Monitoring Strategies for Scalable Dynamic Checkpointing

Swann Perarnau
Argonne National Laboratory

Leonardo Bautista-Gomez
Barcelona Supercomputing Center (BSC)

***Abstract*—Resilience is an important challenge for extreme-scale supercomputers. Failures in current 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. The detection of those periods is important in order to adjust the system to new conditions. In this paper we present a monitoring system that listens to hardware events across computing nodes and forwards important events to the fault tolerance runtime so it can react to those regime changes. Our evaluation at scale shows several aspects of this dynamic checkpointing scheme, critical to understanding its applicability on production systems, as well as to identifying possible avenues for future improvements. In particular, we evaluate the ability of our system to monitor as many types of events as possible, measure their importance, and forward them to the resilience runtime.**

***Index Terms*—Supercomputers, Fault Tolerance, Resilience, Introspective Systems, Failures, High-Performance Computing.**

I. Introduction

Today, most science domains use supercomputers as important research tools to accelerate research. These supercomputers have been increasing in size for the past two decades and they are expected to keep improving in performance over the coming years. This evolution of supercomputers is also imposing new challenges. Resilience is one of these challenges that must be addressed for extreme scale computing. Indeed, as the number of components increases, the frequency of failures in modern supercomputers keeps increasing, bringing down the mean time between failures (MTBF) [1], [2]. In addition, power constraints are leading the industry to reduce the voltage at which these computing devices operate, increasing the probability of voltage disturbances and hence generating more soft errors.

In High-Performance Computing (HPC) the most common way to cope with failures is checkpoint/restart (CR). CR consists of saving the important data of an application into reliable storage, so that the application can restart from that state after a failure occurs. The state saved by the application is usually large enough to force the application to pause during the checkpoint and resume only after all the data has been successfully moved to disk. This process involves a certain overhead, which is why HPC applications do not checkpoint frequently. In fact, CR is an optimization problem, and it has been studied in numerous research works [3], [4]. It depends on two parameters: the MTBF and the checkpointing overhead. These formulas are good approximations based on the assumption that failures follow an exponential distribution. However, recent studies [5], [6] have shown that failures in supercomputers are correlated in time and generate time periods of higher failure activity. We call those periods of time the *degraded regime*, as opposed to the *normal regime*.

Although current systems are unable to recognize in real time when the system is functioning in degraded mode, we recently developed a monitoring system and the accompanying dynamic checkpointing runtime to take advantage of this information [7]. Our system design is straightforward: daemons running on all compute nodes monitor as many types of events as possible, use operational knowledge to determine whether those events are the sign of degraded regimes, and forward the critical events to the checkpointing runtime. The runtime can then adapt its checkpointing interval accordingly.

Such a scheme relies on the ability of the system to detect when the machine enters a degraded regime. Such ability depends on several critical functions. First, the system must be able to monitor as many relevant sources of events (i.e. indicators of regime change) as possible. Second, critical events coming from all those sources must be accurately identified. Third, the dynamic checkpointing runtime must be notified as quickly as possible.

In this paper, we study the scaling and performance of this monitoring and dynamic checkpointing scheme. First, since the dynamic checkpointing runtime uses a centralized decision process to adapt its checkpointing interval, we focus our scalability study of the event monitoring scheme on the impact of increasing the amount of nodes monitoring and reporting events to a single node for classification and notification. Second, as some types of components are prone to cascading failures (clusters of components failing all in a short amount of time), we evaluate the ability of our monitoring scheme to cope with event bursts at scale. Finally, we study the throughput of the regime change notification as more compute time is dedicated to the analysis of an event to detect regime change.

The rest of this paper is organized as follows. Section II explains briefly the presence of regimes in HPC production systems. Section III then describes the design and implementation of our monitoring and dynamic checkpointing scheme. Section IV details our experimental setup, our evaluation methodology, and discusses the results of this evaluation and their impact on future designs for monitoring and dynamic checkpointing schemes. Section V discusses related work, and Section VI briefly summarizes our conclusions and mentions

possible avenues for future work.

II. Motivation

Extreme-scale HPC systems will be more dense and complex than current supercomputers, because of their need to deliver orders of magnitude higher performance. The density and complexity of those systems are expected to create dependencies and correlations between different hardware components that will be observed on post-mortem failure analysis. A minimalistic example of such correlations is compute nodes that share the same power supply, as is quite common in current systems. Hybrid systems with deeper storage hierarchies and multiple shared components might magnify those effects, generating more complex failure patterns.

A. Failure Frequency and Regimes

Recent studies [5] have shown the existence of a strong temporal correlation between failures in HPC systems. When a failure occurs in a supercomputer, the probability of observing more failures in the following minutes or hours is larger than the average probability of failures during the rest of the time. Several reasons may account for such behaviors. First, the root cause of a failure is not always well understood and may generate more failures if the problem is not solved correctly. For instance, overheating problems on a node might not always be linked to the workload of the node but could also be related to a cooling system malfunctioning, which could cause other nodes to fail in the near future. Accurately designating the root cause of a failure on extreme- scale systems is extremely hard. Second, even if the root cause of the failure was correctly understood, the mitigation plan to deal with the failure might not necessarily produce instantaneous results. To continue with our example of the malfunctioning cooling system, the fix of such a problem takes time, and the overheated nodes might not come back to nominal temperatures as rapidly as desired, leading to more failures. Third, not all types of failures are easy to fix in a timely manner because of their nature. For example, disk reconstructions might generate long delays in the file system, and such a drop in performance will last until the failed disk has been successfully reconstructed, a process that could take many hours depending on the amount of data previously stored.

It has been estimated that current petascale systems spend most of the time in normal regime and less than a third of the time in degraded regime. Moreover, the majority of failures occur during the intervals of degraded regime, acting in some cases like failure bursts. In many cases, those failures bursts are composed of the same type of failures, whereas other failures occur most of the time in an isolated fashion. By analyzing failure patterns, it is possible to classify failures in two groups depending on their likelihood of belonging to a failure burst. Such analysis can be enhanced by the assimilation of other events often monitored by the system, such as temperature sensors and network degradation counters.

B. Introspective Systems

Modern supercomputers gather vast amounts of data about the status of the machine, from temperature sensors to system workload, passing by a wide variety of performance counters. In addition, events such as ECC corrected errors and connection time-outs have been shown to be often linked to multiple types of system failures. Moreover, performance degradation is also frequently followed by partial or total failure of a system component. All these events could help the system forecast failure bursts and ultimately reduce wasted time. However, the amount of data generated by large-scale systems presents a challenge. Indeed, monitoring systems will need to analyze large numbers of events in real-time, without slowing the scientific application running on the same nodes that are collecting system information. Such analysis can be complex because of their lack of regularity (i.e., performance numbers, system messages) and their interaction with a large number of exterior components.

Event analysis and classification in large-scale system have been the subject of many studies, and research has shown that they can lead to good prediction results [8], [9]. One of the major challenges in this endeavor is the impact of false positives. False positives will trigger unnecessary fault-tolerant measures (e.g., proactive checkpointing) that could result in large overheads for the application if the analysis is not accurate enough. Therefore, real-time system event analysis needs to be both extremely fast and accurate in order to be effective in extreme-scale HPC systems.

In order to guarantee speed and accuracy, the event analysis should be performed in a distributed fashion, in which each node monitors its internal data and filters irrelevant events locally. This avoids the multiplication of repeated events across the system before analysis. While this is a good approach, however another challenge also needs to be addressed: the efficient notification of important events across the system. Forwarded events should be taken into account in order to make important resilient decisions, but multiple events could be generated in different parts of the system, creating a cascade of increasingly more aggressive measures to deal with the forecasted degraded regime. Such measures, again, could produce prohibitively expensive overheads for the application if not handled carefully. The rules of what is to be considered an important event and which measures should be taken must be set equally across the system.

In addition, a dynamic fault-tolerance runtime must exist in order to take advantage of the monitoring and system-wide notification system. This involves constant awareness of the state of the machine and clear rules about how to react in each situation. It might also require certain changes in the application's resilience mechanisms to make them more dynamic than current standards. Our previous work [7] shows how such dynamic runtime can be implemented and the theoretical benefits that such a scheme might offer. In this paper we focus more on the monitoring aspects of that system, in order to increase the number of correct regime

detections and decrease the application overhead generated by false positives.

III. Real-Time Monitoring and Dynamic Checkpointing

While current production HPC machines do have fault monitoring, notification, and management systems, those components are typically considered part of the RAS (reliability, availability, serviceability) software stack. As such, their purpose is to notify system administration 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 about 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 [10] 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 this kind of service with a user job.

To this end, we describe here an application-facing event monitoring system capable of notifying application runtimes when a regime change occurs in the machine. While for the purpose of this paper the only relevant events are those that might be indicative of an incoming failure, we do not expect all relevant machine resources or components to be able to self-report on anomalous behavior. Therefore, we consider that, for our purpose, an HPC production machine and a user-facing monitoring system built for it should be decomposed into two distinct pieces: a *monitoring service* and a *classification service*.

A. Monitoring Machine Events

Decoupling the monitoring of the system from the identification of which events are indicative of regime changes allows our system to perform in a distributed fashion. Indeed, we can then place the monitoring service on each node of the machine, while keeping centralized the identification of regimes and the notification of the application runtime. This will result in less messages transiting through the system, and a better performance overall. We identify three key concepts in our monitoring service: *components*, *monitors*, and *triggers*.

Components are any machine resource, software, or piece of hardware for which status can be reported. We consider that the status of a component is an arbitrary binary payload. Examples of components can be the cores, socket, and memory modules inside a compute node, the network links between nodes or the I/O infrastructure of the machine. One could report multiple kinds of information on the status of these resources, for example the temperature of cores and memories, the amount of traffic on a network link, the CPU utilization of a compute node, the free space available on a disk, the occurrence of ECC errors on a memory DIMM, or the list of running daemons on management node. When a component behaves in an anomalous way, we consider that it is possible to classify this behavior into a discreet amount of component-dependant behavior types.

Monitors are any component or piece of software that can report on the current status of a component. Monitors are responsible for polling various components for status information. As an example, one can look at *Machine Check Exceptions* [11] on most recent processors. These exceptions are interrupts generated by processors when a correctable or uncorrectable error occurs in hardware. The interrupt is then handled by the operating system kernel. On Linux, if the exception doesn't create a system panic right away, it is re-encoded and forwarded to a special user-space device. A user code can then poll this device to discover recent MCEs. Another example is a monitor that retrieves at regular intervals information from the temperature sensors in the system. This information might include the location of the sensor (e.g., fan, CPU), the current reading, and the hardware limits (critical levels).

Triggers are the algorithms that receive the status reports from monitors and generate *events* if anomalous behavior is detected. Triggers can be edge-triggered, that is, a single event is generated when a component transitions between normal and anomalous state, or level-triggered, that is, an event is generated at regular intervals while the anomalous behavior is detected. Moreover, triggers might use mechanisms similar to hysteresis (from control systems) to limit the number of events generated. A typical example of an edge-triggered trigger with hysteresis is a trigger that generate an event when the temperature of a component rise above a given value A, and generate another event when the temperature has fallen back to another value B, with B lower than A. This type of trigger ensure that components that tends to have variability in their behavior do not cause events to be generated more than necessary.

B. Identifying Regime Changes

Our classification service runs alongside the master rank of an application (typically rank 0). As most checkpoint/restart schemes handle checkpoints from a centralized location, it is reasonable to co-locate the service responsible for the notification of regime changes on the same resources.

This classification services uses **identifiers**. Those algorithms receive events from the monitoring service and identify if such events are indicative of a regime change. The classification service then retransmit the regime change detected to applications, along with the related event information. Identifiers might use information from several events coming from different sources to identify a regime change.

Our current identification service avoid unnecessary complexity by using platform knowledge to map the type and source of events it receives to the probability that an event occurs during normal or degraded regimes. While we demonstrated previously that such scheme could result in satisfying

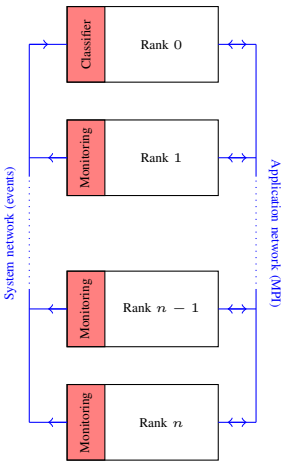

Fig. 1. Overview of a deployment our of system alongside an MPI application.

results [7], one could imagine more complex identifiers using real-time information on the system to adapt its detection.

C. Implementation

From an implementation point of view, our current monitoring service supports polling the log files containing machine check exception information, polling Linux kernel data for network and disk usage monitoring, and temperature checks using `libsensors`. Our monitor service retrieves at regular intervals information from the temperature sensors in the system, including the location of the sensor (e.g., fan, CPU), the current reading, and the hardware limits (critical levels). For all those cases, we implemented edge-triggered triggers, using platform-specific alert levels. Edge-triggered alerts are more efficient from our perspective, as they result in less events being sent across the system.

When an event is generated by our monitoring service, it follows a specific encoding. This encoding includes component, monitor, and trigger identification, as well as the status information that resulted in the event being triggered.

Those events are then forwarded to the classification service with ZeroMQ using pub/sub communications over TCP/IP. Our classification implementation is straightforward. It listens for events, classifies them, and *reacts* to them by either filtering them or forwarding them to an application runtime. The built-in identifiers are implemented with support from the event encoding, as well as user-provided platform information. The main goal of the classification is to attach information about regimes to important events before forwarding them to applications.

Figure 1 gives an overview of how our monitoring and notification system could be deployed along an application. Our classification service runs alongside the application's MPI rank 0 while the other nodes run the monitoring services. In this example, the system manager (or batch scheduler) is capable of launching user services along the application, and services and applications use separate networks. Not represented here is the unix socket connecting rank 0 of the application to the classifier to receive notifications about regime changes.

Platform information is required about the machine upon which this monitoring and notification system is deployed. This platform information is used at the monitoring level to configure triggers so that anomalous statuses can generate events and inside the classification service to react to events, decide whether a regime change occurred, and attach the relevant information to the event forwarded to applications. We expect that this platform information will originate from offline analysis of the machine behavior. For our experiment purposes, the platform information is used to map events to information about their occurrence during normal and degraded regimes.

IV. Evaluation

The applicability of our monitoring, notification, and dynamic checkpointing scheme depends on its ability to perform as intended as the number of components and associated failures increases.

To evaluate such performance, we designed several scaling experiments using the Chameleon testbed. Because this testbed provides us with root access to baremetal nodes, we are able to launch our monitoring and classifier services as actual service daemons across all nodes. All the nodes are dual-socket systems using Intel Xeon E5–2670 v3 "Haswell" processors (each with 12 cores running at 2.3 GHz). Two memory nodes are present on each node, each 64 GiB in size. Each socket also contains a 30 MiB shared L3 cache and 256 KiB L2 caches private to each core. This system is running a vanilla 4.1.3 Linux kernel. The Linux kernel configuration contains support for machine check exceptions (CONFIG_X86_MCE*) and temperature sensors (CONFIG_SENSORS*).

A. Methodology

All our experiments follow the same workflow. We start by dedicating one compute node as the master, where the classification service will be launched. Once the compute nodes are ready (boot, install of appropriate binaries), we configure, launch, and trace this service on node 0. It will run continuously, listening for messages, and outputting a trace containing relevant experimental information. We then launch the configured number of monitors, one per core, in round robin fashion across nodes (i.e. monitor 0 on core 0 in node 0, monitor 1 on core 0 in node 1, and so on). For experiments that require event bursts going through the system, we use an event injector instead of the monitor, using the same communication infrastructure and deployment strategy. We used 33 (32 monitoring + 1 classifier) compute nodes for all experiments.

Since the end goal of our monitoring infrastructure is the notification of user applications when a regime change occurs, we focus our measurements on a straightforward metric: the throughput of the classifier. We measure this throughput by measuring the number of messages the classifier can identify and forward for the duration of each experiment. This is

directly correlated with two other key characteristics of the system: the time it will take for an event to transit from the monitors to the application runtime, and the capability of the classifier to identify regime changes during the execution of an application. Indeed, if the classifier cannot handle messages fast enough, messages coming from the monitors might end up delayed, increasingly the likelihood that a failure happens before the application is notified of the regime change. Similarly, a classification too slow to filter messages cannot be used to implement more complex identification strategies (analysis of aggregated events across multiple resources for example). We consider here that the latency between the first time an anomalous behavior is present in the system and its detection by the monitor can be ignored: it depends only on the rate of monitoring and a negligible parsing/encoding phase that we already validated in our previous work.

All our experiments are run as long as necessary for each monitor/injector in the system to send 1000 messages, ensuring that we capture most of the behavior of the experiment.

B. Experiments

Fist, we look at the performance of our system under continuous load. For this experiment, we modified our monitoring daemon to generate dummy events ten times per second. This might correspond to a configuration where monitors detect too many anomalous events, either because of a bad configuration (alarm level on a level-triggered event too low) or a bad component on all nodes (bug in hardware). The classifier is here configured to discard all events (no processing). As the number of monitors running increases, we observer whether the classifier is able to keep up with the continuous influx of events.

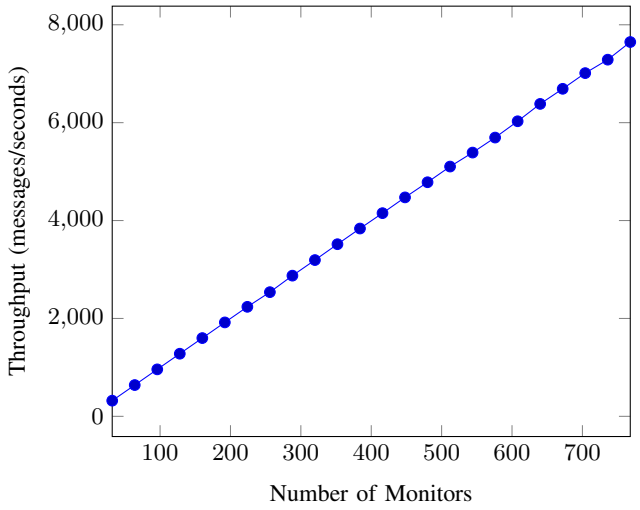

Fig. 2. Performance of our system under continuous load.

Figure 2 presents the results of this experiment. We display the throughput of the classifier as the number of messages it reacted to (by second), for a number of monitors between 32 and 768. As each monitor sends 10 messages per second, we can see that the classifier is able to filter them in as much time as it takes to generate them. From that point of view and at this scale, our classifier seems able to handle moderate traffic from each node without any trouble.

Our second experiment is similar, but this time we use an event injector instead of the monitors to trigger event bursts: short periods of time where many events occur at the same time. This situation corresponds to a more realistic scenario: a key component in a part of the machine is experiencing anomalous behavior (a power or network switch, for example) for a short period of time, triggering many smaller components to behave erratically.

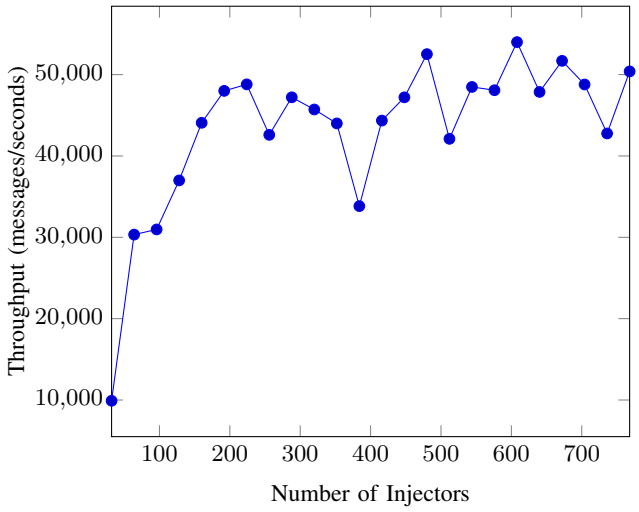

Fig. 3. Performance of our system during event bursts.

Figure 3 displays the result of this experiment. Again, we plot the throughput of the classifier, as the average number of messages filtered (by second) during the experiment. As we can see, the classifier this time is less able to cope with the number of messages arriving all at once in the system. In this setup, each injector sends 100 messages to the classifier, and all the injectors are launched at the same time. This process creates contention in the network and for the classifier, resulting in a filtering rate that fluctuates during the experiment. Nevertheless, the filtering rate hover around 5,000 messages per second. We believe this number is high enough that even in the case of cascading failures, the classifier would be able to notify the application in just a few seconds. In typical setups, such notification time would still be more than enough to adjust checkpoints until the next failure.

Our third experiment considers the identification implementation in the classifier, and its impact on the performance of our monitoring scheme. All other things being equal, the more time it takes for the classifier to identify whether an event is indicative of a degraded regime, the fewer the messages it can analyze in a given period of time. While our current implementation merely maps an event code to its occurrence rate in both normal and degraded regimes, one could easy think of more advanced classification mechanisms, using offline analysis of the failure history of the machine, for example.

For this experiment, we launch monitors the same way as in the first experiment but modify the classifier to sleep a configurable amount of time when an event is received.

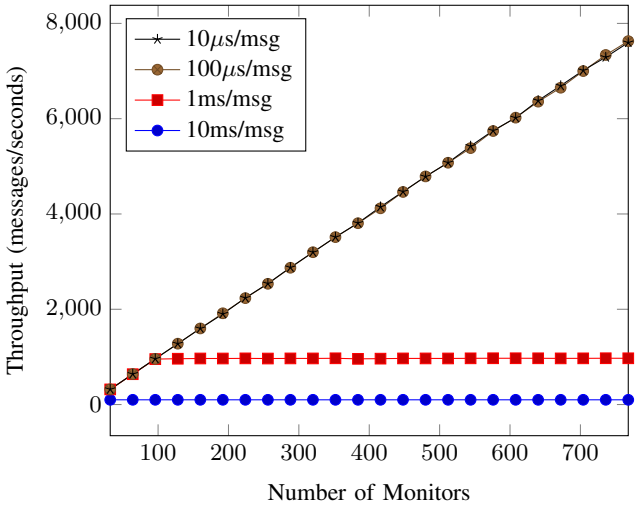

Fig. 4. Performance of our classifier depending on various event classification times.

Figure 4 gives the results of this experiment. We tested several classification times for the classifier, from 10μs per messages to 10 ms per message. As we expected, when the time to process a message increases too much, the classifier becomes unable to handle the messages sent from the monitors, ending in severely reduced throughput (100 messages per second). Accordingly, the application notification will suffer from the same issue, increasing the likelihood that a failure happens before the application runtime can be notified.

All in all, those experiments showcase the kinds of issues a notification system might encounter. If monitors send too many messages to the classifier, the classifier is more likely to have trouble handling any sudden influx of messages. Likewise, if the classifier takes too much time processing messages, it will have trouble handling even a moderate number of messages coming from the monitors. These experiments confirms that our approach should provide the right amount of flexibility. By monitoring events at the node level, we limit the number of messages that transit through the system, and we avoid loading the classifier too much. Similarly, by using platform knowledge both in the monitor and in the classifier to filter events that are not indicative of a degraded regime, we ensure that any critical event will be handled in a timely fashion.

V. Related Work

Fault tolerance has been deeply studied in the HPC field [1], [2], [12] with many works pointing at the importance of scalable fault tolerance runtimes for extreme scale computing [13], [14]. Most of those research studies focus on the design and evaluation of scalable techniques for checkpointing [15], [16]. Other have proposed rejuvenation techniques to minimize software failures [17], [18]. In addition, failure prediction has been studied [8], [19] and has showed promising results that could be leveraged for online system monitoring. While those studies are fundamental for achieving high fault tolerance at scale, they all consider a rather stable MTBF and they do not take failure bursts into consideration. For instance, the optimal checkpoint interval for an HPC application is always computed assuming a static MTBF [3], although such an assumption is not realistic.

In order to have a good understanding of the failure patterns and correlations in large-scale systems, multiple studies have been done with recent supercomputers [2], [6], [20], and they have shown larger failure rates than previously expected. Other studies have focused on failures on heterogeneous clusters [21]–[23], and they have shown that accelerators such as GPUs are often more vulnerable to memory errors than are classic devices. Similar results have been demonstrated also in petascale machines with tens of thousands of GPUs [24]. Concerning the nature of soft errors, some studies [25] have shown that failures induced by cosmic rays are very unpredictable. Others research works have studied the different types of correlations between failures that exist in modern supercomputers. Strong temporal [5] and spatial [26] correlations have been shown to exist in current systems. Others have proposed in order to exploit spatio-temporal correlations to proactively avoid failures [27]. Those studies are fundamental for the work presented in this paper, given that such correlations can be exploited by using monitoring tools and introspective systems; but the studies do not propose any integration between those correlations and the fault tolerance runtime.

Several studies about system monitoring have demonstrated the feasibility of such an approach [28], [29]. Although those works present important results about system monitoring in HPC, they do not try to make the connection with different failure regimes and dynamic resilience runtime in order to minimize wasted time. In addition to monitoring techniques, notification systems have been proposed [30] to spread information about failures and system status across the machine. Such notification frameworks are complementary to our monitoring tool proposed in this work. Event-driven time series have been modeled with semi-Markov models [31], which is also an important contribution for system monitoring tools. Log analysis has been studied in the HPC context [32] for large supercomputers such as BlueGene/P machines; however, this has not been done online in the context of failure regime detection with dynamic adaptation.

In a previous work [7], we show how to leverage failure correlation to reduce wasted time. Although that work included a preliminary analysis of the different types of event and a simple monitoring tool, the paper focuses more on giving a picture of the whole structure and not on monitoring. Here we build on top of that work, but focus solely on the monitoring techniques at large scale and how they can be improved and optimized for failure regime detection and dynamic fault tolerance adjustment.

VI. Conclusions

In this work we have presented a scalable monitoring tool capable of analysing thousands of events per second. This monitoring tool can filter unrelated events and forward important alerts to the resilience runtime for dynamic adaptation. We evaluated our monitoring mechanisms at large scale and demonstrated the scalability of the approach. The results of our evaluation show that one can implement introspective systems based on low-level system event analysis and classification.

The scalability of our scheme is ensured through two components. First, we designed the scheme so that most of the logic to detect key events in the system is included into a node-level monitoring service. This ensures that the complexity of the parsing of various log and error formats and the continuous monitoring of system elements is decoupled from the identification of regime change in the machine. Moreover, if a node experiences failures, the overall scheme will continue to work, and might be able to detect this event through error messages coming from other nodes. Second, because monitors encode events before sending them to the classification service, this component is able to implement regime change detection in a straightforward way: simply map an event code to platform-dependent knowledge of how much this event is indicative of regime changes. This last component of our monitoring scheme is critical: by integrating platform knowledge, sourced from offline analysis of the machine for example, into the monitoring and classification services, we ensure that event detection and regime change detection are tuned to each machine it is deployed unto. This can result in some performance improvements, for example by filtering sooner (at the monitor level) the case of events that occur often but are not correlated with regime changes.

Finally, we would like to acknowledge that such monitoring and regime detection scheme can only be deployed on machines where the system administrators are willing to help with the generation of the platform-dependent configurations, and by ensuring that user-provided code can have access to low-level information regarding the current state of the machine. This might proves difficult, as this information tends to be regarded as proprietary or confidential on some systems. Nevertheless, we expect that for exascale system, such information will become more readily available, as the increase in failure rates should push for more infrastructure so that application runtimes can use more advanced resilience schemes.

As future work, we want to focus on the analysis of real HPC logs for multiple large scale supercomputers and evaluate the level of accuracy of the approach.

Acknowledgments

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under Contract DE-AC02-06CH11357. This research has also received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the Mont-Blanc 2 Project (www.montblanc-project.eu), grant agreement $No610402$ and it has been supported in part by the European Union (FEDER funds) under contract TTIN2015-65316-P.

References

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

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

[3] J. W. Young, "A first order approximation to the optimum checkpoint interval," *CACM*, 1974.

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

[5] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems," in *DSN*, 2014.

[6] C.-D. Lu, "Failure data analysis of hpc systems," *Technical Report CoRR abs/1302.4779*, 2013.

[7] L. Bautista-Gomez, A. Gainaru, S. Perarnau, D. Tiwari, S. Gupta, C. Engelmann, F. Cappello, and M. Snir, "Reducing waste in extreme scale systems through introspective analysis," Argonne National Laboratory (ANL), Tech. Rep., 2016.

[8] 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.

[9] M. S. Bouguerra, A. Gainaru, L. B. Gomez, F. Cappello, S. Matsuoka, and N. Maruyam, "Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 501–512.

[10] P. Beckman, M. Snir *et al.*, "Argo: An exascale operating system and runtime research project," http://www.argo-osr.org/.

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

[12] N. Taerat *et al.*, "Blue Gene/L log analysis and time to interrupt estimation," in *ARES*, 2009.

[13] M. Y. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *NSDI*, 2004.

[14] T. J. Hacker, F. Romero, and C. D. Carothers, "An analysis of clustered failures on large supercomputing systems." in *JPDC*, 2009.

[15] L. A. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *SC*, 2011.

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

[17] A. Andrzejak and L. M. Silva, "Deterministic models of software aging and optimal rejuvenation schedules." in *Integrated Network Management*, 2007.

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

[19] S. Rani, C. Leangsuksun, A. Tikotekar, V. Rampure, and S. Scott., "Toward efficient failure detection and recovery in hpc," *High Availability and Performance Workshop*, 2006.

[20] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series 78:012022*, 2007.

[21] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers, "A large-scale study of soft-errors on GPUs in the field," in *HPCA*, 2016.

[22] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell *et al.*, "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation," in *HPCA*, 2015.

[23] D. Tiwari, S. Gupta, G. Gallarno, J. Rogers, and D. Maxwell, "Reliability lessons learned from GPU experience with the Titan supercomputer at Oak Ridge Leadership Computing Facility," in *SC*, 2015.

[24] C. D. Martino *et al.*, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *DSN*, 2014.

[25] A. Hwang, I. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of dram errors and the implications for system design," *SIGARCH Comput. Archit. News*, 2012.

[26] S. Gupta, D. Tiwari *et al.*, "Understanding and exploiting spatial properties of system failures on extreme-scale HPC systems," in *DSN*, 2015.
[27] S. Fu and C. Xu, "Quantifying temporal and spatial fault event correlation for proactive failure management," *SRDS*, 2007.
[28] K. Yamanishi, "Dynamic syslog mining for network failure monitoring," in *KDD*, 2005.
[29] 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.
[30] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A coordinated infrastructure for fault-tolerant systems," in *ICPP*, 2009.
[31] F. Salfner, "Modeling event-driven time series with generalized hidden semi-markov models," *Technical Report 208, Department of Computer Science, Humboldt University*, 2006.
[32] Z. Zheng and L. Yu, "Co-analysis of RAS Log and Job Log on Blue Gene/P," *IPDPS*, 2011.