$f_i = P_i(\mathrm{e}^{\frac{\alpha_i + \beta}{M_i}} - 1) \tag{4}$$

Where P_i is equal to $\frac{Ex \times px_i}{\alpha_i}$. Therefore, the time spent in restart phase can be expressed as:

$$Rt_i = \frac{Ex \times px_i}{\alpha_i}(\mathrm{e}^{\frac{\alpha_i + \beta}{M_i}} - 1) \times \gamma \tag{5}$$

The time spent in the re-execute phase is the number of failures in the ith region (as calculated earlier) multiplied by the average lost work per computation and checkpointing phase. If the average fraction of lost work per failure is denoted by ϵ, then the lost work per computation and checkpointing phase can be estimated as $\epsilon(\alpha_i + \beta)$. Therefore, the time spent in the re-execute phase can be expressed as:

$$Rt_i = \frac{Ex \times px_i}{\alpha_i}(\mathrm{e}^{\frac{\alpha_i + \beta}{M_i}} - 1) \times (\epsilon(\alpha_i + \beta)) \tag{6}$$

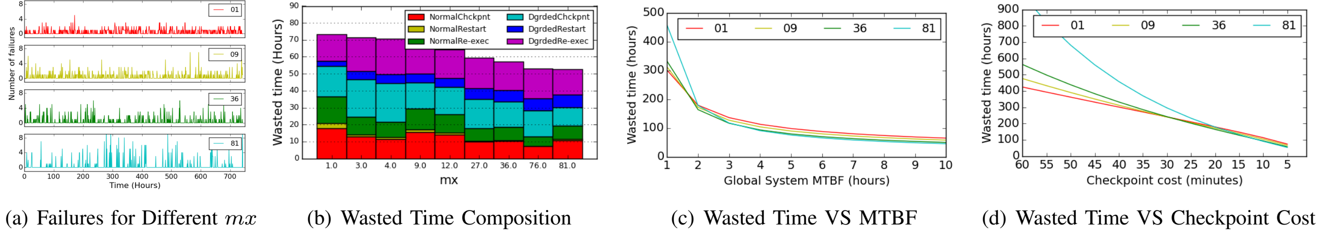
Altogether, the total wasted time can be expressed as:

$$\begin{aligned} T_{waste}^{total} = \sum_{i=1}^{R} ((\frac{Ex \times px_i}{\alpha_i}) \times \beta \\ + \frac{Ex \times px_i}{\alpha_i}(\mathrm{e}^{\frac{\alpha_i + \beta}{M_i}} - 1) \times (\epsilon(\alpha_i + \beta) + \gamma)) \end{aligned} \tag{7}$$

We note that ϵ, average fraction of lost work per failure, is dependent on the inter-arrival time of failure events as reported by a previous study [16]. If the failure inter-arrival times follow an exponential distribution, then ϵ can be approximated as 0.50. If the failure inter-arrival times follow an Weibull distribution, then ϵ can be approximated as 0.35. However, it does vary depending on the same parameter. The failure regime observation is better aligned with the failure inter-arrival times following an Weibull distribution because failure regimes essentially indicate that there is temporal locality in failures. Previous work has shown that temporal locality in failures in better modeled by the Weibull distribution [16]. Furthermore, Young's formula for the optimal checkpointing interval is $\sqrt{2M_i\beta}$ [32]. We can substitute α_i by this in the previous equation to simplify the expression for total wasted time.

B. Wasted Time Reduction Projections

We next want to evaluate how much reduction of wasted time we can achieve by exploiting the idea of multiple failure regimes and a dynamic checkpoint interval. We use the analytical model presented in Section IV-A to compute the wasted time for each regime. Following the analysis in Section II, we limit the number of regimes R to 2, the *normal regime* and the *degraded regime*. We create a battery of 9 systems with



(a) Failures for Different mx (b) Wasted Time Composition (c) Wasted Time VS MTBF (d) Wasted Time VS Checkpoint Cost

Figure 3. Analysis of wasted time in exascale systems with different regime characteristics

different failure regime characteristics. We characterize the systems with a new parameter $mx = \frac{MTBFnormalregime}{MTBFdegradedregime}$. When a system exposes highly temporally clustered failures, degraded regimes tend to have short MTBF and normal regimes long MTBF, giving as a result a high mx value. In contrast, a system with similar regimes will have a $mx = 1$.

Figure 3(a) shows the failure frequency for a system with four different mx values. In all four cases the system has an overall MTBF of 8 hours but different regime characteristics. We notice that the first scenario has a uniform distribution of failures ($i.e., mx = 1$), with rarely more than two failures hitting the system in the same hour. For the same reason, we see few areas of no failures in this system. As we move to higher mx values, we notice that the system shows higher failure bursts and also longer periods of time without failures. According to our study of HPC systems, $mx = 09$ corresponds to the Tsubame 2.5 supercomputer, where about 80% of the failures occur in only about 30% of the time. We observe that such a system could benefit from two different checkpoint intervals. We also study systems with a higher failure density on the degraded regime, since future systems, with more shared components, could show a higher degree of correlated failures.

We use our model to compute the wasted time for each of the phases and regimes of systems with largely different regime characteristics. We use the same overall MTBF of 8 hours, and a checkpoint cost of 5 minutes, and restart cost of 5 minutes for all cases. The only variation among them is the regime characteristics (i.e., mx). In Figure 3(b) we can see that is mx increases, the wasted time decreases. For a system with $mx = 81$ the wasted time can be reduced by 30% in comparison with the same system but with $mx = 1$. We also notice that the wasted time of degraded regime is larger than the wasted time in normal regime. This is consistent with the fact that most failures happen in degraded regime.

The MTBF of exascale systems is uncertain. Thus, we study how the overall MTBF of the system impacts the wasted time for four different regime characteristics. Using our model, we compute the wasted time for an exascale system with an MTBF varying between one to ten hours. We assume a fixed checkpoint cost of 5 minutes. In Figure 3(c) we plot the wasted time for the same system but with the four different regime characteristics presented in Figure 3(a). We observe that the wasted time decreases as the MTBF increases, which was expected. However, we also observe that systems with high mx perform badly for short MTBF. Then, as we increase the MTBF this reverts, to the point that a system with high mx spends 30% less wasted time than a system with a low mx. To understand this behaviour, we examined the composition of the wasted time for each one case and we discovered that systems with low MTBF and high mx will spend an extremely large amount of time in degraded regime. This is because the MTBF in degraded regime is so low that is comparable with the checkpoint cost, making almost impossible to progress in the execution.

Then, in Figure 3(d) we study how the checkpoint cost impacts the wasted time for systems with different regime characteristics. We assume a fixed MTBF of 8 hours and a checkpointing cost between 1 hour and 5 minutes. This is assuming a transition from file system checkpointing to burst buffers and then checkpointing in NVM. Again, we observe that for systems with costly checkpoints and high mx the overhead is extremely high. After detailed analysis, we understood that in those cases, the checkpoint interval in degraded regime is very close to the checkpoint cost, making, again, execution progress substantially difficult. However, as the checkpoint cost decreases, the trend reverts and systems with high mx show up to 30% reduction in wasted time.

Overall, we understood that for systems where the MTBF is not substantially longer than the checkpoint cost, the situation worsens on degraded regime, making execution progress challenging. This leads to an extremely high wasted time and overhead. However, systems where the MTBF is significantly larger than the checkpoint cost, show clear benefits (over 30% reduction in wasted time)of detecting failure regimes and adapting to them dynamically.

C. Discussion

We have found that multi-petaflop machines used in production today expose clear periods of time with high failure density, sometimes up to three times higher than the average rest of the time. There are several reasons that could explain such phenomena. For instance, after a machine is upgraded with new hardware, the early *infant mortality* failures generate a period of time with higher failure density. Another explanation for the presence of such regimes is when a shared component (e.g., Parallel File System)

fails intermittently producing multiple job crashes before shutting down completely. Indeed, from the moment a failure occurs until the system administrators finds the root cause of the failure, the same component could generate multiple failures before being fixed. Furthermore, some failures can be detected and understood quickly, but the measures taken to address the problem could take significant time to produce the desired effect. For instance, a malfunctioning cooling system could be detected and fixed in a timely fashion, but the temperature of the racks could take a certain time to decrease, during which multiple failures could happen. These are just some examples of the reasons that could explain the presence of such degraded regimes. Unfortunately, we don't have enough data about the systems we studied, to explain all the causes of the degraded regimes in those systems, although some patterns were found.

Indeed, we analyzed the types of failures that happen more frequently in each regime in order to detect regime changes. Given that most failures occur in degraded regime we try to discover which ones have a high probability of occurring in normal regime, so that they are filtered by our monitoring system and they do not trigger any action in the resilience runtime. We proved that our monitoring system could efficiently forward enough information that will allow the runtime to dynamically adapt to the regime changes. However, our monitoring system is just a proof of concept and could be greatly improved using more sophisticated techniques coming from the system monitoring and data analytics literature. The goal of this research is not to study how to build a real-time monitoring system, but rather to show what could be achieved if we had one, in the context of fault tolerance for large scale HPC system.

We note that regime detection and failure prediction are two different problems. A monitoring system trying to detect regimes has constant uncertainty about when and where the next failure will occur, while a failure predictor is constantly trying to reduce that uncertainty to zero. Failure predictors try to foresee events in the future, in regime detection we analyzed events that already happened to try to understand the current status of the machine. This study show insights in what could be more beneficial in the future. That is, if we know that a certain type of failures happens in bursts and another type of failures occur following a uniform distribution, one should put more effort on limiting the later type of failures, as the number of failures occurring during burst is irrelevant as far as the burst does not last too long.

V. Related Work

There are multiple studies that look at the statistics of events and failures generated by several systems, including the root cause of failures, the mean time between failures, the mean time to repair and the distribution of failures.

Table V presents an overview of these studies, the analyzed systems and their observed failure distribution. All studies

Table V
FAILURE DISTRIBUTION FOR SEVERAL SYSTEMS

System	Failure distribution	Citation
O2K	Weibull distributions	Lu et al [25], 2013
Titan	Weibull distribution	Tiwari et al [16], 2014
Platium	Exponential distribution	Lu et al [25], 2013
Blue Gene/L	Weibull distribution	Taerat et al [15], 2009
Blue Gene/P	Weibull distribution	Harper et al [21], 2009
20 systems at LANL	Weibull distribution with decreasing hazard rate	Schroeder et al [27], 2010

show Weibull, exponential or lognormal distributions for the failure inter-arrival time. In [28], the authors investigate how the time since the last failure influences the expected time until the next failure. In general, the system they analyze are well fit by a Weibull distribution, in most cases with a shape parameter of less than 1, indicating that the hazard rate function is decreasing.

The study from [30] uses the failure trace obtained from prominent HPC platforms to study and compare different distributions, Exponential, Weibull and Lognormal for fitting the failures that affect applications running on k nodes. Their results indicate that Weibull distribution results in the better reliability model in most of the cases for the given data. In our experience, an upgrade of the filesystem will be followed by numerous failures of different types.

System monitoring: In general, HPC system vendors integrate management systems (e.g., Cray's Node Health Checker [4] or IBM's Cluster Systems Management [8]) that aim at ensuring job completion and providing system administrator with a way to check the sanity of the machine. Cray's node health checker is automatically invoked by the scheduler upon the termination of an application.

One method used extensively in the past was to measure each node's behavior and compare it to all other nodes executing similar workloads. An event is categorized as a failure in case of a significant deviation [33], [29]. Another similar type of method models the components and their interactions and then monitors the model. Most examples are using pattern recognition [26], [31] algorithms to model the system. Others include context free grammars [9] and mathematical equations [2]. More specific methods focus on the software stack of a HPC system.

For our purposes, these methods are orthogonal since our focus is not failure detection, but rather regime detection. Since most of the failures are part of a degraded regime, we only need to identify patterns in failures the occur in normal regimes in order to avoid unnecessary regime changes. While these methods have the potential of being adapted to detect regimes, it is beyond the scope of this paper.

VI. Conclusions

In this paper we study several HPC systems and discover the presence of periods with up to three times higher failure density than the rest of the time. We analyze failure patterns and we identify regime changes based on the failures types. We design and test a monitoring system that listen, analyzes, filters, and forwards fault events to the reliability runtime to notify about regime changes. We implement a dynamic

checkpointing scheme capable to adapt on-the-fly to regime changes. Then, we build an analytical model able to predict waste reduction for future supercomputers. We demonstrate that some systems can reduce over 30% of wasted time using this approach. To the best of our knowledge, this study of failure regimes in HPC systems and the benefits that can be obtained from it, is the first of its kind. As future work, we would like to improve our regime detection mechanisms using more sophisticated analytics.

REFERENCES

[1] National Center for Supercomputing Applications at the University of Illinois. *www.ncsa.illinois.edu*.

[2] Artur Andrzejak and Luis Moura Silva. Deterministic models of software aging and optimal rejuvenation schedules. In *Integrated Network Management*, 2007.

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

[4] J. Becklehimer, C. Willis, J. Lothian, D. Maxwell, and D. Vasil. Real time health monitoring of the cray xt3/xt4 using the simple event correlator (sec)., 2011.

[5] Pete Beckman, Marc Snir, et al. Argo: An exascale operating system and runtime research project. http://www.argo-osr.org/.

[6] Buddy Bland. Titan - early experience with the titan system at oak ridge national laboratory. *SC Companion:*, 2012.

[7] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 2005.

[8] V. Castelli, R.E. Harper, P. Heidelberger, S.W. Hunter, K.S. Trivedi, K. Vaidyanathan, and W.P. Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 2001.

[9] Mike Y. Chen, Anthony Accardi, Emre Kıcıman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[10] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual.

[11] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 2006.

[12] Gupta et al. Understanding and exploiting spatial properties of system failures on extreme-scale hpc systems. In *DSN*, 2015.

[13] Martino et al. Lessons learned from the analysis of system failures at petascale: The case of Blue Waters. In *DSN*, 2014.

[14] Nie et al. A large-scale study of soft-errors on gpus in the field. In *HPCA*, 2016.

[15] Taerat et al. Blue gene/l log analysis and time to interrupt estimation. In *ARES*, 2009.

[16] Tiwari et al. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *DSN*, 2014.

[17] Tiwari et al. Reliability lessons learned from gpu experience with the titan supercomputer at oak ridge leadership computing facility. In *SC*, 2015.

[18] Tiwari et al. Understanding gpu errors on large-scale hpc systems and the implications for system design and operation. In *HPCA*, 2015.

[19] D.G. Feitelson. The supercomputer industry in light of the Top500 data. *Computing in Science Engineering*, 2005.

[20] S. Fu and C. Xu. Quantifying temporal and spatial fault event correlation for proactive failure management. *SRDS*, 2007.

[21] Thomas J. Hacker, Fabian Romero, and Christopher D. Carothers. An analysis of clustered failures on large supercomputing systems. In *JPDC*, 2009.

[22] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. Modeling and tolerating heterogeneous failures in large parallel systems. In *SC*, 2011.

[23] William Kramer. Introduction to the Blue Waters Project. *National Center for Supercomputing Applications*, 2014.

[24] Leonardo Bautista-Gomez. Dynamic version of FTI. https://github.com/leobago/fti/tree/dynamic, 2015.

[25] Charng-Da Lu. Failure data analysis of hpc systems. *Technical Report CoRR abs/1302.4779*, 2013.

[26] Felix Salfner. Modeling event-driven time series with generalized hidden semi-markov models. *Technical Report 208, Department of Computer Science, Humboldt University*, 2006.

[27] B. Schroeder and G.A. Gibson. A large-scale study of failures in high-performance computing systems. *TDSC*, 2010.

[28] Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series 78:012022*, 2007.

[29] Jon Stearley and Adam J. Oliner. Bad words: Finding faults in spirit's syslogs. *CCGRID*, 2008.

[30] T. Thanakornworakij, R. Nassar, C. B. Leangsuksun, and M. Paun. Reliability model of a system of k nodes with simultaneous failures for high-performance computing applications. *IJHPCA*, 2013.

[31] Kenji Yamanishi. Dynamic syslog mining for network failure monitoring. In *KDD*, 2005.

[32] John W Young. A first order approximation to the optimum checkpoint interval. *CACM*, 1974.

[33] Z. Zheng, Y. Li, and Z. Lan. Anomaly Localization in Large-Scale Clusters. *Cluster*, 2007.

[34] Z. Zheng and L. Yu. Co-analysis of RAS Log and Job Log on Blue Gene/P. *IPDPS*, 2011.