IMPROVING FAULT TOLERANCE FOR EXTREME SCALE SYSTEMS

BY

EDUARDO BERROCAL

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Advisor

Chicago, Illinois
May 2017

ACKNOWLEDGMENT

It would have been impossible for me to complete this work without all the people in my life that held my back in the darkest times. Among those are, first and foremost, my beautiful (in all senses of the word) wife Sophie, for all the love; my parents and sister for their support and rock-like faith in me; and of course my adviser Dr. Zhiling Lan, who encouraged me to never surrender. There are surely countless technical lessons learned during a Ph.D., but the most important lesson—at least for me—is the personal and spiritual growth. For that I will forever be thankful.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Mean Time Between Failures (MTBF), now calculated in days or hours, is expected to drop to minutes on exascale machines. In this thesis, a new approach for failure prediction based on the Void Search (VS) algorithm is presented . VS is used primarily in astrophysics for finding areas of space that have a very low density of galaxies. We explore its potential for failure prediction using environmental information and compare it to well known prediction methods. Another important issue for the HPC community is that next-generation supercomputers are expected to have more components and consume several times less energy per operation. Hence, supercomputer designers are pushing the limits of miniaturization and energy-saving strategies. Consequently, the number of soft errors is expected to increase dramatically in the coming years. While mechanisms are in place to correct or at least detect soft errors, a percentage of those errors pass unnoticed by the hardware. Techniques that leverage certain properties of iterative HPC applications (such as the *smoothness* of the evolution of a particular dataset) can be used to detect silent errors at the application level. Results show that it is possible to detect a large number of corruptions (i.e., above 90% in some cases) with less than 100% overhead using these techniques. Nevertheless, these data-analytic solutions are still far from fully protecting applications to a level comparable with more expensive solutions such as full replication. In this thesis, partial replication is explored to overcome this limitation. More specifically, it has been observed that not all processes of an MPI application experience the same level of data variability at exactly the same time. Thus, one can smartly choose and replicate only those processes for which the lightweight data-analytic detectors would perform poorly. Results indicate that this new approach can protect the MPI applications analyzed with 7–70% less overhead (depending on the application) than that of full duplication with similar detection recall.

CHAPTER 1

INTRODUCTION

HPC is changing the way scientist make discoveries. Science applications require ever-larger machines to solve problems with higher accuracy. While future systems promise to provide the power needed to tackle those science problems, they are also raising new challenges. For example, resilience in HPC systems at exascale, where supercomputers are projected to have hundred of thousands of nodes and millions of cores, will drop Mean Time Between Failures (MTBF) from days and hours in current petascale machines, to just minutes [1, 2]. Moreover, typical Fault Tolerance (FT) approaches – such as classic global Checkpoint/Restart (C/R) – will be rendered useless as it will be impossible to checkpoint the whole applications' memory state in an amount of time small enough to keep a high utilization rate.

As an illustration of this problem, consider Figure 1.1. Here, the effective utilization of a system is calculated as a function of the checkpoint time and the recovery time, both relative to the MTBF. The checkpoint interval is calculated using the well known Young's Equation $\tau_{opt} = \sqrt{2C(\text{MTBF})}$ [3], where C is the checkpoint time. Supposing we want to achieve a utilization rate of more than 80%, Figure 1.1 indicates that checkpoint time needs to be kept at 1%-2% of MTBF and recovery time at 2%-5% of MTBF [1]. Assuming a MTBF of 30 minutes at exascale, we would need to have a checkpoint time of at most 20 seconds, and a recovery time of at most 1 minute. With the exponential grow that applications will experience at exascale, it is highly unlikely that global checkpoints will be able to be completed in such a limited time-frame.

Numerous studies have been devoted to the problem of reducing the large overheads imposed by the classic global C/R model. For example, [4, 5] look into data compression as a solution to scale classic C/R into the future. Although their

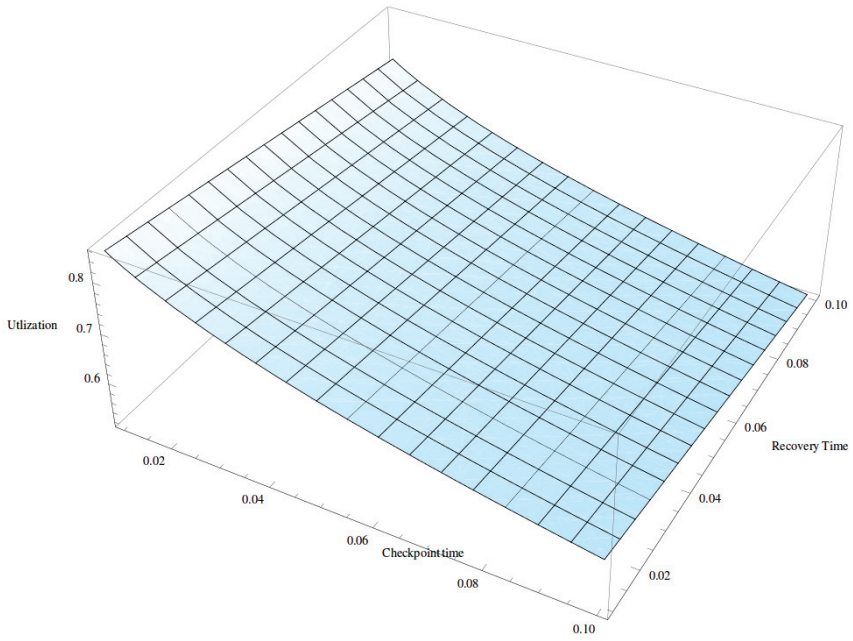


Figure 1.1. System utilization as a function of checkpoint and recovery time (from [1])

results are promising, compression does not break itself apart from the traditional C/R architecture, so it is unclear yet for how long it will be viable at extreme scales. More promising is multi-level (hierarchical) checkpointing, where local – and very fast – checkpoints to the local storage (such as SSD disks or RAM disks), or to neighboring nodes' storage, can be combined with global checkpoints so as to reduce total global checkpoint frequency [6, 7]. Furthermore, recoveries can also be done very fast by reading back local checkpoints from local storage in the case of soft errors, or from neighboring nodes' storage in the case of hard errors (e.g., the failed hardware components went down and can not be booted up again). The global checkpoint is used only in cases where local checkpoints are impossible to be recovered due to multiple failures.

Those works can be categorized as *reactive*; they do not attempt to understand failures occurring in the system, but rather to minimize their harmful effects by reducing the amount of work that needs to be redone after a failure. Another type of methods aim to decrease checkpoint cost by predicting when and where failures are

going to take place in advance. For example, if we know in advance where a failure is going to take place, we can just checkpoint that portion of the system instead of the whole memory state. In addition, checkpointing frequency can be substantially reduced if we can predict a large portion of the total number of failures [8, 9, 10].

The first part of this thesis (chapter 2) focuses on this last strategy. Traditionally, only the Reliability, Availability and Serviceability (RAS) logs produced by different types of systems has been used for failure prediction [11, 12, 13, 14]. RAS logs are also called *event logs*, since they are generated only when some event is triggered in the system (e.g., the time to complete an I/O operation is too long, or CPU temperature is too high). These logs are generated by a centralized process, usually using one-reading-at-a-time threshold mechanisms to throw events, making them extremelly simple; important patterns hidden in the data may be lost.

In this work, environmental logs are explored instead. These logs are composed of numerical values directly read from the hardware sensors spread all over the system such as fan speed, CPU temperature, input voltage, and so on (hence, they are also referred to as sensor data or sensor logs). Since every new generation of supercomputing systems come with better hardware sensors and profiling capabilities, it is of great importance to understand environmental data especially with respect to resilience. A new density-based failure prediction algorithm – based on the Void Search (VS) family of algorithms – that works with environmental logs, instead of RAS logs, is proposed.

It is well understood by the HPC community that problems will also arise as transistor size and energy consumption of future systems must be significantly reduced, steps that might dramatically impact the soft error rate (SER) according to recent studies [15, 16]. When soft errors are not detected and corrected properly, either by hardware or software mechanisms, they have the potential to corrupt the

applications' memory state. These type of errors are commonly known as silent data corruption (SDC), and are extremely damaging because they can make applications silently (without the user knowing it) produce wrong results.

The problem of data corruption for extreme-scale computers has been the target of numerous studies. They can be classified in four groups depending on their level of generality, that is, how easily a technique can be applied to a wide spectrum of HPC applications. They also have different cost in time, space, and energy. An ideal SDC detection technique should be as general as possible, while incurring a minimum cost over the application. These four groups are: hardware-level detection, full replication, algorithm-based fault tolerance, and approximate computing.

Hardware-level detection is the more general technique, since applications do not need to worry about proactively protecting their data, and it works with any type of application. One of its shortcommings is the difficulty to protect all levels of hardware (and not just main memory and maybe some levels of cache) due to the high cost imposed by adding those protections. Another problem is pointed out by recent work [17], which shows that error-correcting codes (ECCs) alone cannot detect and/or correct all possible errors.

Full replication (i.e., duplication, triplication, etc.) has been considered too costly to be used in HPC, and algorithm-based fault tolerance and approximate computing are not general enough, since only a subset of kernels and applications can benefit from them.

In this work (chapters 3 and 4), the data behavior of datasets produced by HPC applications (i.e., the applications' state at a particular point in time) is used for detection, since it has been observed that these datasets have characteristics that reflect the properties of the underlining physical phenomena that those applications

attempt to model. More specifically, these datasets are *smooth* in both time and space. This *smoothness* can be used effectively to design a general SDC detection scheme with relatively low overhead. In this work in particular, the spatial and temporal behavior of HPC datasets is leveraged to predict an interval of *normal* values for the evolution of the datasets, such that any corruption will *push* the corrupted data point outside the expected interval of *normal* values, and it will, therefore, become an *outlier*.

Of course, not all applications' data sets behave smoothly or, if they do, not all of the time. There are special cases, such as applications dealing with collisions or explosions, where sudden changes make it impossible to use any data-based solution to predict future values. In this work (chapter 4), a solution is proposed for these cases where partial process replication (i.e., replicating only a subset of all the applications' processes) is combined with data-based protection in order to achieve a good enough protection level without incurring in excessive overhead. The idea is to replicate only those processes whose data behavior makes it difficult to effectively predict future data values while, at the same time, using cheaper data-based protection on those processes where data is behaving smoothly. Furthermore, sharp data changes could be concentrated not only in a particular place in space but also in time. To deal with these applications, a solution is proposed where the number of processes to replicate can change over time elastically, allowing for a higher replication reate on those time steps where sharp changes concentrate the most.

In summary, this thesis makes three contributions. First, it highlights the importance of using environmental data for predicting *hard* errors in HPC (errors causing components in the system to stop working), as opposed to using only RAS logs. One observation for the future is that this type of information will become more prevalent and rich, since every new generation of supercomputers come with more

and more environmental sensors and profiling capabilities. Second, a new lightweight data-based solution for SDC detection is introduced which uses the nature of the evolution of the data in a large number of scientific applications (more specifically, its *smoothness* in the space and time dimensions) to check for corruptions. And third, an adaptive algorithm—combining lightweight data-based detection and partial replication—is proposed for those applications' variables where sudden changes in the data make it impossible to use only data-based detection. Resent research in SDC [18] shows that data corruptions affecting a large number of bits in a memory word, although rare, can not be easily predicted using detectable errors (usually by looking at previous errors in the same memory module), which indicates that hardware-based protection is not the only answer moving forward and that general software-based solutions, such as the ones presented in this work, will need to be taken into consideration in future research.

CHAPTER 2

PREDICTING HARD FAILURES ON EXTREME SCALE SYSTEMS

2.1 Introduction

Mean Time Between Failures (MTBF), now calculated in days or hours, is expected to drop to minutes in exascale machines [2, 1]. In the past, a number of technologies have been presented to improve fault tolerance (FT) of large-scale systems, and new resilience techniques are emerging to address new challenges posed by extreme-scale computing [6, 7, 19, 4, 5]. *The advancement of resilience technologies, however, greatly depends on a deeper understanding of faults* arising from hardware/software components. According to literature [20], faults are defined as hardware defects or software flaws. They can cause system components to change from a normal state to an error state, and the use of components in an error state can lead to failures. The work presented in this Chapter focuses on *failure prediction* by detecting faults, i.e., detecting whether a component is in an error state even though the system still considers it to be healthy.

Fault detection is critical to confine faults, avoid or limit their propagation, and recover quickly from them. For example, failure prevention methods (e.g., preemptive process migration [21]) rely on good fault detection to avoid impending failure; the checkpoint operation cost could be substantially reduced when instructed as to when and where to perform checkpoints.

Numerous works have been devoted to failure prediction on large-scale clusters [11, 12, 13, 14]. However, the totality of existing studies are based on RAS (Reliability, Availability and Serviceability) logs[1]. These logs are simple and limited since they are designed to be read and understood by humans (e.g. system admin-

[1]RAS logs are also called event logs.

istrators). RAS logs are composed of messages generated by a centralized monitor process, which queries the different components at a given frequency and may raise a warning if some value is above a given threshold. For example, the time to complete an I/O operation is too long or CPU temperature is too high. Or, it may raise an error if a component is not functioning correctly or not working altogether. On high-end supercomputers such as IBM Blue Gene machines and Cray X series systems, dedicated hardware and software facilities are deployed to collect and archive RAS events [22, 23]. By looking at one value at a time, however, RAS logs may miss important patterns hidden in the data. These patterns have the potential to signal an error state early on.

Environmental logs[2], on the other hand, are numerical values directly read from the hardware sensors spread all over the system. These sensors report environmental conditions such as motherboard, CPU, GPU, hard disk and other peripherals' temperature, voltage, and/or fan speed. Since every new generation of supercomputing systems come with better hardware sensors and profiling capabilities, it is of great importance to understand environmental data especially with respect to resilience. *This study shows that using environmental data to detect hardware faults (and hence predict potential failures) is not only possible, but it can actually outperform RAS logs-based methods.*

In this work, a system called **PULSE** (**P**redicting fail**U**re through **L**earning from hardware **SE**nsors) is proposed. PULSE is an on-line failure predictor using environmental data and it is built on a *machine learning approach using the Void Search (VS) algorithm.* VS has been studied extensively in the field of astrophysics [24, 25, 26, 27], primarily in the context of searching for regions of space with a low density of galaxies. Unlike traditional data mining algorithms which are

[2]Also known as hardware sensor logs.

focused on extracting patterns from existing data points, VS looks for patterns of empty space – or low density regions – and uses these patterns to detect anomalies in the data. Generally, VS is very suitable for situations where faults, or outliers, are scarce (or nonexistent) in the training data in comparison to "normal" data (in other words, when outliers are very difficult to characterize).

PULSE uses historic sensor data for both normal as well as faulty behavior in order to build a prediction model. Once this model is built, or "learned", we use it as a predictive tool for each instance of the same class of component we wish to make predictions for (i.e., it is *universal* for the whole system). This model construction is based on the VS algorithm, which is dependent upon the values of two critical parameters. Due to the impossibility of calculating such parameters analytically or by brute force, we use a Genetic Algorithm (GA) to find optimal values for such parameters.

PULSE's design overcomes two critical shortcomings associated with event log based prediction. Namely, locating exactly where the failure will hit, and ensuring a lead time (or time window) before the failure actually occur. Locating failures is critical if we want to make FT mechanisms for HPC succesful at extreme scales. It is understood by the community that the traditional Checkpoint/Restart (C/R) FT model will be unfeasible in the future due, basically, to smaller MTBFs and bigger checkpointing times. The latter due, in turn, by larger memory states and a widening gap between that memory and the network capability to transfer it efficiently to remote storage. Equally important, a lead time is key if we want to ensure systems have the time to react accordingly by performing the necessary FT actions.

PULSE is evaluated using real logs from four months of 2012 as well as one production year (2014) of the Mira supercomputer, a 48-rack IBM Blue Gene/Q at Argonne National Laboratory (ANL) [28]. The experiments show that PULSE can

detect a large number of faults maintaining an extremelly low false positive rate. More specifically, PULSE can achieve an f-score (see Equation (2.1) for a definition) between 0.716 (recall=0.63, precision=0.83) for a lead time of 1 minute to an f-score of 0.55 (recall=0.54, precision=0.55) for a lead time of 5 minutes. These results seem to be better than those obtained by existing prediction studies based on RAS logs [11, 13, 14], and led me to believe that the use of this design on environmental data had real potential to improve failure prediction. However, as it will be pointed out in Section 2.7, the predicted failures during the production year of 2014 were not failures as such, but rather scheduled system shutdowns.

Although this work focuses on IBM Blue Gene/Q systems, the design of PULSE is generally applicable to other clusters which collect environmental data as well. The proposed work is the first of its kind and therefore will open a new research direction in the area of fault prediction on extreme scale systems.

The rest of this chapter is organized as follows. Section 2.2 gives an overview of environmental logs, VS algorithms, and GAs. Section 2.3 presents the design of PULSE. In Section 2.4 we show the specific details of our data analysis as well as our GA-based parameter optimization. The results for the experiments with 2012 data can be found in Section2.5. The results for the experiments with 2014 data can be found in Section 2.6. Section 2.7 presents a discussion about the findings with PULSE. Finally, Section 2.8 ends with related work.

2.2 Background

In this section, environmental data and the VS algorithms are described, laying the foundation to further explain how the two can be combined in order to design an effective prediction algorithm. Genetic algorithms are also briefly described, which are used in the design to find the optimal parameters for the VS-based predictor in

PULSE.

2.2.1 Hardware Sensors and Environmental Data. Computer systems are commonly deployed with various sensor chips for health monitoring. In particular, *environmental sensors* are often deployed on various locations inside the computer. These sensors report physical readings such as motherboard, CPU, GPU, hard disk and other peripherals' temperature, voltage, current, fan speed, etc. Depending on the system, these environmental data can be accessed via different facilities in an "out-of-band" fasion. In the case of general Linux clusters, this usually takes the form of a separate processor polling the sensors and making the information available via an IPMI interface on a lower bandwidth Ethernet transport network not used for application traffic. On capability systems such as Cray XT/XE/XC and IBM Blue Gene systems, they provide their own "out-of-band" monitoring mechanisms for collection that utilize dedicated processors and networks [22, 23].

One of those capability systems is Mira, the IBM Blue Gene/Q system currently in production at the Argonne Leadership Computing Facility [28]. Mira has a theoretical peak of 10,066.3 TFlop/s, and a Linpack performance of 8,586.6 TFlop/s. Each of the 48 racks has two midplanes. Each midplane is composed of 16 node boards each having 32 compute cards. The compute card is composed of one chip module and 16 GB of DDR3 memory. Finally, the computing chip has a single 18-core PowerPC A2 processor [29]. Out of these 18 cores, 16 are for applications, one is for system software, and one core is left inactive. Each core has four hardware threads. Thus, BlueGene/Q has a total of 1,024 nodes per rack, or 16,384 cores per rack.

The machine is built with a number of hardware sensors placed on multiple components all over the machine. Depending on the sensor, the type of the data collected can be temperature, coolant flow and pressure, fan speed, voltage, or current. The sensor data is gathered by the Midplane Monitor and Control System (MMCS)

Table 2.1. An example of environmental information

location	**time**	**inletFlowRate**	**coolantPressure**	...
R1A-L	2012-09-01 00:03:03	2680	3949	...
R0F-L	2012-09-01 00:03:03	2417	4049	...
R10-L	2012-09-01 00:38:10	2673	3985	...

and stored in an IBM DB2 relational database with a given periodicity of about 4 minutes on average (although it can be configured to be anywhere between 1 and 30 minutes).

Table 2.1 shows an example of sensor data for the coolant system in Mira. Location is a human readable unique identifier, and all sensor data are strictly numerical. Since the coolant system is installed with at a rack granularity, locations here only represent racks (R10-L is the coolant system for rack number 10). In general, sensors can be located in a variety of components such as service cards, node boards, I/O boards, coolant systems, bulk power modules, and optical modules. The MMCS organizes sensor information in different tables based on the component which the sensor applies to. There are tables for the service card, the node board, the I/O board, the coolant monitor (as shown in table 2.1), and so on. More information about location IDs and environmental data in Blue Gene/Q can be found in [22].

Since components in the system already form a natural hierarchy, it is possible to think about sensors as forming a hierarchy too. This is useful when trying to consider which sensors can provide pragmatic information for potential faults in a particular component. For instance, in the case of the Blue Gene/Q, if the power to a rack fails, all 1,024 nodes in that rack are going to go down, inevitably experiencing a failure.

Hardware faults are extracted from the RAS logs, which in Blue Gene/Q are

categorized into informational (INFO), warnings (WARN) and fatals (FATAL). Only FATAL events are severe faults that presumably lead to failures [22]. For the purpose of this study, only FATAL events directly affecting hardware components of interest are considered as faults.

2.2.2 Void Search Algorithms. Void Search (VS) algorithms are a family of methods aimed at finding regions of empty – or low density – space for a given set of data points. These empty regions are known as voids, and in astrophysics they are essential for studying galaxy formation and the structure of the cosmic web [24]. For PULSE, voids are essentially patterns of feature space for low density data regions. This contrasts sharply with traditional algorithms which search for patterns of feature space for existing data points. Voids can be of great interest in the case where one of the data classes in a two-class learning problem is hard to characterize.

Formulation of the problem is quite simple: given a space of N dimensions, the mission of a VS algorithm is to find regions with a very low density of points[3]. Figure 2.1 shows an example of voids for a given set of randomly generated points in a 2-dimensional space. In this particular example, only voids with zero density are shown.

The definition of a void, however, can change depending on the nature of the data. For example in [25], the authors partition the space into a grid of cells of equal size and declare every cell that meets the density criteria as a starting void. Furthermore, they merge voids only if its centers are closer to each other than they are to any existing data point. This definition avoids the creation of tunnels between voids, which in turn helps approximate voids by large spherical or elliptical shapes. This makes sense when working with the structure of the universe, since it has been

[3]Definition of low density depends on the particular algorithm.

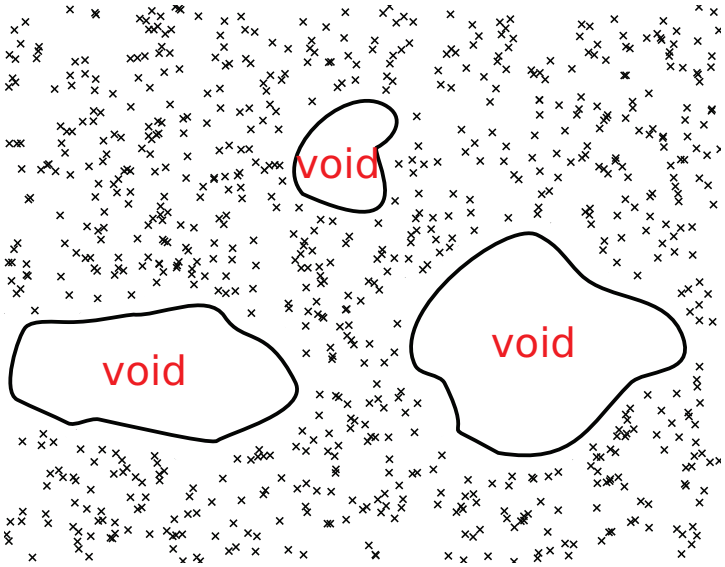


Figure 2.1. Three voids in a given set of randomly generated points

observed that voids tend to become more spherical over time due to gravitational instabilities causing the collapse of high density regions [30]. Another definition can be found in [26] and [24], where void centers are defined as very low density points, and "walls" between voids are defined as regions surpassing a density threshold relative to its surrounding voids' centers. This definition tends to fit very well for data that spreads like a web of walls fencing low density regions (similar in structure to a Voronoi tessellation).

Independently of how we build voids, outliers can simply be discovered by checking whether they fall into one of the voids. It is clear that the more stable a void is[4], the more accurate the algorithm will be as time progresses. My observations indicate that environmental data tends to be very stable under normal conditions. For illustration, consider Figure 2.2, where temperature readings for a period of one

[4]A stable void is a void whose density value does not change drastically when new data is used to build voids.

month in a fault-free node board are shown. As the figure shows, temperatures stay fairly stable between 25 and 35 degrees Celsius throughout the period. Nevertheless, it is possible to imagine a scenario where voids are adapted over time given major changes in the data.

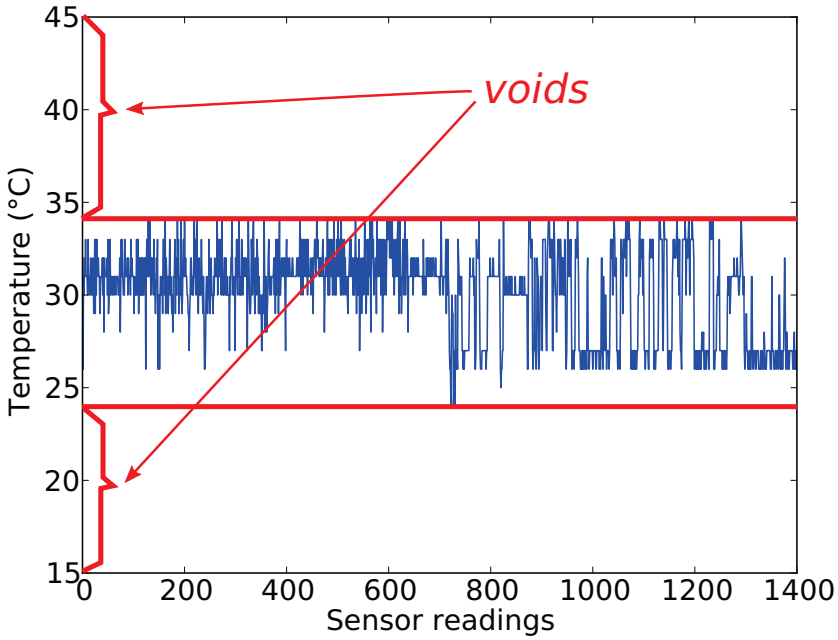


Figure 2.2. Temperature readings for a period of one month in a non-faulty node board. Voids in this case (one sensor only) are the values outside those observed during normal operation

2.2.3 Genetic Algorithms. Genetic Algorithms (GAs) were first proposed by Holland [31] in 1975 as a way to design algorithms able to find the optimal solution to an optimization problem for which an analytical – or other (such as brute force) – solution does not exists or it is very hard to compute. GAs search for solutions by imitating the process by which natural selection adapts populations of species to their environments. The natural selection process works by the so called "survival of the fittest" theory. This theory states that the most able and adapted individuals in each population of a particular species have a better chance of survival and reproduction, having then also a better chance of passing their genetic information to the next

generations. Combining survival of the fittest with random mutations, additions, and deletions of DNA material in some individuals, natural selection ensures that groups of individuals as well as whole species can adapt to their environments, or evolve if the conditions in the environment change to some degree, i.e., with the introduction of new predators, climate change, etc.

An optimization problem can be formulated as a GA as follows. First, a particular enconding for solutions needs to be defined, which in GA terminology are called individuals. This enconding translates a particular solution to a DNA string that can be operated upon by the GA during reproduction and mutation. The typical way to accomplish this is by using the binary representation of the solution, where the bits represent the genes of the individual. The second step in formulating a GA is to define a *fitness function*. This function is usually the problem we want to optimize, which takes as input a particular solution (DNA) and produces as output a value representing the fitness of that solution. It is customary to normalize the output of this function to be between 0 (not fit) and 1 (perfect fit) [32]. In the third step, we need to define what is the strategy for the selection of which individuals will reproduce, and how much, among the whole population. Different strategies exists. For example, one can define an *elitist* strategy where only a percentege of individuals (the most fit) reproduce. Another strategy could be that everyone is elegible to participate but the probability to be picked for reproduction depends on the fitness value of the individual. As with other optimization algorithms, there is not a silver bullet here. Both strategies have their own characteristics and chosing one or the other depends significantly on the problem at hand.

Finally, the two strategies for the genetic operations of reproduction (also known as crossover) and mutation are defined. In the classic crossover operator, two parents and a crossover point in the DNA (e.g., the middle) are picked and an

offspring created by using the left part of the DNA from one parent and the right part from the other. More complex strategies can be defined (like multiple crossover points and/or parents). For mutation, a probability P_m is defined. After the new offspring is created during the crossover phase, the GA will "flip" every bit (i.e., change the value from 0 to 1 or from 1 to 0) in the DNA with probability P_m. Again, there are not silver bullets for the crossover strategy or for the value of P_m. Some values could make the GA converge to a local maximum very fast, while others may lead to a global maximum but in a very long (and maybe unacceptable) time. These and other problems related to the nature of GAs are out of the scope of this work; ample literature exists for those who wish to go deeper.

Algorithm 1 Genetic Algorithm.

```
1:  procedure GA(N, t, P_m)
2:      P ← initPopulation() /* DNAs created randomly */
3:      F ← sort( calcFitness(P) )
4:      i ← 0
5:      while i < N and F[0] <= t do
6:          Pa ← selectParents(P, F)
7:          P ← doCrossover(Pa)
8:          P ← mutateGenes(P, P_m)
9:          F ← sort( calcFitness(P) )
10:         i ← i + 1
11:     end while
12:     return P, F
13: end procedure
```

A high level pseudo-code description of a typical GA is shown in Algorithm 1. The first part (lines 2-4) is initialization. Here one can create an initial population by randomly generating DNA sequences. Starting from a checkpoint of known solutions from previous runs is also possible. In the GA, the *calcFitness* function runs the defined fitness function on each and every individual in P. The main loop (lines

5-11) performs the main steps in the GA until either of two conditions are met: (1) The fitness value of the best solution is above some defined threshold t, or (2) we have reached the defined number of maximum iterations N.

GAs can be both distributed and parallelized. For example, it is possible to run multiple GA instances where every instance evolves its own population independently. This can avoid the problem of one single population getting trapped in a local maximum. At the same time, GAs can also be parallelized internally by running the fitness function on individuals in parallel (inside *calcFitness* in lines 3 and 9), as it is usually the case that there is not any dependency between single individuals.

2.3 Design Overview

The prediction algorihtm in PULSE uses a period-based design for data aggregation [33]. This design, illustrated in Figure 2.3, is defined by four parameters: (1) an *evidence window* W_e, during which information from the different sensors in each component is gathered to build a vector representing the state of the component (or components) for which to make predictions (when more than one value is read for a single sensor, a new value is created using the average); (2) the size of the *time step* T_s, or how much the windows are slided forward to make the next prediction; (3) a *lead time* T_l, or the minimum amount of time available for the system to react until the failure hits; and (4) a *prediction window* W_p, which is the time window in the future during which, if a positive prediction is made, the failure will most likely hit.

The first step in the design of PULSE is to determine an appropriate granularity for data analysis. For example, analyzing the whole machine at once trying to make our algorithm predict faults for the entire system was considered. However, this approach proved infeasible as the number of features became unmanageable. If we think of sensors as the features for learning, the whole machine has hundreds of

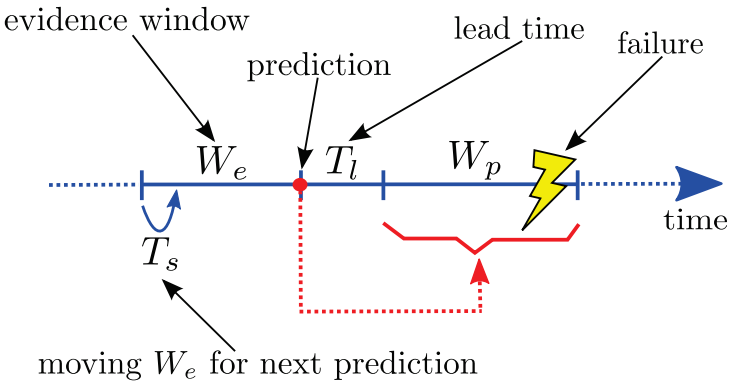


Figure 2.3. Illustration of period-based failure prediction. The figure corresponds to a particular moment in time when a prediction is made (red dot). After this, W_e is slid forward for T_s minutes to gather new data and do another prediction

thousands of features. Another problem with this centralized method is the difficulty of locating faults. Predicting fault location is key in order to make preventive actions (such as migrating processes or doing local checkpoints) effective in extreme scale systems.

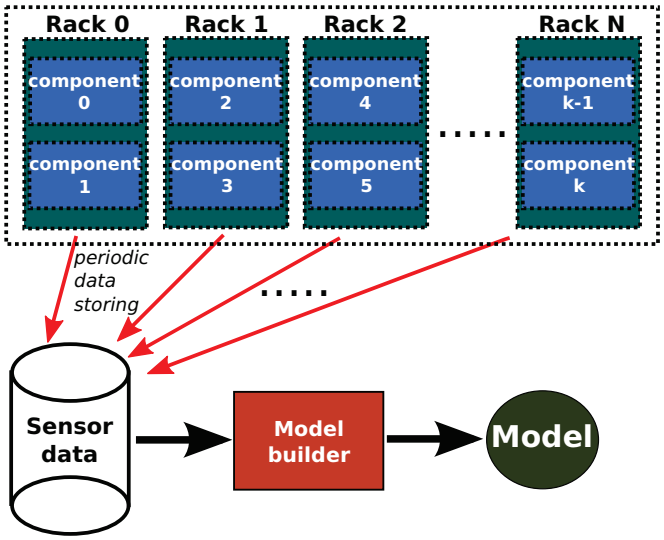
Instead, each hardware component is analyzed independently using only sensors that are relevant to that component. Since prediction is performed per component, it is possible to know the locations of the detected faults, (which can lead to future failures) immediately. Although this design allows for a significant reduction in the number of features, a reduction of dimensionality is still needed in order to avoid a very complex model.

The high-level design of PULSE, presented in Figure 2.4, is divided into two major parts. The first, presented in Figure 2.4(a), corresponds to the building of the predictive model. For this case, historic sensor data for both normal as well as faulty behavior is used. This historic data is what is usually known in the Machine Learning (ML) community as the *training set*. Once the model is built, or "learned", it is used as a predictive tool for each instance of the same class of component we wish to make predictions for. This is shown in Figure 2.4(b). New data for each component

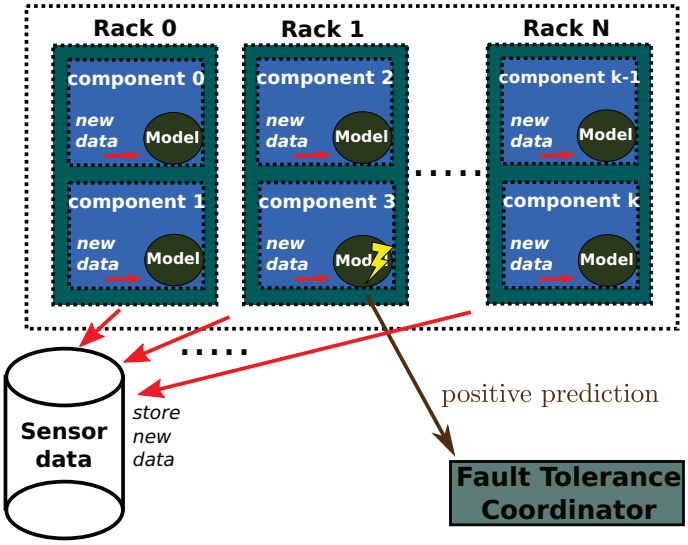
is gathered regularly in an "on-line" fashion and fed to the prediction algorithm to determine whether a failure is likely to occur in the near future or not. If this is in fact the case, an alarm can be sent to the centralized component in the system which deals with FT actions such as checkpointing or process migration. Moreover, this new data is also stored in the event that the model needs to be re-computed in the future.

It is important to point out that Figure 2.4(b) is a general design and by no means imply that all the predictions need to always be carried out localy in each component. It is also possible to have a centralized process receiving streams of information from the different components (in much the same way as to how it is streamed now to be stored remotely) and making a prediction for each one of them independently (this is, of course, assuming predictions can be done fast enough for them to be useful). Generally, modern petascale systems have spare local resources, such as spare cores, in order to perform system tasks without affecting the performance of running applications. It is also valid to assume that this will continue to be so moving forward to exascale machines. PULSE's design assumes such an scenario, wihch makes prediction naturally scalable.

The second step in the design of PULSE involves choosing the system resolution, i.e., for which type of component we make predictions. Here it is decided to use midplanes. Midplanes are the minimal scheduling unit in Blue Gene/Q supercomputers, making them more suitable for FT tasks at the system level than other components such as node cards, for example. It is possible to say that our algorithm is *midplane-centric*. Potential faults in other components are not relevant unless those components can potentially induce a faulty behaviour in a particular midplane. For example, link card failures are only important in the event that they make a particular node, or a subset of nodes, of a particular midplane experience one or more



(a)



(b)

Figure 2.4. The high-level design of PULSE is divided into two major parts. The first (a), model building, is only required once every few months. Historic data is used to build the predictive model. Once a model is learned, a new prediction (b) is done every time new data is available. Realize that the model used in each component is exactly the same

faults.

Sensors relevant to each midplane are selected based on the hierarchy of components inside the system. Sensors from the rack's bulk power and coolant environment,

the link cards connected to the midplane's node cards, the node boards where the node cards are inserted, and the sensors in each of the 512 node cards in a midplane are compiled, making a total of 1,067 sensors.

2.4 Methodology

In this section the methodology used for the different data analyses performed in PULSE is presented. First, how dimensionality reduction is done is explained, followed by an explanation of our VS-based algorithm (both learner and predictor). The section ends by describing how the parameters for the VS algorithm are optimized.

2.4.1 Dimensionality Reduction. To reduce dimensionality principal component analysis (PCA) [34] is used, a well known and widely used tool in data analysis. PCA transforms the features of a given data set to a new set of uncorrelated features called principal components (PCs) which are a linear combination of the original ones. PCA quantifies the importance that each component has in describing the variability of the data, allowing us to use fewer dimensions by picking the most relevant components, while still preserving all the important patterns in the data. PCA, however, has some limitations, such as the impossibility to project the patterns found in fewer dimensions (such as voids) back to the original dimensions. Even so, this is really not a problem in PULSE because the relevant information is knowing if a particular component is going to fail; it is not of concern as to why that particular component is going to fail. Root cause analysis can always be performed post-mortem with the untransformed log data if necessary.

2.4.2 Void Construction. Algorithm 2 shows how voids for a given training set "midplanes" in an n dimensional space are created, where the training data corresponds to points in time with no faults for each and every midplane in the system. In order to discover where voids are, the algorithm needs to discretize the feature space

in units called cells. Here δ represents the longitude of each cell's side in the grid of cells. In line 2 the limits L of the grid of cells are calculated. The limits are the minimum and maximum values of every dimension for which we have a data point, adjusted so every side of the grid has a natural number of cells. This is equivalent to drawing an enclosing rectangle around a cloud of data points in a two dimensional space.

Algorithm 2 Void construction algorithm.

```
1:  procedure CreateVoids(midplanes, n, δ)
2:      L ← calculateLimits(midplanes, n)
3:      M ← initCells(L, n, δ)          /* all False */
4:      for i ← 0, |midplanes| do
5:          D ← midplanes[i]
6:          for j ← 0, |D| do
7:              if D[j].class ≠ fault then
8:                  p ← calcPosition(δ, n, L, D[j])
9:                  M[p] ← True
10:             end if
11:         end for
12:     end for
13:     return M, L
14: end procedure
```

Line 3 initializes M, which will be the returned set of voids (this is the model to be learned). This initialization mostly depends on the data structure used to represent voids. In this case, two data structures are considered: (1) *A bitmap*, or array of bits, where every position in the bitmap represents a particular cell, and a (2) *k-ary* tree with a maximum height of n. For the bitmap, the bit representing a cell is set to one (or "True"), if the cell holds data points corresponding to normal hardware functioning. Moreover, if the bit is zero (or "False"), it is defined that the cell is a void. It is possible to see that the algorithm is an extreme case of VS where density has to be zero for a cell to be regarded as a void. Other options are possible.

For example, instead of a bit, a quantity indicating point density on the cell can be stored and a threshold defined to determine if the cell is, or is not, a void. Realize that operations in this type of structures are constant time $O(1)$.

When the dimensionality used is high (e.g., bigger than 10), however, the curse of dimensionality makes voids in the edges extremely large. A more memory efficient structure in this case is a tree with n levels, or one per dimension. A point can be stored as a leave in this tree, and voids are defined as missing branches and leaves. Although operations in this case are bounded linearly by the number of dimensions $O(n)$, n is usually kept relatively small, specially compared to the number of points $|D|$. Recall that PCA sorts components by their importance in describing the variability of the data. Hence, every new dimension added has an ever decreasing value at a linearly increasing computational cost and, in the case of array structures, at an exponentially increasing memory cost.

An example of these two data structures can be found in Figure 2.5. In the example, only two dimensions ($n = 2$) are used, with 5 cells in the first dimension and 3 in the second one.

One thing that is worth noting about Algorithm 2 is its simplicity with regard to parallelization as it is embarrassingly parallel with respect to the outer loop (line 4). Although the model M is common for all the midplanes in the system, independent models can still be learned in parallel and combined at the end in a "all-gather" fashion. Taking this into consideration, it is possible to say that the algorithm scales linearly with the size of the supercomputer. How much time does it take to train the model for a single midplane is shown in Section 2.6.2. Under this design, it is possible to make every midplane compute its own voids and detect its own faults while having a centralized approach doing all the FT logic, i.e., process migration, checkpointing, etc.

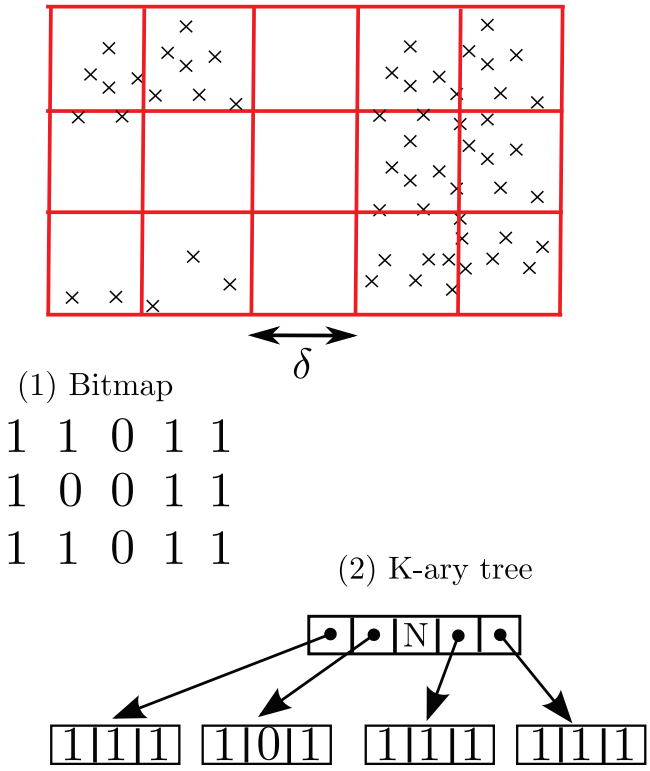


Figure 2.5. Data structures used for VS. The (artificial) example data has 2 dimensions ($n = 2$), with 5 cells for the first and 3 for the second

2.4.3 Detection. Once the void model M is calculated, faults can be detected using Algorithm 3. Here $\vec{d}$ represents a given environmental data vector, which is expected to be previously projected to the new n PCs. If the cell to which $\vec{d}$ fits is found to actually be a void, then *True* is returned indicating faulty behavior and that a failure is likely to hit the midplane soon (depending on the lead time); proactive measures, then, are needed for this midplane. The runtime of Algorithm 3 depends on the runtime of the *calcPosition* function, which is either constant $O(1)$ in the case of the bitmap array or linear with respect to the number of dimensions $O(n)$ in the case of the k-ary tree. In practice it is found that, for the Blue Gene/Q environmental data, predicting faults for a midplane always runs under 1 millisecond in any average modern CPU.

Algorithm 3 Outlier detection algorithm.

1: **procedure** DETECT($\vec{d}, M, L, n, \delta$)
2: $p \leftarrow$ calcPosition($\delta, n, L, \vec{d}$)
3: **if** $M[p] = False$ **then**
4: **return** $True$
5: **else**
6: **return** $False$
7: **end if**
8: **end procedure**

2.4.4 Parameter Optimization. In this section, we describe how GA is used to find the optimal values for the parameters of the VS algorihtm. More specifically, two parameters that are key for VS algorithm need to be optimized [35]: (1) the number of dimensions n in the PC-projected data, and (2) the side of the cells δ created by our VS algorithm.

This step assumes that the parameters related to the period-based data aggregation (PBDA) are given (these are evidence window W_e, size of time step T_s, lead time T_l and prediction window W_p; for more information see Section 2.3), and optimize the VS parameters n and δ. As it is shown in Section 2.2.3, the first step in defining a GA is to choose an enconding for the parameters we want to optimize. Starting with the number of dimensions, n is restricted to be no more than 1,023 which gives us a representation using only 10 bits. Although the maximum number of dimensions is 1,067, solutions with more than 100 dimensions are very rare, so it is unlikely that optimal solutions will have more than 1K dimensions. This is due to the curse of dimensionality. The more dimensions we use, the more prone we are to predict false alarms as the sizes of the voids grow exponentially.

The parameter δ is not as trivial as n, in the sense that it is not an integer but a real number. In modern computer architectures real values are represented

as floating-point numbers. A major problem with this representation is that the distribution of represented numbers is biased toward zero, i.e., there are more strings of bits representing numbers close to zero than further away from it. Fortunately, there are multiple options to create a DNA representation that will fix this problem. The most common one is to translate the number from floating-point to fixed-point by fixing the number of decimals we want to maintain [32]. It is possible to simply represent a real number as a combination of two integers (one for the integer part proper, and one for the decimals). In the case of δ, an after observing the properties of the PCA-projected data (e.g., data ranges for different dimensions), δ is set to be no bigger than 16 with 4 decimals for the non-integer part. Moreover, δ can be represented using 4 bits for the integer part and 14 bits for the decimals, making a total of 18 bits. Adding the bits for n, each individual is represented as a 28-bits DNA string.

As fitness function, the traditional F-measure is used, also known as F_1 score or f-score, presented in Equation (2.1).

$$F\text{-measure} = 2 \times \frac{\text{recall} \times \text{precision}}{\text{recall} + \text{precision}} \tag{2.1}$$

Here *recall*, presented in Equation (2.2), is the rate of total failures predicted among all failures in the data while *precision*, presented in Equation (2.3), is the rate of real failures among all the predicted failures. Realize that the F-measure is just the armonic mean of recall and precision.

$$\text{recall} = \frac{\text{true\_positives}}{\text{true\_positives} + \text{false\_negatives}} \tag{2.2}$$

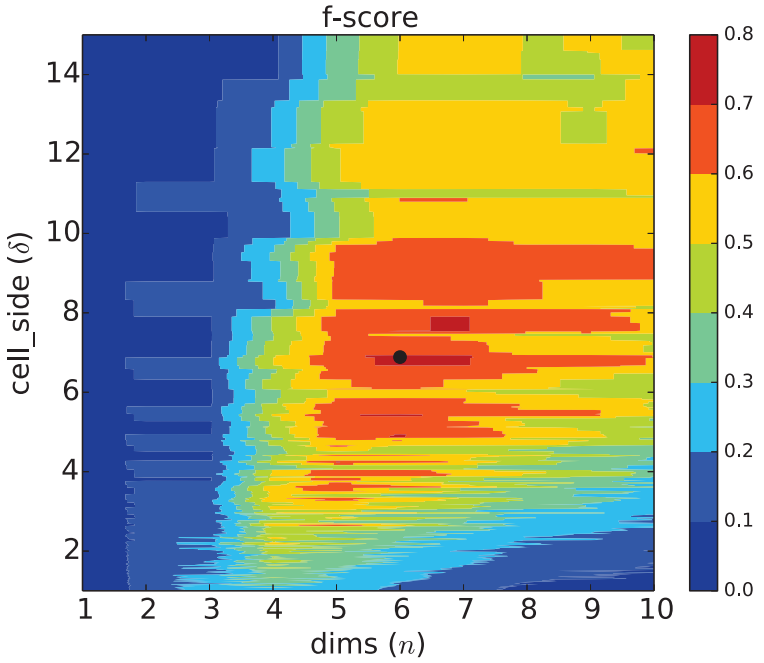


Figure 2.6. Contour plot of the GA optimization results for the VS parameters n (dims) and δ (cell_side). The PBDA parameters used are $W_e = 30$ minutes, $T_s = 15$ minutes, $T_l = 5$ seconds, and $W_p = 30$ minutes. The best solution found (f-score= 0.722 for $n = 6$ and $\delta = 6.8852$) is shown as a thick black dot

$$\text{precision} = \frac{\text{true\_positives}}{\text{true\_positives} + \text{false\_positives}} \tag{2.3}$$

F-measure fits well into the GA since it measures the accuracy of a prediction algorithm (such as VS) with just one number ranging from 0 to 1.

With respect to the reproduction selection, an *elitist* strategy is chosen where only the top 10% of the most adapted individuals are used to produce the 90% of individuals of the next generation. The other 10% are the parents themselves (the best solutions from one generation to the next are maintained). For crossover, the classic approach of two parents is used (making sure that two different parents are always picked) and middle point crossover. Finally, the probability of mutation is set to $P_m = 0.1$.

The parameter optimization is done using data from the (production) year of 2014. The year is divided into two parts. The first, corresponding to the first 8 months of the year, is used for the pupose of training. The second part, on the other hand, is used for testing only.

An illustration of one of our GA optimization runs is shown in Figure 2.6 in the form of a contour plot. The PBDA parameters used are the following: Evidence window $W_e = 30$ minutes, size of time step $T_s = 15$ minutes, lead time $T_l = 5$ seconds, and prediction window $W_p = 30$ minutes. Reasons are given in Section 2.6 as to why these values are used initially and also show results for other combinations of these parameters.

As it is possible to see, the best solutions (red) seem to cluster between a δ (cell_side) of 4 and 10. Moreover, solutions improve dramatically as we move from 1 dimension to 5. After 8 dimensions, solutions decline slowly as more dimensions are introduced (in the figure we only show up to 10 for clarity). The best solution (marked with a thick black dot) corresponds to the parameters $n = 6$ and $\delta = 6.8852$, with an f-score of 0.722 (recall=0.64 and precision=0.83).

2.5 Evaluation for 2012

In this Section, the detection design is evaluated using a four-month RAS and environmental log collected from the Mira supercomputer from September 1st to December 31st of 2012. For this data, a node-centric analysis is used, instead of a midplane-centric one, and the parameters n and δ are not optimized. In addition, only the nodes with four or more faults are used for the analysis. In total, 4325 hardware faults from 362 nodes during a period of 4 months are evaluated.

Three sets of experiments are conducted: first, the detection accuracy of our design is assessed, second, runtime is evaluated, and finally, VS is compared with

other outlier detection algortihms.

All of the evaluations, with the exception of the clustering algorithm (discussed later), are done using 10-fold cross validation, which is a widely used technique to evaluate a particular algorithm with a given set of labeled data (data from which we know the class). It first randomizes the order of the data. Then, it runs the algorithm 10 times, splitting the data into 10 (almost) lculates the average of ten runs to compute the desired metrics. for testing in every run. When the 10 runs are completed, it calculates the average of ten runs to compute the desired metrics.

2.5.1 Metrics. Three metrics for evaluation are used: *sensitivity*, *specificity*, and *S-measure*. Sensitivity represents the rate of total faults that were predicted among all faults in the data. This metric is also commonly known as *recall*, and it can be computed as follows:

$$sensitivity = \frac{Tp}{Tp + Fn}, \tag{2.4}$$

where Tp and Fn are the true positives (real faults that were detected) and false negatives (real faults that were missed) respectively. Specificity, on the other hand, represents the rate of total non-faults that were predicted among all non-faults in the data. This metric can also be called *negative recall*, and it is computed this way:

$$specificity = \frac{Tn}{Tn + Fp}, \tag{2.5}$$

where Tn and Fp are the true negatives (real non-faults that were predicted as such) and false positives (real non-faults that were detected as faults incorrectly) respectively.

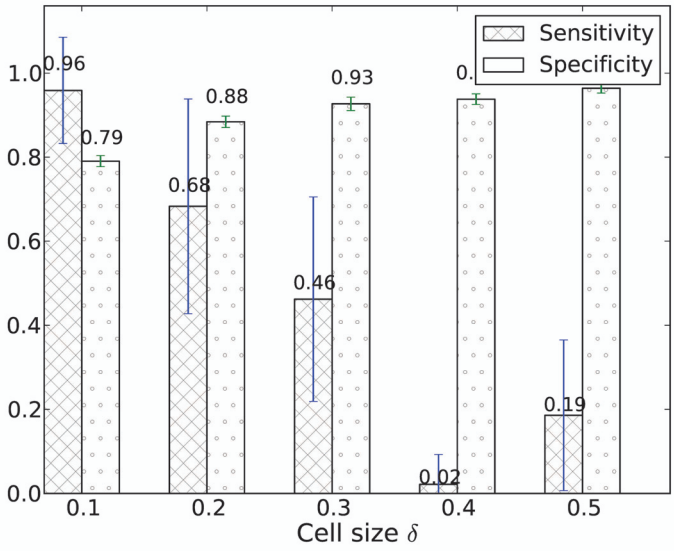
In addition, a new metric called *S-measure* is introduced (inspired by the

existing F-measure) to compare the results of our VS with that of other algorithms. This metric combines sensitivity and specificity given they have an equal weight in the analysis, i.e., it is an armonic mean of the two. It is defined as follows:

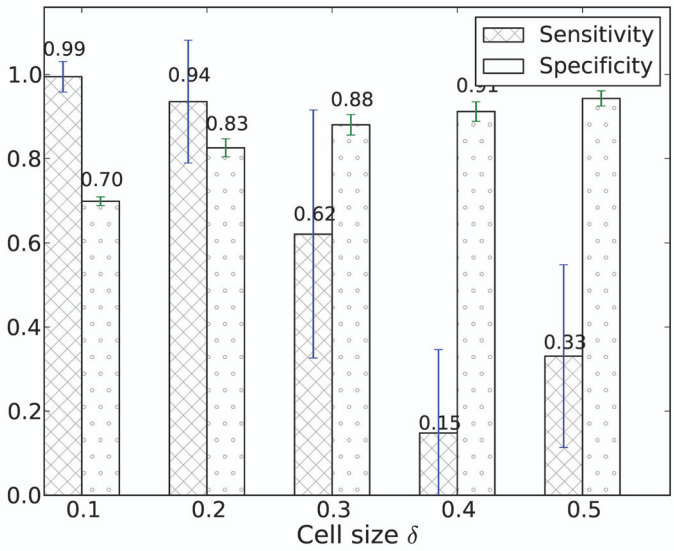
$$S\text{-}measure = 2 \times \frac{Sensitivity \times specificity}{Sensitivity + specificity} \tag{2.6}$$

2.5.2 Detection Accuracy of the VS Algorithm. In the first set of experiments, the VS based algorithm is assessed under various parameter settings. The results are presented in Figure 2.7. The two graphs presented correspond to the same experiments using different numbers of dimensions – 2 dimensions for (a) and 3 dimensions for (b). Both graphs evaluate VS by changing the δ parameter from 0.1 to 0.5. In both cases sensitivity increases as δ decreases, reaching values greater than 0.9 for $\delta <= 0.1$ in (a) and for $\delta <= 0.2$ in (b). However, it is possible to see that specificity tends to decrease as cell sides decrease. This is expected since smaller cells do ultimately produce some overfitting as number of voids increases. This results in more "normal data" falling into voids than it would do with a bigger δ.

Fortunately, specificity never gets below 0.7 in the case of 3 dimensions and 0.79 in the case of 2 dimensions, which means that, at most, preventative actions are performed 30% of the time they should not have. Although this may seem like a lot at first, these predictions are done for every node separately. Under a hierarchical checkpointing scheme, for example, checkpoints to the local SSD disk – or even to a neighboring node's SSD disk – can be done in seconds. Nevertheless, [9] and [10] prove that, when combining checkpointing and prediction, checkpointing algorithms benefit the most when the salient characteristic of a predictor is its sensitivity (recall). Missing a failure is much more expensive than having some false alarms. In other words, *it is better to be safe than sorry* [9]!



(a) $n = 2$



(b) $n = 3$

Figure 2.7. Results for the VS based algorithm using 10-fold cross validation over a period of four months of 2012. The results shown are averaged across the 361 faultiest nodes. Two plots are presented, representing the cases after feature reduction to two and three dimensions, respectively

2.5.3 Comparison with Other Algorithms. Another set of experiments were conducted in order to evaluate how other popular outlier detection algorithms could perform using environmental data for fault detection. More specifically, two classes

Table 2.2. Comparative study of our VS based algorithm with other detection algorithms

	Sensitivity	**Specificity**	**S-measure**
NB	0.493	0.857	0.625
SVM-RBF ($\mu = 0.1, \gamma = 10^{-3}$)	1	0.458	0.628
ANN	0	1	0
K-MEANS ($k = 2$)	1	0.516	0.681
VS ($\delta = 0.2, n = 3$)	0.935	0.825	0.877
VS ($\delta = 0.1, n = 2$)	0.959	0.791	0.867

of algorithms were evaluated: classifier based and cluster based. For classifier based, Naive Bayes (NB), Support Vector Machines (SVMs) and Artificial Neural Networks (ANNs) are run. For a representation of clustering methods, the simple K-means algorithm is used.

NB is a very simple probabilistic classifier based on the Bayes' theorem. It assumes that all features are completely independent from one another given the class variable Y. Moreover, NB calculates the posterior probabilities $P(X_j|Y = y_i)$ of features X_j given the training data for each class y_i. New data can be classified computing the probability for each class and picking the biggest among them.

SVMs are classification algorithms that aim to find linear decision boundaries which maximize the distance to the closest points from each class. This linear boundary can be extended to a non-linear one with the so called "kernel trick", which uses kernels to compare data points in a different feature space where linear boundaries correspond to non-linear ones in the original feature space [36]. In outlier detection problems, SVMs are used with just one class, in such a way that a boundary is constructed for "normal data" only. When a new point is evalvalued, it is considered an outlier if it falls outside of this created boundary. Different kernels can be used to build this boundary. The kernel we use in these experiments is the radial basis func-

tion (RBF) kernel, which allows SVMs to learn complex regions of feature space [37]. We set the kernel parameters μ and γ experimentally, trying different values and picking the ones that produced the best prediction.

ANNs are graphical models inspired by the functionality of the brain, where nodes are connected to other nodes analogous to how neurons are connected in the brain. Nodes in ANNs combine values from multiple inputs and produce an output by a defined function (such as sigmoid or a step function). This output can be transferred to other nodes in the next layer, or to the output of the network [36]. An algorithm learns a neural network by adjusting the weights of the inputs to the different neurons in the graph so as to best fit the training data. In this work, a multi-layer preceptron (MLP) ANN with one hidden layer is used.

The last algorithm compared is simple K-means. In simple K-means, cluster's centers (the means) are first chosen at random. Each point, then, belongs to the cluster whose mean is the closest to the point. After this step, the means of every cluster are re-calculated and points are re-assigned to the cluster with the closest mean. This procedure is repeated iteratively until the clusters no longer change [38]. In this work K-means is used with two clusters ($k = 2$), and evaluated with the *classes to clusters* evaluation. Under this evaluation, the algorithm first creates clusters with unlabeled data, and then each cluster is labeled with one of the classes based on the greater number of instances of the class within each cluster. Once clusters are labeled, we can compute metrics by counting how many instances of each class fall in each different cluster.

The results for these experiments are shown in Table 2.3. For comparison purposes, VS is also shown for two different combinations of parameters n and δ. These combinations correspond to the best achieved accuracy from each value of n. These results indicate that VS outperforms all of the other algorithms by having the

highest S-measure values. Although ($\delta = 0.2, n = 3$) is the winner given its very good specificity, in the end it all boils down to how much overfitting a system is willing to tolerate in order to have a little bit of extra sensitivity, or a substantial reduction in execution time.

One interesting discovery is that both SVM and K-means are able to effectively discover all faults in the data ($sensitivity = 1$). In the case of K-means, this is an indication that faults do cluster together in the feature space (since they all fall in the same cluster), and that a pattern can actually be learned. Both cases, however, produce too much overfitting. SVM leaves too many non-faults outside the boundary while, K-means, which tends to create clusters of approximately the same size, produces too many false positives. This is due to the fact that the number of instances of one class is much larger than the number of instances of the other. Nevertheless, both approaches can be used under a sceneraio where false positives do not produce excess overhead.

Finally, it is possible to see that both NB and ANN are not appropriate choices. In the case of NB, sensitivity is too low to be considered a viable option. As for ANN, the algorithm is unable to detect any hardware faults at all (it classifies all data as non-fault).

2.6 Evaluation for 2014

In this section, a sensitivity analysis is done to test the effectiveness of our VS-based algorithm to forecast failures in Blue Gene/Q peta-scale systems during the year 2014, as well as a performance evaluation showing the runtime of our VS implementation in PULSE.

2.6.1 Sensitivity Analysis. The initial PBDA parameters are chosen heuristically given observed properties of the data to values we believe should work well. As it

has already been discussed in Section 2.2.1, sensor data in Blue Gene/Q is gathered with a given periodicity of anywhere between 1 and 30 minutes. Because of this, the evidence window W_e is never set to be less than 30 minutes, which ensures that we have at least one reading per sensor. Likewise, and in order to avoid losing any sensor readings, the size of the time step T_s is never set to be more than 30 minutes. The value chosen for the lead time T_l represents a trade off between prediction accuracy and available time to perform FT actions, i.e., the larger the value of T_l is, the more time the system has for performing FT actions but also the harder it becomes to perdict faults. W_p, or prediction window, should always be bigger than T_s in order to avoid the creation of "time holes" with no predictions. Moreover, setting a large value for W_p can produce a significant amount of resource waste, specially in the case when we have a false alarm (since the component needs to be put in quarantine). In addition, intuition tells that it is unlikely that a bigger proportion of faults can be predicted by enlarging W_p, as faults should become harder to predict the further away in the future they are.

Given this, W_e is set to 30 minutes initially. If there are more than one reading from the same sensor during the window W_e, the value used is the average. T_s is set to always be $T_s = 15$ minutes (e.g., with an evidence window of 30 minutes, half of the readings from one time step are re-used in the next). Regarding the lead time T_l, no minimum or maximum amount is required. The value of T_l will depend on how much time is needed by the system to perform FT actions before the predicted failure hits. In order to test the sensitivity of prediction for different values of T_l, a very small value of 5 seconds is chosen initially. Finally, the value of the prediction window is initially set to $W_p = 30$ minutes.

For each of the experiments in this analysis we have a different combination of PBDA parameters, and for each of these combinations we run our GA optimization

to find the best VS parameters (i.e., number of dimensions n and cell side δ). The results of these optimizations are presented in Table 2.3.

In the first set of experiments, shown in Figure 2.8, the Lead Time (T_l) is varied from 5 seconds all the way to 10 minutes (the other parameters are set to $T_s = 15$ minutes, $W_e = 30$ minutes, and $W_p = 30$ minutes). There are three curves in the plot representing recall, precision and f-score. The last is the armonic mean of the first two (see Equation (2.1)). As it is possible to see, results stay fairly stable for values of T_l between 5 secods and 1 minute. The f-score drops a little bit due to an small (almost imperceptible) drop in recall. After one minute, however, there is a large drop in recall from values close to 0.65 for $T_l = 1$ minute, to around to 0.55 for $T_l = 2$ minutes. This shows how it becomes harder to predict failures as we try to do it further away in the future. Recall starts to drop significantly again after 5 minutes all the way to 0.45 for $T_l = 10$ minutes. Precision stays above 0.8 for lead times of less than 1 minute. This indicates that the model is very robust against false positives. After 1 minute, however, precision drops significantly and stays sligthly above recall all the way to 10 minutes.

An interesting observation from these experiments is the fact that there is not significant differences – in terms of predicion accuracy – between a lead time of 5 seconds and a lead time of 1 minute. This gives the system a "free" minute where FT actions can be performed without incurring in major pernalies in detection accuracy. Modern Solid-State Drives (SSDs), for example, have bandwidths of up to 10 GB/s. One minute gives the system plenty of room to perform checkpointing to local storage of processes from a large number of applications. Likewise, state-of-the-art fast network interconnections used in current supercomputers (such as InfiniBand) have bandwidths well above 1 GB/s, making it possible to transfer large chunks of memory to remote locations.

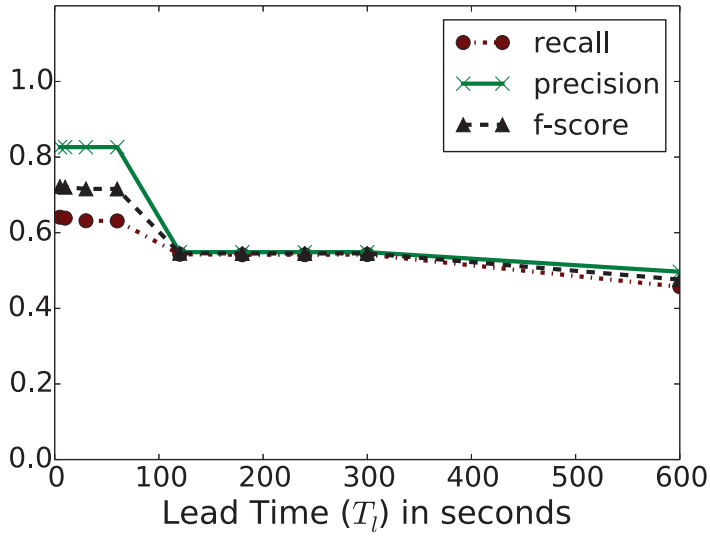


Figure 2.8. Sensitivity of the VS-based algorithm to lead time (T_l) in seconds

The idea of using local SSDs for reducing checkpointing overhead has already been explored successfully in the FT community in the form of multilevel checkpointing [6], where frequent and fast checkpoints to local SSDs are combined with less frequent and slower checkpoints to remote SSDs and/or to remote parallel file systems. In [6], for example, authors show that 10 GB of process memory can be locally checkpointed in less than 30 seconds (using technology of 2011). In any case, it is always possible to go beyond one minute if that is not enough. After one minute, prediction accuracy drops significantly until $T_l = 2$ minutes where, again, stays stable until $T_l = 5$ minutes. This gives the system a "second level" for FT operations that can be performed in less than five minutes but not in less than one.

In the second set of experiments, presented in Figure 2.9, the evidence window (W_e) is varied while maintaining the other parameters to the initial values of $T_s = 15$ minutes, $T_l = 5$ seconds and $W_p = 30$ minutes. Again, there are three curves in the plot representing recall, precision and f-score. As it is possible to see, increasing the evidence window beyond 30 minutes seems to always have a negative impact in

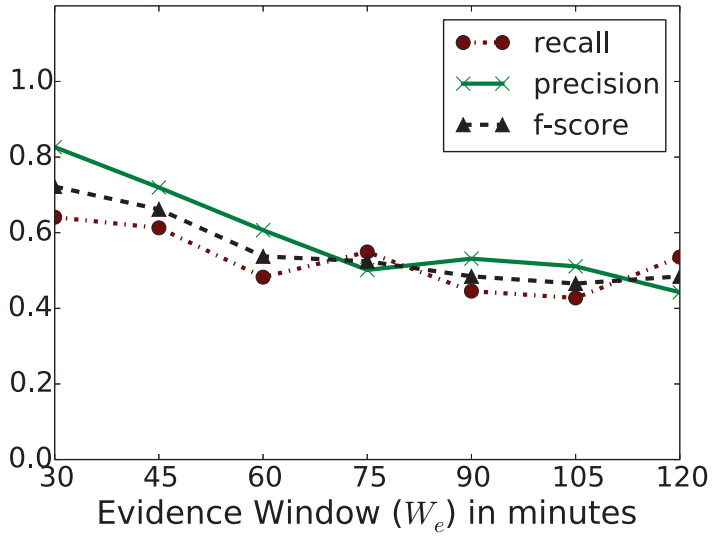


Figure 2.9. Sensitivity of the VS-based algorithm to lead time (W_e) in minutes

prediction accuracy, specially with respect to the number of false positives, which keeps increasing as W_e becomes larger (since precision drops faster than recall). This is a clear indication that the model suffers from overfitting. By using larger evidence windows, training points that are more similar to each other are created (and hence larger voids are created too). This effect is known as regression toward the mean. As it is described above, the average is taken (also known as arithmetic mean) of all the readings for each particular sensor during a window. The more readings (samples) one take out of a particular polpulation, the more similar the average of those readings is to the actual population average. And the more similar each and every average is to the population average, the more similar they are to each other.

Surprisingly, recall does not suffer as much in this case. Although the general trend is clearly downwards, it has some upward points such as in $W_e = 75$ minutes and in $W_e = 120$ minutes.

Finally, in Figure 2.10 the third set of experiments are presented where the prediction window (W_p) is varied while the other parameters are maintained to the

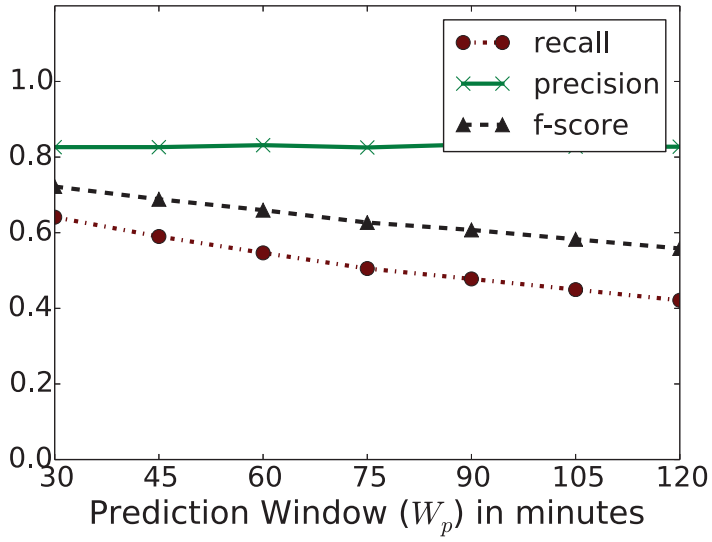


Figure 2.10. Sensitivity of the VS-based algorithm to lead time (W_p) in minutes

initial values of $T_s = 15$ minutes, $T_l = 5$ seconds and $W_e = 30$ minutes. As expected, increasing the prediction window beyond 30 minutes only seems to negatively affect detection accuracy (again, because failures become harder to predict the further away they are in the future). In this case, the number of true positives goes down linearly, affecting recall. Precision, however, is not affected at all and stays stable between 0.81 and 0.83. As shown in Equation (2.3), the only way precision can stay constant while the number of true positives drops is if the number of false positives also drops (and in the same proportion). This is in fact what happens, where some of the false positives seen for smaller values of W_p are just perdictions made *too early*.

2.6.2 Performance Evaluation. The run time of our C implementation of VS is evaluated in both the learning and prediction phases of the algorithm. Figure 2.11 presents the distribution of run times for the learning phase (void construction) of the model for a single midplane using 8 months of data and running on one core on a common CPU (Intel Xeon ES-1650 v3 at 3.50GHz). This run time includes all steps of the learning phase: (1) Querying the data from the database (we store all this data

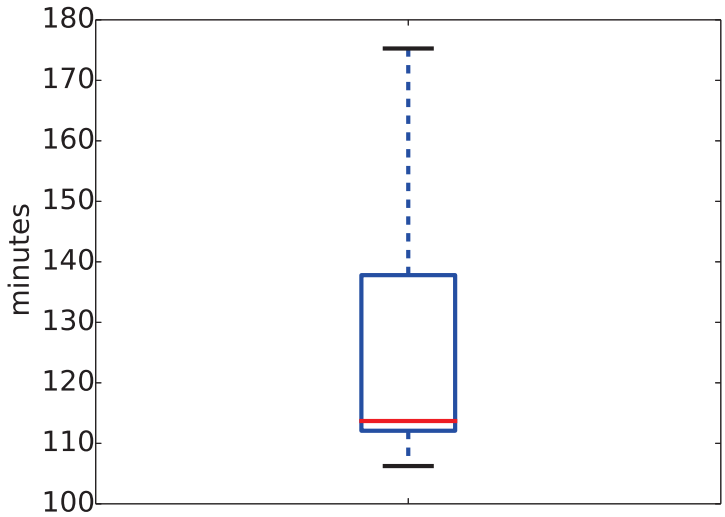


Figure 2.11. Distribution of training time for one midplane running on one code in an Intel Xeon CPU ES-1650 v3 at 3.50GHz. Time includes: (1) Data querying and aggregation, (2) data transformation using PCA, and (3) void (model) creation

locally on the same server in a MySQL database), (2) transforming the data using PCA, and (3) creating the actual voids (as explained in Algorithm 2).

As it is possible to see, this is an expensive operation. The average runtime is a little over 2 hours (close to 127 minutes). The major contributor to this large runtime is the actual querying of the data from MySQL (around 100 minutes out of the 127), which also involves data aggregation, i.e., the output of the querying is a matrix where every column stores the aggregated value for each sensor (using the PBDA parameters) and each row stores the moment in time where we make a prediction. If, for example, the evidence window is $W_e = 30$ minutes, the stored sensors' readings corresponding to a 30 minutes chunk need to be queried and combined into a single average for each sensor. These averages, then, represent the features, which are used to create a row vector.

A fast back-of-the-envelope calculation tells us that if we multiply this runtime by 96 midplanes, we need almost 200 hours (8 days) to train the final model. This

is, of course, way too slow to be practical. The good news is that the model for each midplane can be learned independently and then combined at the end in an "all-gather" fashion (see Section 2.4.2). Given this, a single core in each midplane can be dedicated to compute that particular midplane's model. Other option is to have a separate parallel cluster with at least one core per midplane and compute the model there. Considering that PULSE may only need to be trained once every 4 to 6 months, 2 hours is arguably reasonable.

There are, of course, some optimizations that can be done to speed up void construction if it is needed to iterate multiple times over the same data (as it happens when the VS parameters need to be optimized using a GA). In this case, one can simply query and aggregate the data once using the desired PBDA parameters and store the resulted matrix in a new database table (or file) which is much faster to read than the original database (since it is possible to read it sequentially and there are not any query constrains such as time or location). This intermediate matrix can then be re-used multiple times.

With respect to the prediction phase, runtimes are measured in hundreds of microseconds per midplane. Given this, it is possible to either have each midplane compute its prediction in parallel on a spare system-dedicated core, or gather the information in a centralized process (much in the same way as to how the MMCS operates in order to store the data in the centralized DB2 database) and do the predictions for all the midplanes there. Either way, predictions for a whole petascale supercomputer can be done in less than a second.

2.7 Discussion

One goal of this work, apart from exploring VS algorithms, was to show the great potential of environmental logs, as opposed to RAS logs, for fault detection.

There seems to be three clear advantages. First, by using environmental logs, it is easier to create a decentralized design where localizing failures is straightforward. Learning complex relationships between different events' types, or combining algorithms with topology information is not needed in order to isolate a particular fault. The second reason is the possibility of predicting failures in advance with a minimum lead time, during which the system can perform FT actions safely. The third reason is the results achieved using environmental data, specially with respect to recall. The state-of-the-art in RAS based failure prediction provides a recall of about 0.5 [39] (and without accounting for lead time), while environmental logs seems to achieve well above 0.6.

Given the prediction results obtained in these experiments, VS seems to be an ideal ML algorithm for outlier detection using environmental data in petascale systems. Nevertheless, a further analysis was conducted in order to understand what kind of failures PULSE was predicting and when. This analysis showed that we jumped to conclusions too early.

Taking a closer look at the data, something odd seemed to be going on: The majority of failures happened in the months of May and September (see Figure 2.12). Moreover, the number of failures concentrated in May and September is even bigger that it seems at first, because our analysis considers a failure when there is any (it can be more than one) FAIL event during the prediction window W_p, i.e., a failure is annotated whether there is one FAIL event or hundreds of them. This unusually high number of FAIL events required further inquiry to the administrators of Mira. According to them, the problem was due to some power issues that the supercomputer suffered in two occasions that year. For instance, in September a power problem on one of the building's power feeds caused the system to fail over to the redundant feed as it was supposed to, but the control system for the cooling water loop pumps

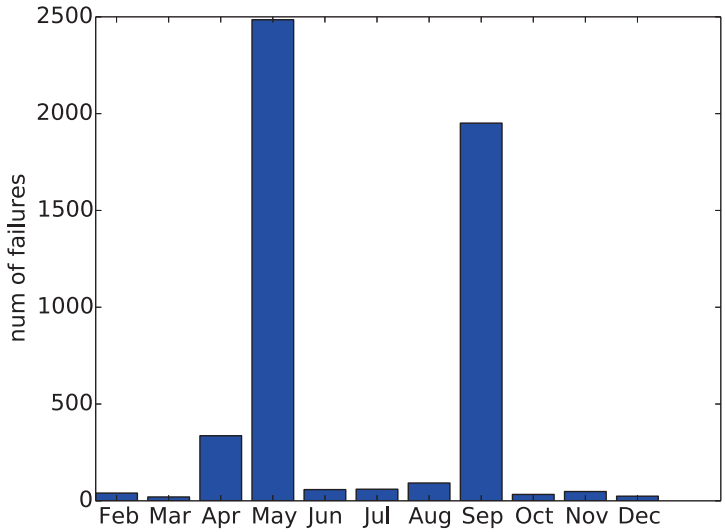


Figure 2.12. Total number of failures by month

had been misconfigured at some point and they did not ride through the power outage properly. The water ceased to flow through the racks, which caused them to automatically shut down.

After this discovery, another important question was inmediately raised: Which failures is PULSE actually predicting? every kind or just power failures affecting the whole system? To answer this question, another experiment was performed where testing was done on each month with training of the model done on the previous months. For example, to test on February, the model was trained on January. To test on March, the model was trained on January and February and so on. Failures were then divided into *category-componet* pairs. For example, a category can be *Message_Unit* and a component *FIRMWARE*. Of all the possible failures, PULSE was able to predict with high accuracy only the following four: *AC_TO_DC_PWR-MMCS*, *Coolant_Monitor-MMCS*, *Node_Board-MC*, and *AC_TO_DC_PWR-BAREMETAL*. Interestingly, those types of failures were only present on May, September, or both, and concentrated on the days of the power outages, which clearly answers the question as

to whether PULSE is only predicting power failures affecting the whole system or all failures in general.

Finally, one last test would be to go back to 2012 data and check whether this is also happening there. A lack of time made this impossible. Nevertheless, 2012 was not a production year (Mira entered production on January 2013), and so it is the oppinion of this author that 2012 was not as relevant or interesting.

2.8 Related Work

Failure prediction on HPC systems is broadly approached from two directions. One is to analyze system logs composed of critical events or other textual events generated by various health monitors, such as RAS events. The other (to which this work belongs) is to detect when a fault occurs. By detecting the faults that are likely to lead to failures, effective fault detection leads to failure prediction.

Numerous works have been devoted to the understanding of failures in HPC systems based on log information. For example, Sahoo et al. [12] explored three classes of algorithms in a 350-node IBM cluster: time-series, rule-based, and Bayesian networks; Zheng et al. [13] also used a GA to learn rules that can predict failure times as well as failure locations; in [14], rule-based, SVMs and Nearest Neighbor based classifiers are explored for failure prediction on a Blue Gene/L system; Lan et al. [11] proposed a dynamic meta-learning prediction engine to combine the benefit of multiple algorithms to boost accuracy; and Gainaru et al. combines signal-processing concepts and data-mining techniques to predict fault occurrences [39]. A detailed survey of failure prediction methods can be found in [40]. There has also been increasing work on failure analysis in the past years. These studies typically extract statistical information about failure distribution and use this information to prepare for potential failures [41, 42, 43, 44].

Fault detection in HPC has also been explored in the literature. Existing detection techniques include classification based, clustering based, statistical modeling, information theoretic techniques (e.g., entropy), and spectral techniques [37]. Prior work on fault detection on HPC and other enterprise-scale systems has primarily focused on collecting various types of system metrics (e.g., CPU utilization, cache miss rate, packet rate, or file system operation rate) and using statistical tools to detect anomalous values for these metrics. For applications that place a consistent load on all the nodes of a system (e.g., SPMD applications or replicated services), the typical approach assumes that during failure-free operation the metrics collected from all the nodes should be very similar (i.e., peer similarity) [45, 46, 47]. It then detects failures by looking for nodes where observations vary significantly compared to observations on other nodes. Since workloads may vary over time, the model of normal behavior can be adapted to give more importance to more recent observations. For applications that utilize different system nodes differently, an alternative approach has been explored that focused on individual application code regions and collects system metrics for each region [48]. Although similar in spirit, this work is different from these studies in that we use hardware sensor readings, an important but often overlooked data source for detection. Furthermore, a new a new algorithm is proposed for fault detection (Void Search) that, to the best of our knowledge, has not yet been explored for fault detection in the literature.

In previous work [49], online data reduction techniques were presented (instance and feature selection) to remove redundant and noisy data in environmental logs. Under that approach, all environmental data available from the whole system is gathered in a centralized buffer and analyzed together. The problem with this approach, however, is that it is not as scalable as the number of hardware sensors (and hence features in the data) grows linearly with the size of the machine. For that reason, in this work a decentralized path is followed, where environmental logs get

analyzed in a component per component basis.

Table 2.3. Optimization using our GA of the VS parameters (n and δ) given different values for the PBDA parameters

T_s	W_e	T_l	W_p	n	δ
15 mins	**30 mins**	**5 secs**	**30 mins**	**6**	**6.8852**
15 mins	30 mins	**10 secs**	30 mins	**6**	**6.8852**
15 mins	30 mins	**30 secs**	30 mins	**6**	**6.8852**
15 mins	30 mins	**1 min**	30 mins	**6**	**6.8852**
15 mins	30 mins	**2 mins**	30 mins	**6**	**4.2383**
15 mins	30 mins	**3 mins**	30 mins	**6**	**4.2383**
15 mins	30 mins	**4 mins**	30 mins	**6**	**4.2383**
15 mins	30 mins	**5 mins**	30 mins	**6**	**4.2383**
15 mins	30 mins	**10 mins**	30 mins	**6**	**4.2383**
15 mins	**45 mins**	5 secs	30 mins	**5**	**5.0801**
15 mins	**60 mins**	5 secs	30 mins	**6**	**6.8863**
15 mins	**75 mins**	5 secs	30 mins	**6**	**4.8720**
15 mins	**90 mins**	5 secs	30 mins	**5**	**4.5212**
15 mins	**105 mins**	5 secs	30 mins	**6**	**6.5157**
15 mins	**120 mins**	5 secs	30 mins	**6**	**4.3408**
15 mins	30 mins	5 secs	**45 mins**	**6**	**6.8852**
15 mins	30 mins	5 secs	**60 mins**	**6**	**6.8852**
15 mins	30 mins	5 secs	**75 mins**	**6**	**6.8852**
15 mins	30 mins	5 secs	**90 mins**	**6**	**6.8852**
15 mins	30 mins	5 secs	**105 mins**	**6**	**6.8852**
15 mins	30 mins	5 secs	**120 mins**	**6**	**6.8852**

CHAPTER 3

DATA-ANALYTIC-BASED SILENT DATA CORRUPTION DETECTION

3.1 Introduction

High-performance computing (HPC) is changing the way scientists make discoveries. Science applications require ever-larger machines to solve problems with higher accuracy. While future systems promise to provide the power needed to tackle those science problems, they are also raising new challenges. For example, transistor size and energy consumption of future systems must be significantly reduced. Such steps might dramatically impact the soft error rate (SER) according to recent studies [15, 50].

Random memory access (RAM) devices have been intensively protected against soft errors through error-correcting codes (ECCs) because they have the largest share of the susceptible surface on high-end computers. Recent studies, however, indicate that ECCs alone cannot correct an important number of DRAM errors [51]. In addition, not all parts of the system are ECC-protected: in particular, logic units and registers inside the processing units are usually not protected by ECC but by other methods because of the space, time, and energy cost that ECC requires in order to work at low level. Historically, the SER of central processing units was minimized through a technique called *radiation hardening* [52], which consists of increasing the capacitance of circuit nodes in order to increase the critical charge needed to change the logic level. Unfortunately, this technique involves increasing either the size or the energy consumption of the components, which is prohibitively expensive at extreme scale. Thus, a non-negligible ratio of soft errors could pass undetected by the hardware, corrupting the numerical data of HPC applications. This is called silent data corruption (SDC).

Runtime data analysis can be used to leverage the fact that some datasets produced by iterative HPC applications (i.e., the applications' state at a particular point in time) evolve *smoothly* from one time step to the next. This characteristic can be used effectively to design a general SDC detection scheme with relatively low overhead. In particular, we will show that an interval of *normal* values for the evolution of the datasets can be predicted, such that any corruption will *push* the corrupted data point outside the expected interval of *normal* values, and it will, therefore, become an *outlier*.

In previous work [53, 54], the feasibility of using data analysis with simple and lightweight linear predictors to detect SDC efficiently was showed.

In this work, we extend upon this previous work by providing new critical insights on how to use this approach effectively on real applications and produdction level input data sets:

- The characteristics of iterative HPC data are explored. On one hand, we verify the smoothness of this data over time. On the other, we study the propagation of corruption on one particular application, including the transfer to other processes, to show the potential impact of SDCs.
- A new detection model based on the user-expected accuracy and past prediction errors is introduced, and we theoretically analyze different prediction cases in order to calculate optimal parameters for this model.
- A comprehensive evaluation using all our detectors with a number of popular iterative HPC applications and kernels is performed, which cover a wide spectrum of scientific use cases in HPC. It is showed that our detectors can guarantee over 90% of SDC coverage on real application runs in many cases.

- An in-depth analysis of the performance and memory overheads incurred by the proposed detectors and the trade-off between detection cost and accuracy is provided.

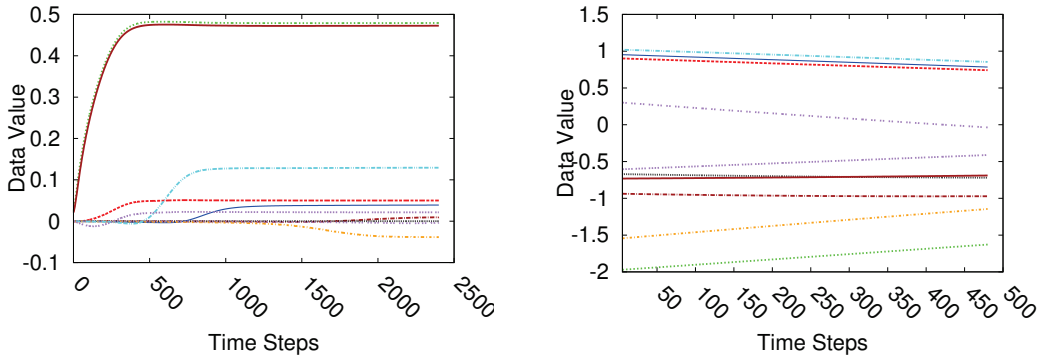
We also observe that it is viable to detect, using the underlying physics of the simulation, corruptions in one dataset by just using another, thus opening the door for memory saving strategies.

The rest of the chapter is organized as follows. In Section 3.2, real-world HPC production traces are characterized in order to deeply understand and explore the characteristics of HPC data. In Section 3.3 the proposed detectors are presented. In Section 3.4 the analytical model with optimized detection range sizes is introduced. In Section 3.5 the evaluation and results are presented. Section 3.6 shows related work. Finally, in Section 3.7 a discussion of the results and the viability of our approach is provided.

3.2 Understanding HPC Data

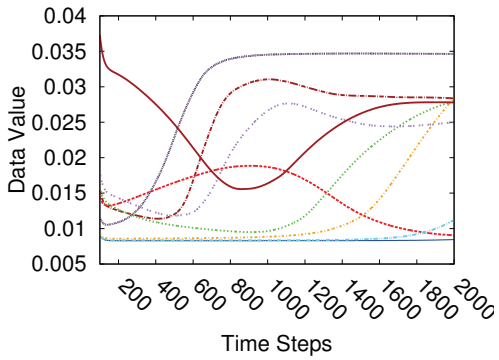
Understanding the properties of production iterative HPC data is key to design effective SDC detection methods. In this section, the characterization of production HPC data traces is presented, based on which one can observe both how data evolve *smoothly* from one time step to the next, as well as how corruption propagation affects applications' results depending upon different bit-flip errors.

3.2.1 The Smoothness of HPC Data. To illustrate the property of *smoothness*, consider Figure 3.1, where two applications (DoE applications Nek5000 and FLASH. For more information regarding these applications see Section 3.5), and four data sets are shown. In each plot, 10 randomly selected points for a number of time steps are shown depending on each application case. As it is possible to see, the data evolution of these points is *smooth*, since it does not change sharply from one time step to the

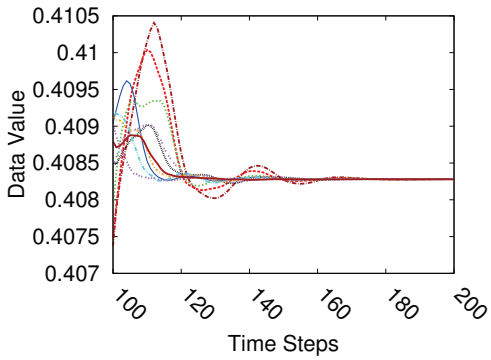


(a) Nek5000-Vortex

(b) Nek5000-Eddy



(c) FLASH-Sedov



(d) FLASH-Sod

Figure 3.1. Smoothness of numerical simulations from real-world HPC application datasets

next.

Although not all numerical simulations may have the property of smooth dataset evolution, many applications do exhibit such a feature, allowing us to design low-cost, effective SDC detectors.

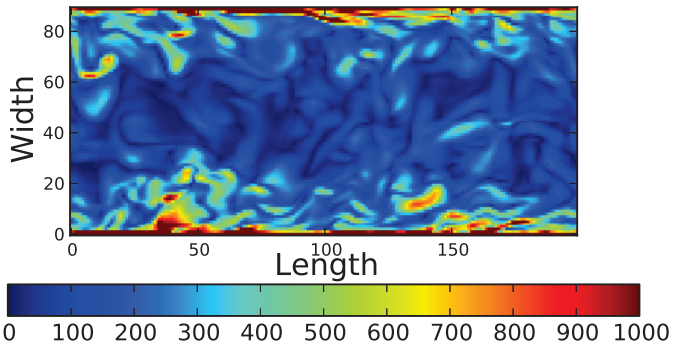
3.2.2 Corruption Propagation. In this Section, the way corruption propagates is analyzed in classic iterative HPC applications , such as the Turbulence-CFD code[5]. CFD applications produce vorticity plots to show the turbulence of the fluid. Figure 3.2a, for instance, shows the vorticity of the fluid on a 2D cut out of the 3D duct.

[5]Turbulence-CFD is a computational fluid dynamics (CFD) miniapp. See Section 3.5 for more details.

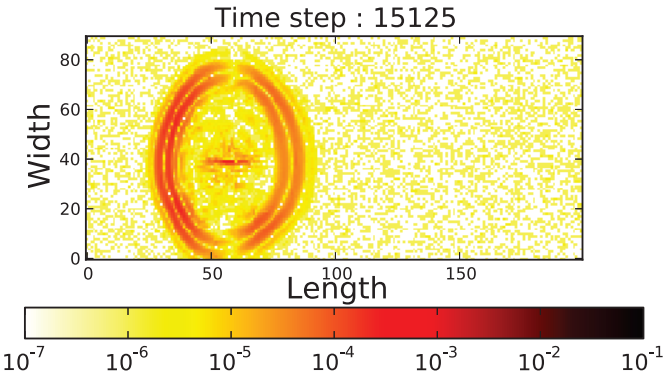
The vorticity, however, is computed from the velocity fields in the three axes and is never stored in a variable. Therefore, an error deviation observed in the vorticity plot is likely to be the consequence of a corruption in one of the velocity fields. In this run, a bit-flip (grid point $40X40$) is injected in the 24th bit position, the first bit of the exponent. This corruption is barely visible to the naked eye if plotted on a figure. Yet it will generate large perturbations that will propagate across the domain, reaching other MPI ranks and corrupting the large majority of the domain.

To study the propagation of corruption after an SDC, the following experiment was performed. First, a turbulent flow simulation starting from the initial conditions was launched and let run for 15,000 time steps such that the gas reaches a high level of turbulence. Then, the execution is restarted from time step 15,000 using FTI and the datasets of the execution at each time step recorded for a corruption-free execution. Several repeated experiments confirmed that several corruption-free execution produce identical results. Then, the same experiment was repeated but this time injecting one bit-flip at bit position $2p$ for p in [1,12]. For each experiment a bit-flip was injected in the first twenty time steps and let it run for 500 iterations.

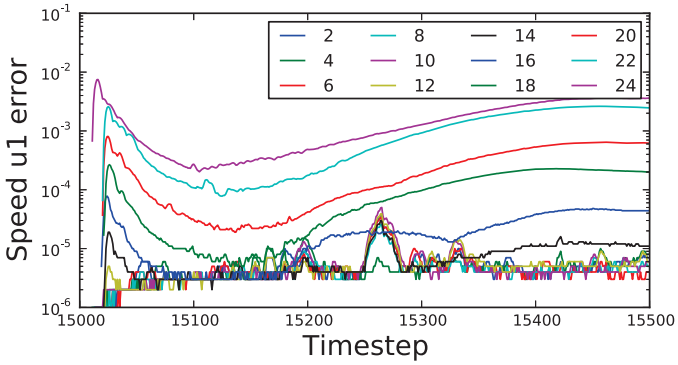
After all the corruption experiments were done, the difference between the corrupted dataset and the corruption-free dataset was computed for each experiment and for each time step . Figure 3.2b plots this deviation a hundred time steps after corrupting the 24th bit of the grid point $40X40$. A logarithmic color scale is used to show the magnitude of the deviation in the different regions of the domain. As one can observe, in only a hundred iterations the corruption has already propagated across the entire domain, and it shows a particularly high deviation in a region with a wave shape that has as origin the corrupted grid point. Although the origin of the corruption was in the grid point $40X40$, the fluid has moved in those hundred time steps, and the corruption wave's epicenter is about 20 grid points to the right, which



(a) Vorticity in turbulent fluid



(b) Error propagation of 24th bit corruption



(c) Maximum deviation after corruption

Figure 3.2. Error propagation in turbulent flow simulation

happens to be located in an MPI rank other than the one where the corruption was originally injected.

Similar figures for every time step of each corruption experiment were plotted, but here, only one is shown for brevity. One should note that the high deviation wave bounces in the walls of the duct and continues propagating in other directions. To get an idea of how deviations behave for different corruption levels, the maximum deviation at each time step was plotted for all the corruption experiments. As it is possible to see in Figure 3.2c, during the first ten time steps or so, no corrupted data occurs. When the bit-flip is injected, however, we observe a sudden jump with a magnitude exponentially proportional to the bit-flip position, which is consistent with the floating-point representation. In addition, immediately after the corruption jump, the deviation starts to decrease. This decrease is due to a *smoothing* effect that takes place when the noncorrupted data interacts with the corrupted data. However, the same influence can go in the other direction. For instance, when the corruption wave bounces in the wall of the duct, it interacts with the other part of the wave that is just arriving at the wall, generating a corruption amplification effect, which is what one can see happening after time step 15,150. Finally, the deviation stabilizes around time step 15,400 and remains stable until the end of the execution.

3.3 Anomaly Detection

In this work, the run-time data computed by iterative HPC applications to detect SDCs is used. This idea originates from our observation that the data sets of some real iterative HPC applications change *smoothly* over time (and/or space), allowing the opportunity to use data analytics to predict the next possible values of the data.

The in-depth analysis of the compute data at runtime also has some special

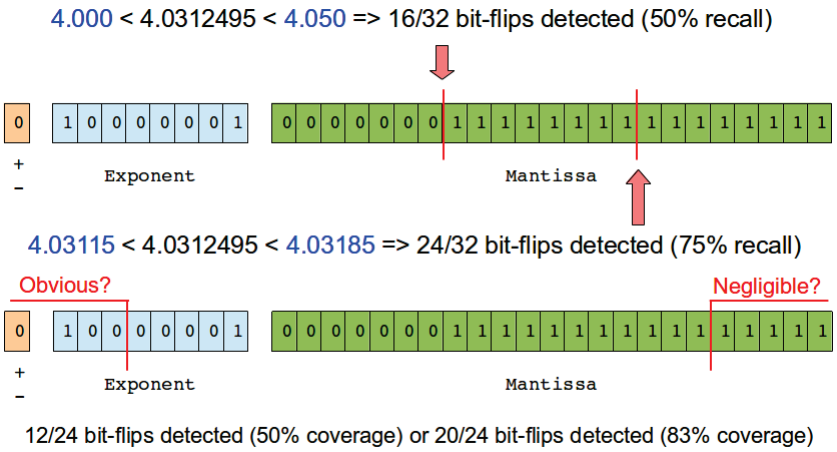


Figure 3.3. Anomaly detection based on prediction

advantages. First, such an approach is completely independent of the underlying algorithm[6] and therefore dramatically more general than algorithm-based techniques. Second, one can develop lightweight data-monitoring techniques that impose a low overhead on the application compared with that from extremely expensive techniques such as double and triple redundancy. And third, data monitoring and outlier detection can be offered at runtime in a fashion transparent to the user.

Anomaly detection has been used in multiple domains such as medical analysis.

The main idea is to monitor the application datasets during runtime in order to predict an interval of *coherent* values for the next time step and then raise alerts when some data points go outside this range (*outliers*). For instance, consider a dataset that is evolving during execution and is being monitored by this detector. The detector analyzes the dataset at each time step and predicts an expected range of values for the next time step. As shown in Figure 3.3, the detector predicts that normal values should be inside the segment $[4.000, 4.050]$. A data point with the value 4.0312495 is inside that range. If that data point were corrupted in one of the 16 most significant

[6]In the sense that no modification is needed.

bits of the IEEE floating-point single-precision (32 bits) representation, the value would automatically move outside the expected range of normal values and would be detected as an outlier. Now, let us imagine that the detector could give a more accurate prediction, giving $[4.03115, 4.03185]$ as the interval of normal values. In this case, any corruption in the 24 most significant bits would push the point outside the new and narrower interval.

The efficacy of this type of detectors is evaluated using two well-known metrics: precision (denoted in this chapter as ρ) and recall (denoted in this chapter as τ), defined in Equations (2.3) and (2.2), respectively. The definitions of precision and recall are reproduced here to help the reader. Here *TP*, *FP*, and *FN* refer to *true positives*, *false positives*, and *false negatives*, respectively. As shown, the accuracy of the prediction has a direct impact on the detection recall.

$$\rho = \frac{TP}{TP + FP}$$

$$\tau = \frac{TP}{TP + FN}$$

Another important point is that not all the bits in the IEEE floating-point representation need to be covered in all cases. For instance, a corruption in the most significant bits is likely to generate numerical instability, inducing the application to crash. Such soft errors might be silent to the hardware but not to the application. On the other hand, corruption in the least significant bits of the mantissa might produce deviations that are lower than the allowed error of the application and, hence, are negligible. With the second detector discussed above, if the 4 most-significant bits (numerical instability) and the 4 least-significant bits (negligible error) are neglected, it is possible to say that the detector has a coverage of 20 bits out of 24 (83% coverage).

In the following subsections, two types of anomaly detection models are introduced: pointwise detection models and cluster-based detection models. The former constructs the normal value range based on a one-step-ahead prediction and an optimized range size at each time step; the latter analyzes the data distribution at each time step to detect the silent data corruption at runtime. Both models are suitable for different detection demands and/or applications.

3.3.1 Pointwise Anomaly Detection Model. The pointwise SDC detection model has two phases: the first phase involves the prediction of the next expected value in the time series for each data point and the second phase determines a range (i.e., normal value interval) surrounding the predicted next-step value. Soft errors can be detected by observing whether a particular value falls outside this computed range. With this approach only one-step ahead prediction using recent data values needs to be performed in order to achieve high accuracy, thanks to the smoothness property discussed above. The range size, or normal value interval, will play an important role in obtaining high precision and recall. Its magnitude depends on the relative location of the predicted value and the user-expected accuracy. In general, different HPC applications have different expected accuracies for their computation results. For example, users may expect the accuracy to be always within 10^{-8}. Hence, for each computed data point, there will be a user-expected (or tolerable accuracy) value interval. In Section 3.4, an analysis about the optimal range size is presented for different predicted values and user-expected accuracies.

The key notations used to formulate the pointwise detection model are presented in Figure 3.4. The range used to detect silent errors is denoted by $[X(t) - \delta, X(t) + \delta]$, where $X(t)$ is the predicted value at time step t. The range size is denoted by $s = 2\delta$. In general, the magnitude of the prediction error is dependent on the prediction method used, as shown in Figure 3.4. In this figure, the real data

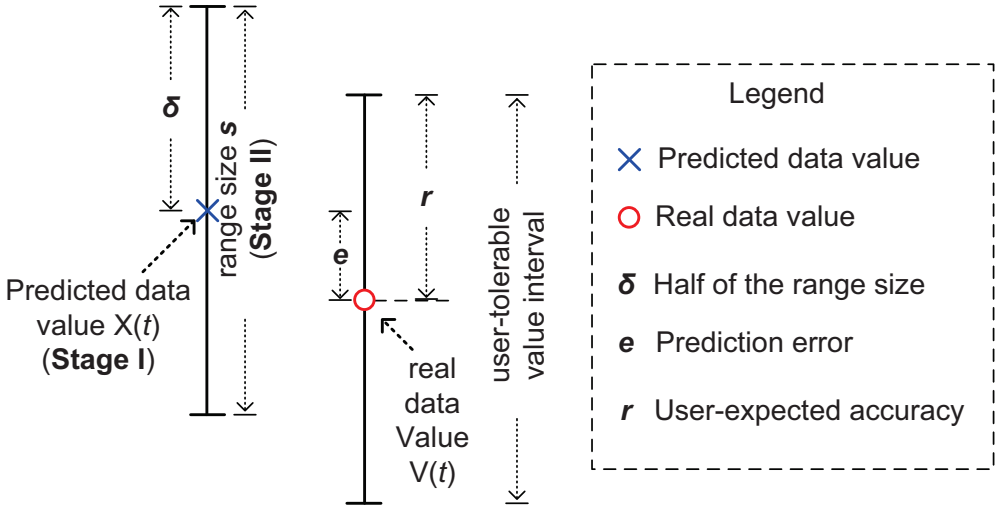


Figure 3.4. Illustration of the one-step prediction model (at time step t)

value is presented as a red circle, and the prediction error (denoted by e) is equal to the difference between the predicted value $X(t)$ and the real data value (denoted by $V(t)$) computed at the current time step. The user-expected (or tolerable) value interval is denoted by $[V(t) - r, V(t) + r]$, as shown in Figure 3.4, where r refers to the user-expected accuracy.

In the following, we introduce a number of different predictors that are the core of our pointwise anomaly detectors. Every predictor involves a specific trade-off between overhead and prediction error.

Linear Curve Fitting

The first predictor, called linear curve fitting (LCF), uses the two most recent previous time steps to fit a linear curve, which is then projected to the next time step in order to predict the next value in the time series. Equation (3.1) shows how this prediction is calculated. $\Delta(t-1)$ is the slope of the curve (velocity) at time $t-1$.

$$\begin{aligned} X(t) &= \Delta(t-1) + V(t-1) \\ &= (V(t-1) - V(t-2)) + V(t-1) \\ &= 2V(t-1) - V(t-2) \end{aligned} \tag{3.1}$$

Acceleration-Based Predictor

The acceleration-based predictor (ABP)[7] uses the two and three most recent previous time steps to extract the velocity $(\Delta(t-1))$ and acceleration $(\Delta^2(t-1))$ of the data, respectively, and then combines them to compute the prediction for the next value in the time series.

By definition we have

$$\begin{aligned} \Delta^2(t-1) &= \Delta(t-1) - \Delta(t-2) \\ \Delta(t-1) &= V(t-1) - V(t-2) \\ \Delta(t-2) &= V(t-2) - V(t-3). \end{aligned}$$

Putting them together results in

$$\begin{aligned} X(t) &= \Delta^2(t-1) + \Delta(t-1) + V(t-1) \\ &= 3V(t-1) - 3V(t-2) + V(t-3). \end{aligned} \tag{3.2}$$

Autoregressive Model

The autoregressive model (AR) assumes that every value in the time series depends linearly on its previous values. Equation (3.3) describes AR, where c is a

[7]It can also be called quadratic curve fitting (QCF).

constant, φ_i are the coefficients of the model, p is the number of coefficients, and $\varepsilon(t)$ is the noise at time t. Without loss of generality, we can assume $\varepsilon(t) \sim \mathcal{N}(0, \sigma^2)$ (normal distribution). The coefficients φ_i are computed by using the first 10 time steps of the simulation by least squares with the Yule-Walker equations. It is assumed that no errors occur during this period; otherwise the coefficients would not reflect the reality of the application's data behavior.

$$X(t) = c + \sum_{i=1}^{p} \varphi_i V(t-i) + \varepsilon(t) \tag{3.3}$$

Clearly, the value chosen for p is critical. A large value of p may give a better prediction, but it will incur a high memory footprint. We note that as opposed to LCF and ABP, which keep only past values V, AR needs to keep both the coefficients φ_i and the past values V in memory, which means a higher memory cost consumed by AR. Based on experience with the evaluation over real-world production datasets, larger values of p may lead to better results, yet with only slight improvement when $p \geq 4$. Accordingly, p is set to four in in this implementation.

Autoregressive Moving Average Model

In the autoregressive moving average ($ARMA$) model, the moving average model and the AR model are combined. In ARMA, there is no need to specify the values of coefficients for both the AR model (p) and the MA model (q). In this implementation, $p = 4$ and $q = 4$ are used. The errors $\varepsilon(t-i)$ are computed by using the past prediction errors.

$$\varepsilon(t-i) = V(t-i) - X(t-i)$$

The complete model is described in Equation (3.4).

$$X(t) = c + \sum_{i=1}^{p} \varphi_i V(t-i) + \varepsilon(t) + \sum_{i=1}^{q} \theta_i \varepsilon(t-i) \tag{3.4}$$

3.3.2 Cluster-Based Anomaly Detection Model. In this section, the idea of time-based prediction is expanded to spatial and spatiotemporal predictors in order to use multiple detectors simultaneously. First an optimization technique to limit the overhead induced by the pointwise prediction methods is introduced. In contrast to the pointwise detectors (see Section 3.3.1), here a scheme is envisioned based on predicting the evolution of a *cluster* of nearby points. That is, all the data will be classified into a number of clusters based on vicinity. The features of each cluster, such as the probability density function (PDF), will be extracted, monitored, and used for prediction. Such a design will be inexpensive because one does not need to monitor each data point independently but treat only the cluster features instead. γ is used to denote a set of cluster features; each element in γ is called a *feature point*. Any feature point outside the expected distribution will be considered an outlier.

Dataset Distribution

The first cluster-based detector proposed estimates the normal value interval for the target dataset γ that the next-step data will fall inside with highest probability. Based on the PDF at time step $(t-1)$ and the evolution of the PDFs at previous time steps (such as $t-2$), this detector predicts the possible PDF for the detection time step (t). Any observed data point located outside the boundaries of the predicted PDF will be treated as an outlier. This detector is called a $\gamma - detector$.

Spatial Anomaly Detector

The second detector proposed analyzes the space variations of the dataset γ. It computes at each time step $t-1$ a distribution for β (beta), where β is computed as shown in Equation (3.5) for a 2D domain.

$$\beta_{(i,j)}(t) = \gamma_{(i,j)}(t) - \gamma_{(g,h)}(t)$$
$$\text{where} g, h \in \{i-1; i; i+1\} \tag{3.5}$$

This computation can take into account the neighbors in one or more dimensions, as well as many neighbor points (e.g., 5-point stencil) depending on the preferences of the user. Then, the detector produces and stores the PDF of the β for the last time step $t-1$ and predicts a PDF for the next detection time step t. As in the previous case, any feature point indicating that $\beta_{(i,j)}(t)$ is outside the expected range of normal values is treated as an outlier. This detector is called a $\beta - detector$.

Temporal Anomaly Detector

The third type of detector developed is based on the temporal evolution of a dataset. For each feature point, the difference ϵ is computed as shown in Equation (3.6).

$$\epsilon_{(i,j)}(t) = \gamma_{(i,j)}(t) - \gamma_{(i,j)}(t-1) \tag{3.6}$$

Then, as with other detectors, the distribution of ϵ at time step $t-1$ is computed and the observed values at next time step t are checked. Any $\epsilon_{(i,j)}(t)$ that violates the predicted PDF is treated as an outlier. Computing $\epsilon(t)$, however, requires saving $\gamma(t-1)$ in memory, thus involving extra memory overhead. To avoid this overhead, the accuracy is sacrificed to a certain extent by leveraging the index of the PDF. Specifically, the domain value range [min_value, max_value] is splitted evenly into 256 bins. Instead of keeping $\gamma_{(i,j)}(t-1)$ for each point in the domain, only a 1-byte word that indexes the value closest to $\gamma_{(i,j)}(t-1)$ from among the 256 bins is kept. In this way, the memory footprint of such detector is reduced from 4 or 8 bytes (for single or double precision, respectively) per data point to only 1 byte.

This detector is called an $\epsilon - detector$.

Spatiotemporal Anomaly Detector

The fourth detector proposed is a spatiotemporal detector that computes the time evolution (denoted by ζ) of the β, as shown in Equation (3.7).

$$\zeta_{(i,j)}(t) = \beta_{(i,j)}(t) - \beta_{(i,j)}(t-1) \tag{3.7}$$

Computing the time gradient of the space gradient gives us an idea of when a dataset increases or decreases its level of *turbulence*. Similar to the temporal anomaly detector, one must keep β values of the previous time step in order to compute the time difference. Thus, the same indexing technique is employed (losing a little accuracy) to reduce the overhead from 4 or 8 bytes per data point to only 1 byte. This detector is called a $\zeta - detector$.

3.4 Analysis of the Detection Cases and Optimization of Range Size

In this section, different prediction cases are theoretically analyzed for calculating the optimal value of the normal data range for the detector, in terms of the prediction error and user expected accuracy.

Based on the experiments with real HPC applications, different range sizes could result in largely different detection results (such as recall) even with the same predictor. Figure 3.5 shows different detection results over the bit-flip errors[8] occurring at different bit positions with the same predictor (ABP). The trace data used in the experiment is from HACC[9] application, particularly regarding particles' position.

[8]Bit-flip errors are injected by a fault injector. The details can be found in Section 3.5.

[9]See Section 3.5 for the description of HACC.

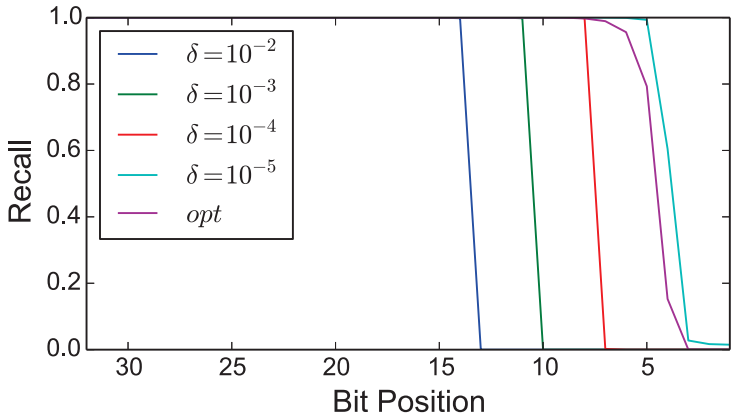


Figure 3.5. Recall for bit-flips injected on HACC's traces using different sizes for the detection range

The detection range sizes are manually set to different values, which increase by an order of magnitude in each curve until the number of false positives (FP) becomes prohibitively expensive (or precision is too low). In this case, using $\delta = 10^{-4}$ would give us a much lower recall than the optimal *opt* computed using the analysis presented in Section 3.4.2. Thus, knowing how to calculate the optimal is critical in order to know "how far" from it the used δ is at a particular moment. Realize that everything to the right of *opt* can not be used due to the drop in precision.

One can argue that it is possible to keep manually searching for a δ that would give us a "good enough" value of recall given the condition that FP = 0. However, this approach becomes infeasible when what we want is to compute a δ that can be used for all execution scales and input values of a particular application.

3.4.1 Analysis of Silent Error Detection Cases. The relative locations of real and predicted values, along with their corresponding detection ranges and user-tolerable value intervals, are categorized into a total of six cases, as shown in Figure 3.6. The notations used in the figure are the same as the ones used in Figure 3.4. In every case, the value space is split into five different parts, each of which corresponds to a particular detection case. In case 1 and case 6, for example, the estimated

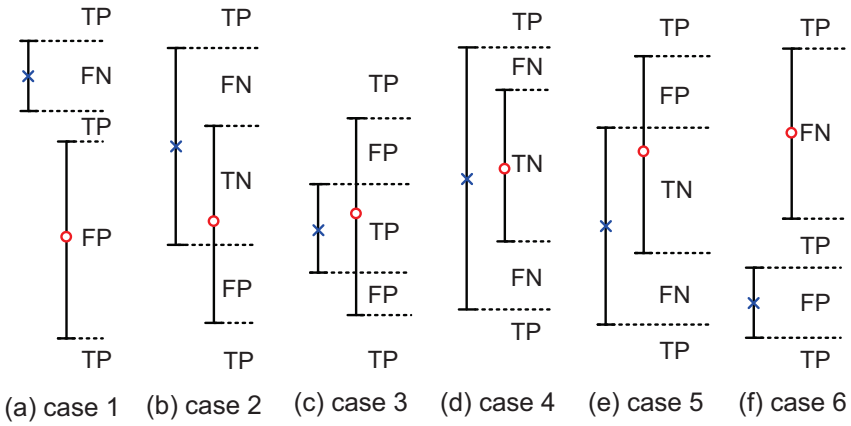


Figure 3.6. Location analysis of detection range vs. user-tolerable interval

detection range stays completely outside the user-expected interval. In case 2 and case 5, the range and the user-expected interval overlap to a certain extent. In case 3, the range completely falls inside the user-expected value interval, resulting in zero false negatives (FN). In case 4, the detection range completely covers the user-expected interval, so there are no any false positive (FP) cases.

3.4.2 Optimizing Range Sizes for Point Wise Detection Model. In this subsection, the range sizes based on different cases are optimized as shown in Figure 3.6. That is, the objective is to optimize the value of δ (half of the range size), given some conditions required by users. $\delta^*_{\{condition\}}$ is used to denote the optimal δ when being subject to a particular condition (e.g., precision $\rho = 100\%$).

Theorem 1. *For a particular predicted value point $X(t)$, the following four propositions hold, where r and e refer to the user-expected accuracy and prediction error respectively.*

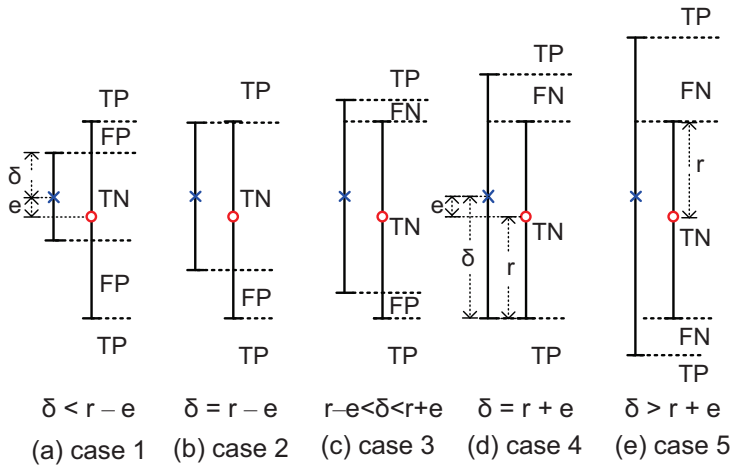


Figure 3.7. The Five situations subject to $e \leq r$

① $\delta^*_{\{\rho=100\%\}} = r + e$

② $\delta^*_{\{e\leq r,\tau=100\%\}} = r - e$

③ $\delta^*_{\{e>r,\tau=100\%\}}$ *does not exist.*

④ $r - e_{max} \leq \delta^*_{\{e\leq r\}} \leq r + e_{max}$

Proof. There are two cases: $e \leq r$ or $e > r$. As for $e \leq r$ (such as Figure 3.6(b)-(e)), the estimated range must overlap the user-expected value interval. Five different situations exist, as shown in Figure 3.7.

Given a particular predicted value $X(t)$ (i.e., r and e are fixed), the detection range may increase as shown in Figure 3.7 (a)-(e). Obviously, $\tau = 100\%$ can occur only in case 1 and case 2 because there are not any FN detections there. That is, for meeting $\tau = 100\%$, $\delta \leq r - e$ must hold. By comparing Figure 3.7(a) and (b), we can see that the number of TP detections is the same while the FP detections in case

2 will be never greater than that of case 1. As a result, case 2 leads to the largest precision under the condition $e \leq r$. In other words, Equation (3.8) (proposition ②) holds.

$$\delta^*_{\{e \leq r, \tau = 100\%\}} = r - e \tag{3.8}$$

Similarly, we can derive Equation (3.9) from Figure 3.7 (d) and (e): $\rho = 100\%$ can occur only in cases 4 and 5 since there are no FP detections there. Hence, $\delta \geq r+e$ must hold to meet $\rho = 100\%$. By comparing Figure 3.7(d) and (e), it is clear that for any scenarios belonging to case 5, there must exist a scenario belonging to case 4 such that the amount of TN detections is smaller than that of case 5. That is, the case 4 leads to the largest recall under the condition $e \leq r$. We can derive Equation(3.10) using a similar analysis for $e > r$.

$$\delta^*_{\{e \leq r, \rho = 100\%\}} = r + e \tag{3.9}$$

$$\delta^*_{\{e > r, \rho = 100\%\}} = r + e \tag{3.10}$$

Combining Equations (3.9) and (3.10), we get proposition ①. To illustrate proposition ③, let's go back to Figure 3.6(a). It is possible to see that, in order to guarantee a τ=100%, we would have to let δ=0; otherwise, $s(=2\delta) \neq 0$, which means that there would be detection cases where FN > 0. However, δ=0 (or $s = 0$) would report all values in all time steps as outliers. In the rollback recovery model, this will lead to no progression, so it is unacceptable in practice.

In the following, proposition ④ is proven. On one hand, it is possible to prove that for any particular prediction error e ($\leq r$), $r - e \leq \delta^* \leq r + e$ must hold. Such a

proposition can be proved by using Figure 3.7, which shows all possible cases where the prediction error e is less than or equal to the user-expected interval r. We can see that precision and recall in case 2 can never be smaller than in case 1. The reason is that FN = 0 holds in both cases but FP is greater in case 1 than in case 2. Hence, $\delta^*_{e\leq r} \geq r - e$ holds. Similarly, we can prove that $\delta^*_{e\leq r} \leq r + e$ based on Figure 3.7 (d) and (e).

On the other hand, we have that $e \leq e_{max}$ $(\forall e)$, therefore we get inequality 3.11 with respect to any error e $(\leq r)$.

$$r - e_{max} \leq r - e \leq \delta^*_{\{e\leq r\}} \leq r + e \leq r + e_{max} \tag{3.11}$$

Consequently, the proposition ④ holds. □

Remark:

- From the perspective of the detection, improving prediction accuracy is more important than optimizing the range size. Namely, one should make sure that the prediction errors are as small as possible (e.g., $e<r$) before using the theorem 1 to optimize the range size in practice, otherwise, there will not be optimal value for the range size as stated by proposition ③.

- In practice, the next-step real data value $V(t)$ is unknown, thus the next-step prediction error, e, is unknown too. However, it is possible to estimate the prediction range by estimating the error e using recent prediction error values. As shown in Section 3.5.2, this works well in practice when errors are small and prediction errors at adjacent time steps exhibit a high degree of autocorrelation.

- There are two ways to get the value of user-expected accuracy r for an application. (1) Its value can be provided by the application's user based on his/her

particular demand. (2) The user-expected accuracy can be estimated based on the characterization of the corruption propagation phenomenon. The experimental results shown in Section 3.2.2, for example, indicate that bit-flips on bit positions < 5 never produce an error greater than 10^{-5} (out target threshold) anywhere in the whole data set, which allows an user-expected accuracy $r = 10^{-6}$ for this particular case.

3.4.3 Optimizing Range Size for Cluster-Based Detection Model. To optimize the range size of our cluster-based detectors, the changing trend of the estimated bounds must be taken into account, that is, the increase/decrease of the normal value interval bounds generated based on the analysis of the past data/features. The reason for this is that the group features of the current time step, which are more coarse-grained than point-wise features, may not always comply with the features at the last time step in practice. Suppose the estimated intervals at last two steps ($t-2$ and $t-1$) are computed as $[lb(t-2), ub(t-2)]$ and $[lb(t-1), ub(t-1)]$, respectively, where lb and ub refer to lower bound and upper bound, respectively. Then, the estimated bounds at the time t will be set as follows

$$lb(t) = 2 \cdot lb(t-1) - lb(t-2) \tag{3.12}$$

$$ub(t) = 2 \cdot ub(t-1) - ub(t-2) \tag{3.13}$$

This approach is the one followed for our cluster-based detectors in order to avoid a large number of false alerts.

3.5 Evaluation

In this section, a set of experimental results are presented to verify the efficacy

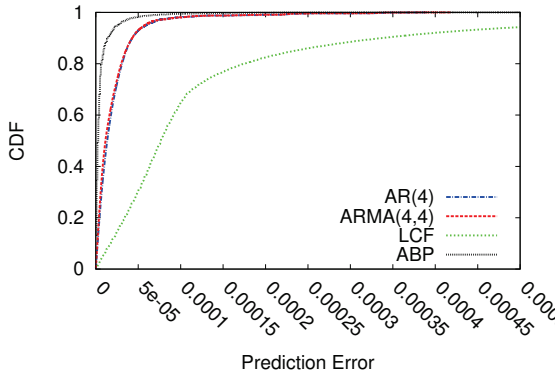
of our SDC detector in production-level iterative HPC applications. All the detectors, as well as the bit-flip fault injector, are already implemented transparently in the fault tolerance interface (FTI) toolkit [6] to protect the execution against silent data corruptions. FTI is originally used by applications to perform efficient checkpoints and restarts against applications' crashes. To protect their executions, users need simply to add a few library calls to the main loop of their iterative parallel program.

Four well-known and widely used HPC applications and computational kernels are evaluated: a computational fluid dynamics (CFD) miniapp [55], Nek5000 [56] (a CFD kernel), FLASH [57] (a multiphysics, multiscale simulation code), and HACC [58] (an N-body cosmology application). The CFD miniapp simulates a turbulent flow in a 3D duct modeled as a large eddy simulation (LES) using a two-stage time-differencing scheme based on higher accuracy for compressible gas using Navier-Stokes equations. The 3D duct is divided into N sections along the length (x axis) of the duct, where N corresponds to the number of MPI ranks in the execution. LES codes are among the most challenging applications in this context because of their chaotic and hard-to-predict behavior. They also represents a large set of HPC applications, ranging from weather prediction to aerospace engineering. Nek5000 is a CFD solver based on the spectral element method. It is also being used for a large number of applications in diverse fields such as reactor thermal-hydraulics and biofluids. The FLASH code solves the fully compressible, reactive hydrodynamic equations and allows for the use of adaptive mesh refinement. It also contains state-of-the-art modules for the equations of state and thermonuclear reaction networks [59]. For FLASH, *Sedov* and *Sod* are selected, two typical application cases. The *Sedov* explosion problem [60] involves the self-similar evolution of a cylindrical or spherical blast wave from a delta-function initial pressure perturbation in an otherwise homogeneous medium [61], while the *Sod* problem [62] is a one-dimensional flow discontinuity problem [63]. HACC (for Hybrid/Hardware Accelerated Cosmology Code) is a cosmology code aimed at under-

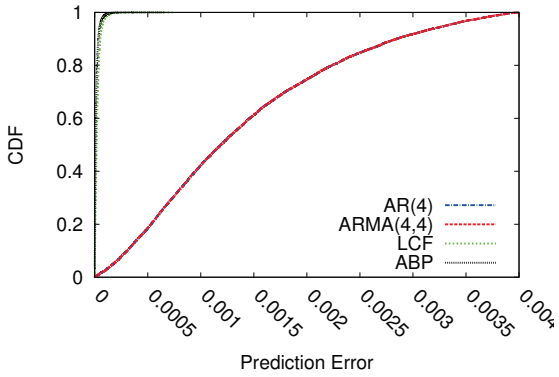
standing the nature of dark matter and dark energy in the universe. It uses N-body methods and is optimized for a wide range of systems, including those using accelerators. HACC divides the computation of the gravitational force into two phases: a long-/medium-range spectral particle-mesh (PM) component, which is common to all architectures, and an architecture-tunable particle-based short-/close-range solver.

3.5.1 Prediction Errors for One-Step-Ahead Linear Predictors. As shown in section 3.4, the optimal value of the detection range ($s^* = 2\delta^*$) depends on the user-expected accuracy r and the prediction error e. To get an idea of the magnitude of the prediction error, the distributions of the prediction errors under different predictors are characterized with the four mentioned iterative HPC application traces regarding different variables. The variables inlcude the position's coordinates (x) of the particles in HACC, the vertical flow and pressure (in a large eddy simulation) for Nek5000, the velocity for Turbulence-CFD, and a variable representing density for FLASH in the Sedov and Sod application cases. Prediction is performed at each time step for each data point, so each trace involves millions of prediction results, which allow us to build a cumulative distribution function (CDF) of the prediction error e, as shown in Figure 3.8.

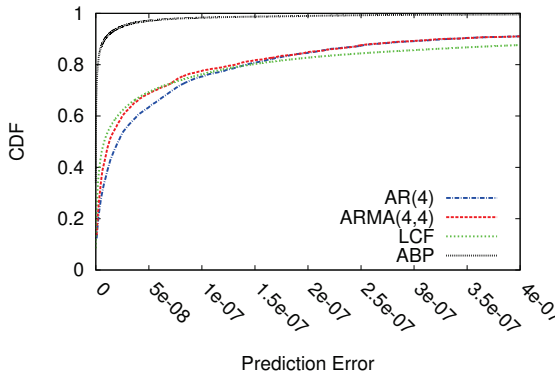
The most interesting result from these experiments is that a relatively simple predictor such as ABP is able to achieve smaller prediction errors than more complex linear models such as the well-known AR and $ARMA$ models for the selected HPC applications. In absolute terms, for the HACC application, up to 90% of the predictions have an error less than or equal to 10^{-5} under ABP, whereas only 8% to 56% of other predictors can reach such low errors. In the case of Nek5000, the prediction errors (90% of predictions) for vorticity and pressure are less than or equal to 8×10^{-9} and 2.8×10^{-10}, respectively. By comparison, other predictors are not able to achieve errors below 10^{-7}.



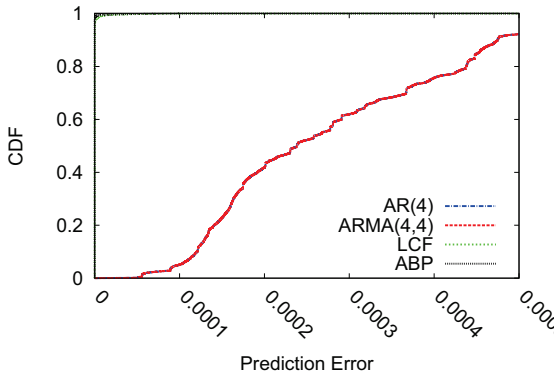
(a) HACC (particles' position)



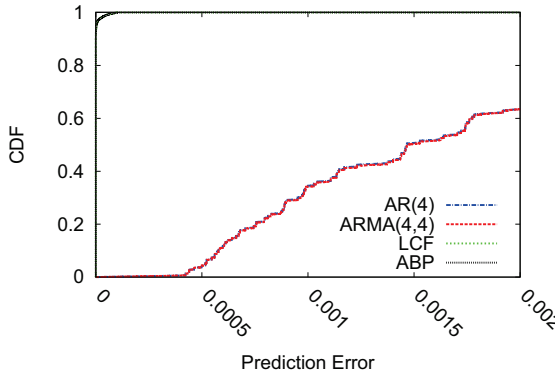
(b) Turbulence-CFD (velocity)



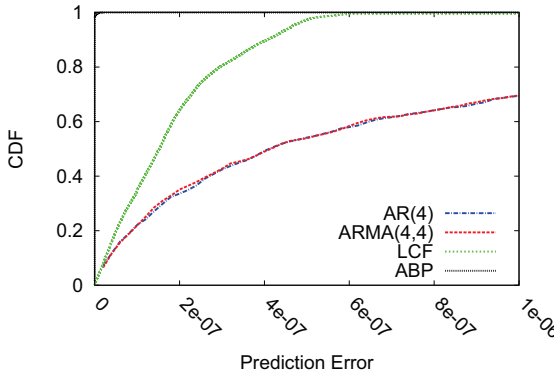
(c) Nek5000 (vortex)



(d) FLASH-Sedov (density)



(e) FLASH-Sod (density)



(f) Nek5000-eddy (pressure)

Figure 3.8. Cumulative distribution function of prediction errors for different predictors and HPC datasets

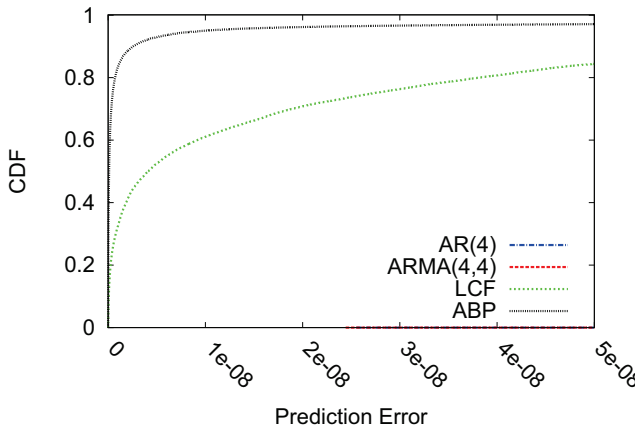


Figure 3.9. CFD of prediction errors for FLASH-Sedov (zoomed)

In the case of FLASH (Figure 3.8(d) and (e)), predictors *AR* and *ARMA* suffer from fairly large prediction errors. Thus, when their CDF curves are shown in the figure, it is hard to see the *LCF* and *ABP* curves clearly. In particular, most of the prediction errors of LCF and ABP stay between 10^{-10} and 10^{-9}, whereas those of *AR* and *ARMA* stay around 10^{-4}. One important discovery extracted from these experiments is that prediction errors between *LCF* and *ABP* are closer in FLASH than in other applications, indicating that the extra memory used in *ABP* may not actually produce significant benefits from the perspective of detection recall. In the case of Sedov (Figure 3.8(d)), we can observe the difference between *LCF* and *ABP* when zooming in as shown in Figure 3.9. For the case of Sod, however, the difference is so small that both curves still overlap even at a smaller observation range.

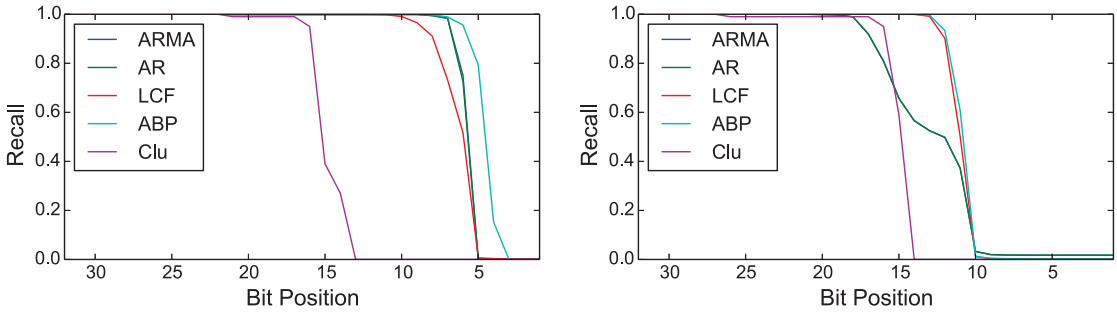
Based on the evaluation of prediction errors, we can conclude that ABP is the best predictor for the SDC detection for two reasons. First, it has the lowest prediction errors overall for all the different HPC application cases. Second, it has a relatively low extra computation cost and memory overhead at runtime. For instance, the *AR* and *ARMA* models require not only more memory sizes per data point but also a coefficient learning phase for a number of time steps (in this case 10) in the training period.

3.5.2 Detection Results on Traces. Before presenting the detection results, a dicussion of how we compute the value of the detection range (or the radius of what will be considered the normal data range: δ) in practice is presented. Note that the optimal value of δ depends on the prediction error e, which is unknown at runtime. Hence, we cannot compute the optimal value for δ. Instead, we try to approximate this optimal by approximating the prediction error e. To this end, the last-step prediction error is adopted, since it is observed that prediction errors at adjacent time steps exhibit a high degree of autocorrelation (as does data itself), as shown in Equation (3.14).

$$e(t) = |V(t-1) - X(t-1)| \tag{3.14}$$

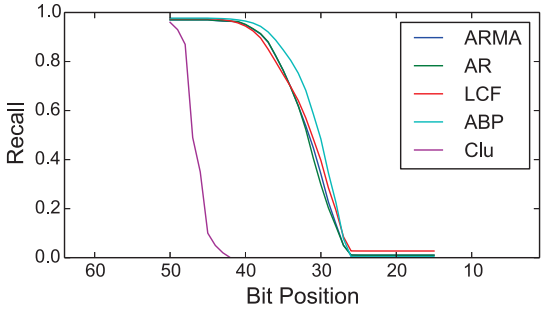
In the first set of experiments, presented in Figure 3.10, the detectors are run using the traces extracted from our six selected HPC datasets. The user-expected accuracy r is set with the help of the actual applications' developers. In the case of the single-precision datasets (HACC and Turbulence-CFD), we set $r = 10^{-6}$; and for the double-precision ones (Nek5000 and FLASH), the user-expected accuracy is set to $r = 10^{-8}$. The range size is set with the help of the presented analytical model (see Equations (3.9) and (3.10)) to maximize precision in order to avoid an excessive number of false positives, which could render the detectors prohibitively expensive, assuming that such false positives always trigger extra actions (e.g., application-aware data consistency checks). For instance, in the rollback recovery model, detectors with poor precision could produce highly frequent rollbacks, making the execution progression impossible.

It is important to clarify at this point the difference between memory overhead *per data point* versus *total* memory overhead. Apart from the memory state, or variables, applications allocate more memory for other tasks and intermediate com-

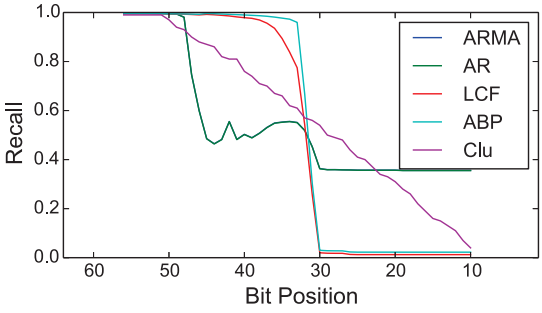


(a) HACC (particles' position)

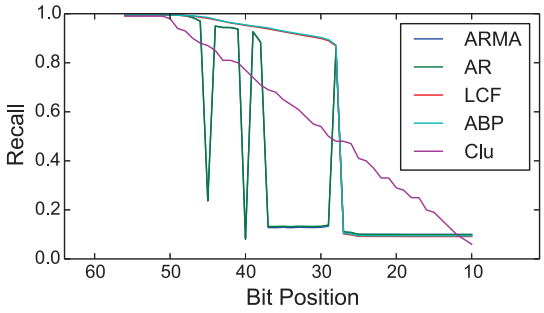
(b) Turbulence-CFD (velocity)



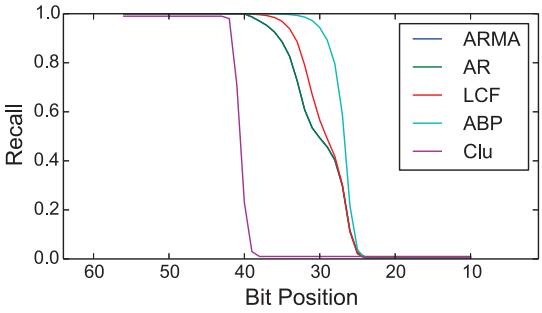
(c) Nek5000 (vortex)



(d) FLASH-Sedov (density)



(e) FLASH-Sod (density)



(f) Nek5000-eddy (pressure)

Figure 3.10. Recall for bit-flips injected on application traces

putations (e.g., fast Fourier transforms). What it is called overhead *per data point* is the percentage increase in memory usage with respect to the memory state, or variables, without counting other memory allocations. This is useful to clearly see how much past data do we need in order to make a one-step ahead prediction. See Section 3.5.4 for a more detailed analysis of memory, CPU, and network performance overheads for one of our one-step-ahead linear predictors.

In the presented results, it is possible to see that the cluster-based anomaly detector *Clu* (see Section 3.3.2), with less than 25% of memory overhead per data point, can guarantee a large coverage (over 50% of recall) against SDC in all cases. Moreover, for these HPC applications, the protected variables represent less than 25% of the used memory, which translates into less than 5% of the memory footprint for the cluster-based detectors. Therefore, these detectors can cover for the majority of corruptions at a negligible overhead.

Another observation is that the time-based pointwise detectors achieve the highest recall compared with the cluster-based detector. This result is not surprising given the amount of memory overhead per data point of these predictors—300% for LCF, 400% for ABP, 800% for AR, and 1600% for ARMA—as well as the more expensive computation involved. In particular, *ABP* achieves the best recall among all the other detectors, a result that is consistent with the prediction error results presented in Section 3.5.1. For *ABP* (whose total memory overhead is still below 100%) there is an overall coverage (all bit positions included) above 67% for the Turbulence-CFD and above 87% for HACC or above 99% if corruptions within the range of the user expected accuracy r are not considered. In the case of Nek5000, more than 85% of the corruptions can be detected in the case of vortex, and more than 95% for pressure if corruptions within the range of the user expected accuracy r are not considered. In the case of FLASH, it is possible to see that *ABP* and *LCF*

achieve similar recall values for both Sedov and Sod, which again is consistent with the prediction error results. Not counting corruptions smaller than the user-expected accuracy, *ABP* achieves a coverage of more than 84% and 92% for Sedov and Sod, respectively.

An interesting observation from these results is the good performance achieved by *Clu* (the cluster-based detectors) with more than 81% coverage in both FLASH application cases. This performance is due to the fact that the variable evaluated starts with very rapid changes at the beginning and levels off for all data points during the last two-thirds of the time steps. This stability suits *Clu* well, since it takes advantage of both temporal and spatial information.

The chaotic *AR* and *ARMA* curves shows what happens when prediction errors are too high $e >> r$ (as shown in Figure 3.8(d) and (f)). High prediction errors (because coefficients are learned during the first 10 time steps, which are not representative of the actual behavior of the data later on) directly affects the error ($e(t)$) estimation, which affects the computation of δ. It was shown that *AR* and *ARMA* compute detection ranges that are outside the user-expected interval (such as in 3.6(a) and (f)) and far from the true value $V(t)$. This, in turn, creates a situation where corruptions in higher-order bits may fall into the range, lowering recall, or corruptions in very low order bits may stay outside the range, artificially boosting recall. However, this high recall at low order bits comes with a high price in the form of a higher number of false positives, thus rendering *AR* and *ARMA* useless for these datasets.

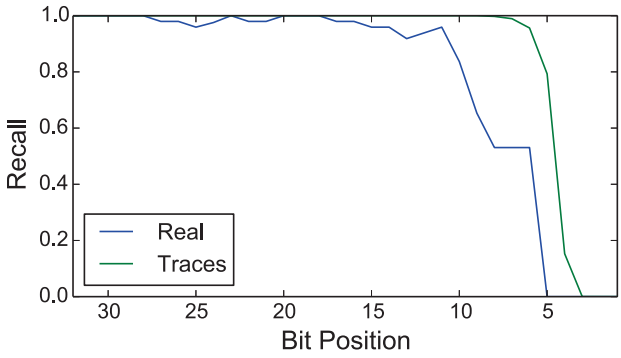
3.5.3 Detection Results at Runtime. In the second set of experiments, the detectors are tested in real application runs. The purpose of these experiments is to study how runtime detection results compare with detection results on traces. The ABP detector was chosen because it has the highest recall. HACC and Nek5000

are used as candidate applications. HACC is run with 512 MPI ranks and around 16 million particles, protecting the position and velocity variables. Nek5000 is run with 64 MPI ranks and a grid of 573,440 data points per rank. Seven variables are protected: position(x,y,z), velocity(vx,vy,vz), and pressure.

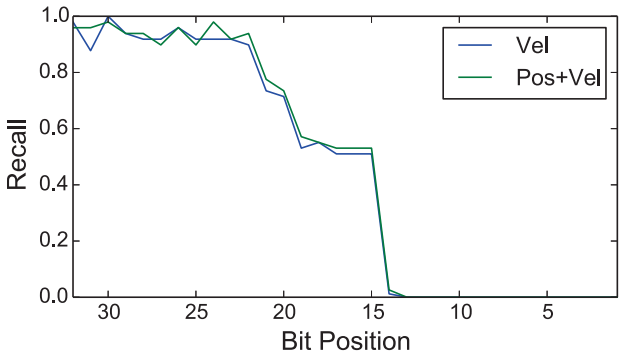
In Figure 3.11 bit-flips are injected at random particles and/or data points on particular bit positions on different datasets. Any detection occurring five time steps after the corruption (including the injection time step) is considered a true positive. In the figure, *vel* refers to injection and detection on the particles' velocity dataset, and *pos+vel* refers to injection on velocity while *detecting on position*. In the latter case (Figure 3.11b), the idea of leveraging datasets' correlation for detection (i.e., having a corruption in one dataset visible by the other) is explored. In the case of HACC, *velocity* is used to move a particle to a new *position*.

Three conclusions can be extracted from these results. First, if one consider only the corruptions outside the range of the user-expected accuracy, this method can cover over 90% of all possible corruptions for the position dataset in HACC (Figure 3.11a). Similarly, if only the corruptions outside the user-expected accuracy are consider, then the method can cover 66% of corruptions for the vortex dataset in Nek5000 (Figure 3.11c).

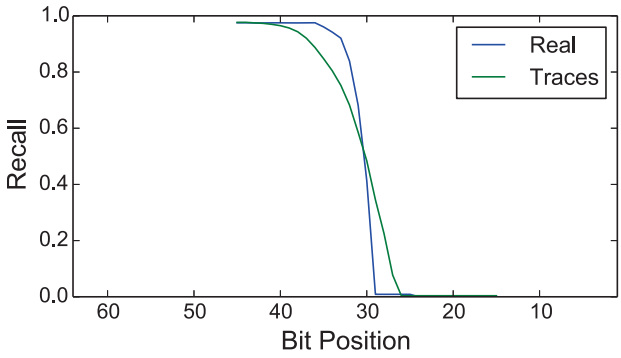
Second, the performance of the detector depends heavily on the underlying dataset (Figure 3.11b). Extra observations (not shown here) shown that position changes are smoother than velocity changes, thus making the next values for velocity more difficult to predict. In fact, these extra observations also shown that prediction errors for velocity are an order of magnitude higher than those for position. The good news is that leveraging datasets correlations works. Apart from the savings in memory overhead, these results indicate that it is possible to achieve a similar recall by monitoring only position, rather than monitoring both.



(a) HACC (particles' position)



(b) HACC (velocity)



(c) Nek5000 (vortex)

Figure 3.11. Comparing recall for bit-flips injected during real executions

Third, the evaluated predictors have different results depending on whether one work with traces or real application runs. The reason for such disparities is that traces do not represent the totality of the application's data state and are used only to construct distributions to help us understand different predictors and parameters. In any case, the results are indeed similar enough to make the author confident in the experiments using traces.

Experiments for other detectors (e.g., cluster-based detectors) were also performed, showing the significance of the trace-based results; but for brevity we do not plot those figures.

3.5.4 Performance Overheads. In this section the cost of our different detectors is analyzed in relation to their performance. By using the *ABP* detector and assuming the protection of all datasets in the HACC application, the overheads imposed on the application are 84% extra memory consumption, 13.75% extra computation time, and 0% extra network communication. These overheads are calculated for a relatively small HACC run (512 MPI ranks). However, the fact that this approach does not have any network overhead makes it automatically scalable to larger runs. As applications scale, communication time tends to become more dominant in general. Since the network overhead induced by these predictors is zero, it is expected that the extra computation time (a relative metric) will decrease as more ranks and larger datasets are introduced.

An 84% memory overhead might be acceptable for applications that are not memory bound, such as Nek5000. However, it might be too expensive for applications such as HACC that are memory bound (in particular, at extreme scale). In such cases, the author recommeds using cluster-based detectors that, with a memory and computation overhead under 5% (in the case of HACC application), can still guarantee over 50% of coverage, or over 60% if only corruptions outside the range of the user-

expected accuracy are considered. The ABP detector improves the recall only by 11% and 4% in comparison with cluster-based detectors for the CFD and FLASH-Sedov codes, respectively, while imposing 16 times more memory consumption per data point. Hence, depending on memory availability, users may choose ABP or the cluster-based detector.

3.6 Related Work

The problem of data corruption for extreme-scale computers has been the target of numerous studies. They can be classified in four groups depending on their level of generality, i.e., how easily a technique can be applied to a wide spectrum of HPC applications. They also have different costs in time, space, and energy. An ideal SDC detection technique should be as general as possible, while incurring a minimum cost over the application.

3.6.1 Hardware-Level Detection. The most general method is to solve the problem of data corruption at the hardware level. This method is extremely general because applications do not require any adaptation to benefit from such detectors. Considerable literature exists on soft errors rates [16, 64, 50, 65] and detection techniques at the hardware level [66, 67]. However, it is difficult to implement these techniques efficiently under the strict constraints of extreme-scale computing (e.g., low power consumption). Moreover, market interest in driving technologies in this direction is uncertain.

3.6.2 Process Replication. Process replication has been used for many years to guarantee correctness in critical systems, and its application to HPC systems has been studied. Fiala et al., for example, proposed using double-redundant computation to detect SDC by comparing the messages transmitted between the replicated processes [68]. The authors also suggested using triple redundancy to enable data

correction through a voting scheme. This approach is general in that applications need little adaptation to benefit from double or triple redundancy. Their customized MPI implementation (RedMPI) assumes that corruption in application data manifests itself by producing different MPI messages between replicas. In [69], the authors take advantage of multithreading and multicore processors to replicate threads of execution, so faults can be detected by comparing outputs from replicated threads. Unfortunately, double- and triple-redundant computation always imposes large overheads, since the number of hardware resources will be doubled or tripled. In addition, the cost of energy consumption is heavily increased when using full replication, not only because of the extra computation, but also because of the extra communication. In contrast with process replication, the techniques explored in this work do not incur any network overhead, and their memory footprints are always below 100% (see Section 3.5.4).

3.6.3 Algorithm-Based Fault Tolerance. A promising technique against data corruption is algorithm-based fault tolerance (ABFT) [70]. This technique uses extra checksums in linear algebra kernels in order to detect and correct corruptions [71]. ABFT is not general, however, since each algorithm needs to be adapted manually, and only some linear algebra kernels have been adapted, which is only a subset of the vast spectrum of computational kernels. Furthermore, even applications that employ only ABFT-protected kernels could fail to detect SDCs if the corruption lies *outside* the ABFT-protected regions. In comparison, the proposed approach aims to protect against any error in any part of the application by covering those datasets (or variables) that represent the memory state in an iterative HPC application.

3.6.4 Approximate Computing. Another type of SDC detection is based on the idea of approximate computing. In this detection method, a computing kernel is paired with a cheaper and less accurate kernel that will produce *close enough* results.

Such results can be compared with those generated by the main computational kernel [72]. This detection mechanism shows promising results, but again it is still not general enough, because each application needs to be manually complemented with the required approximate computing kernels. Furthermore, complex applications also need to adapt multiple kernels in order to offer good coverage.

3.7 Discussion

In this Chapter, a new model to tackle the problem of SDC detection using user-expected accuracy and past prediction errors is developed. A large number of linear predictors that take advantage of the characteristics of iterative HPC application datasets are compared, and a model to tune its parameters to achieve almost perfect precision in the case of small prediction errors is theoretically analyzed. The proposed detectors are implemented and evaluated with production-level scientific applications using both traces and real experiments on supercomputers. The key new insights and important findings are summarized below.

- *Error Propagation Study*: It is shown that corruption on some bit positions might be negligible, since injection in those bits does not deviate over a certain treshold.

- *Prediction Results*: For HACC, up to 90% of predictions have an error lower or equal to 10^{-5} under ABP, compared with only 8% to 56% for other predictors. For Nek5000, the prediction errors can be reduced to $8{\times}10^{-9}$ for 90% of predictions. For FLASH, most of prediction errors of LCF and ABP stay between 10^{-10} and 10^{-9}, while those of AR and ARMA stay around 10^{-4}.

- *Detection Results with Injected Errors*: ABP detectors lead to the highest recall for all the applications and kernels evaluated: up to 95% of SDC can be covered with almost no false positives (FP).

- *Overhead*: Cluster-based detectors are the most cost-effective detectors, covering the majority of corruptions (over 50% of recall) for a negligible cost (less than 5% of overhead in some cases).

It has been shown that it is viable to detect corruptions in one variable (such as velocity) by using another (such as position) in one of our applications (HACC), taking advantage of the fact that these variables are interconnected by the underlying physics of the application.

Any iterative HPC application wanting to protect itself against SDCs using these techniques would need to have the bulk of its variables behaving in a smooth way, as those shown in Figure 3.1. Since a large set of HPC datasets have been studied by working with well-known applications and kernels, one could feel positive that these methods can work on many applications. Observations suggest that *ABP* is the best predictor so far, and *Clu* the most cost-effective; but it is still worth exploring whether other applications not included in this work would benefit from other predictors, or a combination of them.

It is important to note that these techniques work only for iterative HPC applications. Although such applications are the majority today, especially in the scientific area of computational physics, other types of applications are becoming important in the HPC community in areas such as computational biology and statistics. Also, simulations with abrupt variable changes, like applications involving collisions, may need different types of detectors. This thesis present work on this area in chapter 4.

CHAPTER 4

IMPROVING DATA-ANALYTIC-BASED SILENT DATA CORRUPTION DETECTION FOR APPLICATIONS WITH NON-SMOOTH VARIABLES

4.1 Introduction

Silent data corruption (SDC) involves corruption to an application's memory state (including both code and data) caused by undetected soft errors, that is, errors that modify the information stored in electronic devices without destroying the functionality [73]. If undetected, these errors have the potential to be damaging since they can change the scientific output of HPC applications and mislead scientists with spurious results.

External causes of transient faults are usually rooted in cosmic ray particles hitting the electronic devices of the supercomputer. A study by Snir et al. [74] found that SRAM devices get hit once every 10 hours for 1 TB of capacity, latches every 41 days for 10 million units, and DRAM once every 40 days for 100k devices. As HPC systems keep scaling up, the increasing number of devices will make these external faults appear more often. Other techniques introduced to deal with excessive power consumption, such as aggressive voltage scaling or near-threshold operation, may also increase the number of errors in the system [75]. Moreover, software complexity (in the form of more complex operating systems and libraries) may also introduce corruptions throughout obscure and difficult-to-reproduce bugs [75].

Substantial work has been devoted to this problem, both at the hardware level and at higher levels of the system hierarchy. Currently, however, HPC applications rely almost exclusively on hardware protection mechanisms such as error-correcting codes (ECCs), parity checking, or chipkill-correct ECC for RAM devices [76, 77]. As we move toward the exascale, however, it is unclear whether this state of practice can continue. For example, recent work shows that ECCs alone cannot detect and/or

correct all possible errors [17]. In addition, not all parts of the system, such as logic units and registers inside the CPUs, are protected with ECCs. The reason is that ECCs, and other hardware protection mechanisms, utilize extra power, hardware area, and latency for the computation required to encode and decode the check bits [78], making them also unlikely choices to be extended to other parts of an exascale system.

Four major software solutions have been proposed by the community: (1) Full replication [79, 80], (2) algorithm-based fault tolerance (ABFT) [81], (3) approximate computing [82], and (4) data-analytic-based (DAB) fault tolerance [83, 84, 85, 86, 87]. ABFT and approximate computing are not general and have limited applicability, since both require to modify each kernel manually and only a subset of the kernels can be protected. Full process replication provides excellent detection accuracy for a broad range of applications. The major shortcoming of full replication is its overhead (e.g., spatial overhead of 100%+ for duplication, 200%+ for triplication). The DAB approach is a recent solution, where detectors take advantage of the underlying properties of the applications' data (the smoothness in the time and/or space dimensions) in order to compute likely values for the evolution of the data and use these properties to flag outliers as potential corruptions.

Although the DAB solutions provide high detection accuracy for a number of HPC applications with low overhead, their applicability is limited because of an implicit assumption—the application is expected to exhibit *smoothness* in its variables throughout its execution. Nevertheless, it has been observed that not all the processes of parallel applications experience the same level of data variability at exactly the same time; hence, one can *smartly* choose and use more expensive detection methods (such as replication) in only those processes for which DAB detectors would perform poorly.

Another open issue is the need to standarize the evaluation of SDC detection

methods. Currently, evaluation results are mainly reported in the form of single-bit precision and recall rates. This work argues that there is a major problem with this type of reporting, namely, that it focuses exclusively on single-bit corruptions. As it is shown in Section 4.4, single-bit corruptions are the hardest ones to detect using software mechanisms (and the easiest ones to detect at the hardware level). Focusing solely on single-bit errors, it is argued, makes it difficult to compare protection between different types of mechanisms working at different levels of the system hierarchy, such as comparing ABFT with full replication.

In this work, two new contributions for SDC detection in HPC are made. First is *a new adaptive SDC detection approach* that combines the merits of replication and DAB in order to significantly increase levels of detection recall in applications with sharp data changes (i.e., up to 99.9%), without incurring as much overhead penalty as in full replication. Two adaptive algorithms are proposed: One in which the number of processes replicated is fixed throughout the execution, and another where the number of processes replicated is allowed to change dynamically. Next, a *new evaluation method* is proposed based on the probability that a corruption will pass unnoticed by a any detector. In contrast to using overall single-bit precision and recall, this new evaluation allows anyone to compare the detection approach presented against mechanisms working at different levels of the system hierarchy—such as full replication—using a single metric.

Extensive experiments are conducted to compare the adaptive methods with full replication and DAB in terms of detection accuracy and space/time overheads. Four applications dealing with astrophysics explosions from the FLASH code package [88] are used. These applications are excellent candidates for testing partial replication on applications dealing with explosions or collisions because the *smoothness* property, which is essential to guarantee the success of DAB solutions, is lost on

the data of those processes through which the explosions' waves are passing through at that particular moment.

Results show that the adaptive approach is able to protect the applications analyzed (99.9% detection recall) replicating between 23–83% of all the processes with a total overhead of 33–102% depending on the application (compared with 110% for full duplication).

The rest of the paper is organized as follows. In Section 4.2 how DAB SDC detectors work is described. In Section 4.3 the adaptive methods for SDC detection are introduced. Section 4.4 describes the probabilistic evaluation metric used. In Section 4.5 the experimental results are presented. Section 4.6 discusses related work in this area. In Section 4.7 the key findings are summarized and future directions for this work are presented.

4.2 Data-Analytic-Based SDC Detectors

In this section an overview of data-analytic based (DAB) methods and their major limitations is given.

Lightweight DAB SDC detectors are based on the observation that data evolution is *smooth* in the time and/or space dimensions in a range of variables in HPC applications [86]. They are composed of two major parts [86, 87]. The *predictor* component computes a prediction for the next value of a particular data point. The prediction takes advantage of the underlying physical properties of the evolution of the data. After the prediction is done, the *detector* component decides whether the current value of the data point is corrupted. Since the predictions are always going to have some error, detection is done by checking whether the current value of the data is inside a *confident interval* surrounding the prediction. If it is not, it is considered to be a corruption.

For the time dimension, it was found that quadratic curve fitting (QCF) outperformed all the other considered options [87]. QCF uses the previous three time steps of every data point to compute the prediction for the current time step (see Equation (4.1)), incurring some memory overhead. In particular, the memory state is multiplied by 4x. That overhead, however, is calculated with respect to the memory state (the variables used to recover an execution from a checkpoint), which do not represent the whole memory consumed by applications. QCF is usually never above 90% overhead for HPC applications.

$$\begin{aligned} X(t) &= V(t-1) + \Delta^2(t-1) + \Delta(t-1) \\ &= V(t-1) + (\Delta(t-1) - \Delta(t-2)) \\ &\quad + (V(t-1) - V(t-2)) \\ &= 3V(t-1) - 3V(t-2) + V(t-3). \end{aligned} \tag{4.1}$$

Another way to do predictions is by using the spatial information instead of the temporal information. In [89], 3D linear interpolation was used succesfully to predict values in a computational fluids dynamics (CFD) miniapplication. 3D linear interpolation is shown in Equation (4.2), where a prediction is made for the point situated at the (x, y, z) coordinates. The prediction for each single coordinate is computed using the value of the line—at the position of the coordinate—passing through the points at both sides, i.e., points a (left/down) and b (right/up). The final prediction $X(t)$ is the average of the prediction at each coordinate. Using spatial information has the benefit that no extra memory overhead is required, since only data for the current time step is used. Temporal predictors are usually more accurate than spatial ones, so choosing one or the other represents a memory overhead versus prediction accuracy tradeoff.

$$
\begin{aligned}
X_x(t) &= V_{x_a,y,z}(t) + ((V_{x_b,y,z}(t) - V_{x_a,y,z}(t)) \times \frac{x - x_a}{x_b - x_a}) \\
X_y(t) &= V_{x,y_a,z}(t) + ((V_{x,y_b,z}(t) - V_{x,y_a,z}(t)) \times \frac{y - y_a}{y_b - y_a}) \\
X_z(t) &= V_{x,y,z_a}(t) + ((V_{x,y,z_b}(t) - V_{x,y,z_a}(t)) \times \frac{z - z_a}{z_b - z_a}) \\
X(t) &= \frac{X_x(t) + X_y(t) + X_z(t)}{3}.
\end{aligned}
\tag{4.2}
$$

To ease the reader with the notation used throughout this paper, Tables 4.1 and 4.2 show the most important notations along with their description.

Once a prediction $X(t)$ has been made, the detector decides whether the current value of the data $V(t)$ is a normal value by checking whether it falls inside a particular *confident interval* defined as $[X(t)-\delta, X(t)+\delta]$, where δ equals to half the size of this interval. If it does not, it is flagged as corrupted (a high level overview of DAB detectors is illustrated in Figure 4.1) . The value of δ can be calculated by using the maximum prediction error from all data points in a process at $t-1$ multiplied by a parameter: $\delta = \lambda \cdot e_{max}(t-1)$. This parameter λ determines a tradeoff between detection *recall* (how many real corruptions can we actually detect) and *precision* (how many of the detected corruptions are actually real corruptions). These metrics are defined in Equations (4.3) and (4.4).

$$
\text{recall} = \frac{\text{True\_Positives}}{\text{True\_Positives} + \text{False\_Negatives}}. \tag{4.3}
$$

$$
\text{precision} = \frac{\text{True\_Positives}}{\text{True\_Positives} + \text{False\_Positives}}. \tag{4.4}
$$

For all the experiments in this work, the value for λ is chosen to have zero false positives given a particular execution size (i.e., *precision*=100%).

Table 4.1. Description of notation used throughout the paper

Notation	Description
$X(t)$	Prediction made by a DAB lightweight predictor at time t for a particular variable.
$V(t)$	Real value of an application's variable at time t.
$\Delta(t)$	Velocity of data at time t, i.e., amount of change in a variable between times $t-1$ and t.
$\Delta^2(t)$	Acceleration of data at time t, i.e., amount of change in the velocity of a variable between times $t-1$ and t.
δ	Half the size of the *confident interval*. $\delta = \lambda \cdot e_{max}(t-1)$.
λ	Parameter to determine by how much $e_{max}(t-1)$ should be multiplied to calculate δ for step t.
$e_{max}(t-1)$, MPE	Maximum prediction error among all data points in a process at time $t-1$ for a particular variable.
w	Paremeter to determine how often do we change the set of replicated processes.
n	Number of processes in an application.

Table 4.2. Description of notation used throughout the paper (continued)

Notation	**Description**
B, r	Replication budget (number of processes to replicate). r is actually a rate, where $r \in [0, 1]$
S	Vector of scores for all processes. The higher the score for a process, the less smoothness in the data.
α	Factor used to control the size of the elastic budget.
ζ_i	Rate of a particular process MPE to that of the "worst" process during window w.
Ψ	Parameter which defines a threshold for the rate ζ_i, above which process i's data behavior is considered similar enought to that of the "worst" process.
P_f	Probability of undiscovered corruption.
$O(r, w)$	Overhead for a replication budget r and window w.

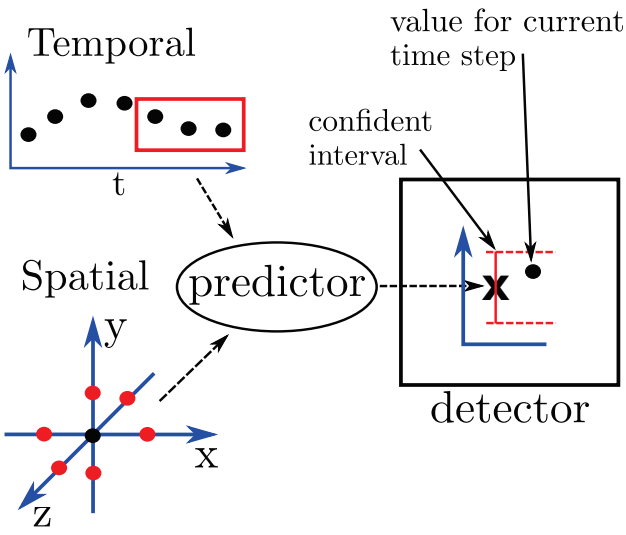


Figure 4.1. High level overview of DAB detectors. The predictor can use temporal information, spatial, or both.

When data values change too abruptly in a particular process, δ becomes far too large to detect barely any corruption.

To illustrate the problem at hand, Figure 4.2 and 4.3 shows detection recall rates for single bit-flips injected on each bit position over two different processes in the variable *pressure* of the Sedov application during a particular period of time (100 iterations). Sedov is a hydrodynamical test code involving strong shocks and nonplanar symmetry [90] from the FLASH simulation code package. One can see how the wave of the explosion passing through rank 87 is making detection recall rates decrease substantially for this variable[10]. In contrast, detection recall is high in rank 99. To get a glimpse of how this data looks like, consider Figure 4.4. Here, the state of the maximum of variable *pressure* is shown right after the window of 100 time steps has passed. In the figure, every square represents the data grid on a different rank.

[10]The rank of a process in MPI is its ID inside a group of processes. In this paper only the rank of the general group to which all processes belong is considered. In this sense, rank(s) or process(es) are used interchangeably.

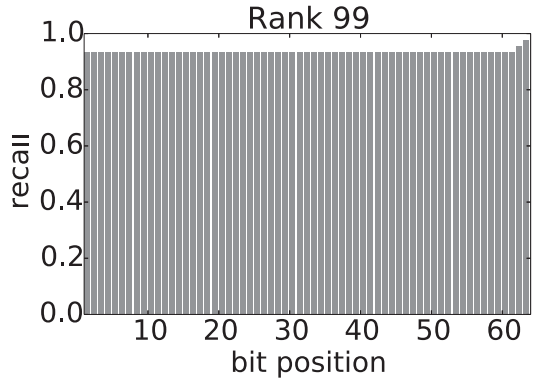


Figure 4.2. Detection recall for process 99 in Sedov during 100 iterations

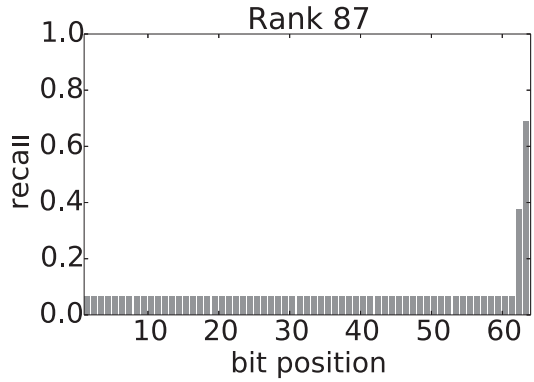


Figure 4.3. Detection recall for process 87 in Sedov during 100 iterations

4.3 Adaptive Method

In this section the advantages and disadvantages of replication are described, focusing particularly on partial replication. After this, the two adaptive algorithms proposed in order to choose what ranks to replicate in each application are shown; the first algorithm, first proposed in [91], uses a fixed number of replicated processes (what is called a *budget*) while the second uses an elastic budget (i.e., the number of processes to replicate changes over time).

4.3.1 Overview. Replication is the best solution so far, in terms of detection recall, for dealing with data corruptions in scientific simulations. It has the benefit of not

being affected by data behavior, since replication always makes sure that the output of all the replicas (or the majority of them) agree with each other bitwise[11]. Full replication, however, is generally considered too costly for HPC because of its high overhead both in the time and the space dimensions.

Partial replication is especially useful for applications for which sharp changes in the data occur only in a small subset of the MPI ranks, such as those involving explosions or collisions (e.g., Sedov). Considering again the example introduced in Section 4.2, we can see that duplicating rank 99 in this situation is a major waste of resources given that the lightweight detectors already do the job well. On the other hand, rank 87 can surely benefit from replication, making detection recall go from below 10% in the majority of bits to 100% in all of them automatically.

One way to detect corruptions efficiently by using replication, proposed by Fiala et al. [79], is by comparing messages in MPI. The idea is that any corruption in the data of a particular MPI rank will ultimately produce corrupted messages that will be sent to other MPI ranks. By comparing messages from replicas of the same rank, one can determine whether that rank (or any of its replicas) got its data corrupted. This detection mechanism avoids the costly comparison of the data itself, which in some cases can be huge, although it generates extra messages in the network, and introduce some runtime overhead. However, multiple optimizations exist, such as only sending one full message from the original replica and hashes from all the others. A corruption, then, is detected at the destination by hashing the only whole message received and comparing all the hashes bitwise.

This work adopts *an adaptive strategy*. For some ranks, partial replication is used, based on the method of Fiala et al. For the other ranks, the lightweight DAB

[11]This can work, of course, only for *code-determenistic* applications, since the same input should always produce the same output.

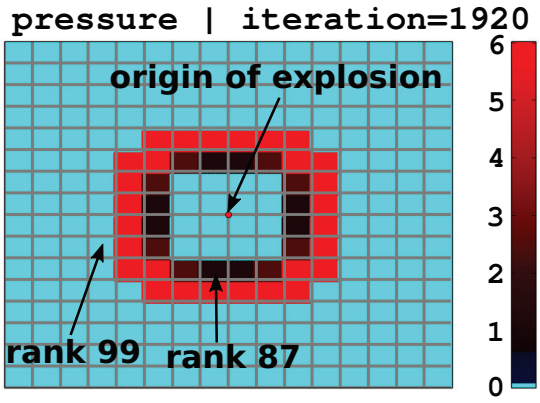


Figure 4.4. Data evolution for the variable *pressure* in Sedov

detectors are used. The set of ranks to replicate (budget) is chosen carefully based on the data behavior in order to improve overall recall significantly (i.e., 100%), but with less spatial and temporal overhead. The ranks belonging to this set are changed dynamically during execution with a predefined frequency. In addition, the size of the budget can also be adjusted over time to accomodate for the situations where sharp data changes concentrate also in time.

In this work, both algorithms are implemented – fixed and elastic budget – and compared to DAB-only detection. In Section 4.3.2, the fixed budget algorithm is presented, and in Section 4.3.3 the elastic one is discussed.

4.3.2 Fixed Budget Algorithm. Before we can choose which ranks to replicate, we need a way to measure which ones would perform poorly with lightweight detectors. It is observed that using the *maximum prediction error* (MPE) among all data points in a rank is a good measure for this particular goal. The reason is that δ (the confident interval) grows and adapts to the new data changes even if only a small subset of the data points in the rank experiences sharp variations. Other measures were considered, such as *average prediction error*. However, it was found that the maximum error was the best. For example, the average is not fast enough to account for sharp changes in the whole rank, given the overwhelming weight that "normal" points have in the

average calculation.

Algorithm 4 Set selection with a fixed budget

```
1: for (iter ∈ iterations) do
2:     ... /* APP computation */
3:     MPE ← maxErrorsPerRank()
4:     R ← sort(ranks(),MPE)
5:     if (iter = 0) then
6:         REPLICATE TOP B RANKS FROM R
7:         S ← newArray(n × [0])
8:     else
9:         for (i = 0; i < n; i++) do
10:            S[R[i]] ← S[R[i]] +i
11:        end for
12:        if (iter % w = 0) then
13:            R ← sort(ranks(),S)
14:            REPLICATE TOP B RANKS FROM R
15:            S ← newArray(n × [0])
16:        end if
17:    end if
18: end for
```

The following strategy is implemented, shown in Algorithm 4, in order to select the replication set and to dynamically adapt it over time. After the first iteration, a subset of processes to replicate is chosen, given the MPE in that first iteration (lines 5–8). The number of processes to replicate is determined by the *replication budget* B and it is fixed during the whole computation. During the following w application time steps (where w defines a *window*), an array S of size n is created, which is the number of processes in the application. After every time step, all processes are sorted in ascending order given their MPEs (line 4) and their positions are added in S (lines 9–11). For example, if at a particular time step, rank 12 is the one with the highest prediction error and there are 128 processes, then $S[12]$ += 128. When w steps have

passed, the score S represents an aggregation of the relative positions of each rank with respect to the others given their prediction errors during the window w. At this point (lines 12–16) all processes are sorted by their score S, the top B picked (which is the allocated budget) as the new replication set, and S is reset to start a new window again.

4.3.3 Elastic Budget Algorithm. When working with a fixed budget, we might replicate too many processes during some periods of the execution in order to compensate for peak periods of sharp data changes. The elastic method addressed the above issue, in which the number of processes that are replicated changes dynamically given the properties of the data. The main idea is that sharp data changes could be concentrated not only in a particular place in space but also in time.

In order to have an elastic budget, we need a way to measure, at a particular point in time, how many processes are behaving "bad enough" to merit replication. Like in the fixed budget case, the MPE is used for such a measure. Since data behavior greatly diverge between different data sets (even for the same application), it is found that it is not possible to use a fixed MPE amount as a threshold. Instead, a rate is defined to compare a particular process MPE with that of the "worst" process. When this rate is above a defined threshold, it is possible to say that the data behavior for that particular process is "similar enough" to the worst behaving process and, therefore, it is a good candidate for replication. This rate is described in Equation (4.5).

$$\zeta_i = \frac{\sum_{j=0}^{w-1}(\text{MPE}[j][i])}{\sum_{j=0}^{w-1}(\text{MPE}[j][R[0]])}. \tag{4.5}$$

The same notations as in Algorithm 4 are used, changing MPE from a vector of size n to a matrix of size $w \times n$, where MPE$[j][i]$ corresponds to the MPE of process i

at time step j, where $0 \leq j < w$. R holds the processes' ranks (i.e., IDs) sorted in descending order by their score S (as shown in line 13 of Algorithm 4). Therefore, $R[0]$ holds the rank of the worst behaving process during the time window w. Realize that the rate ζ_i is a rate of two averages, i.e., the average MPE of process i divided by the average MPE of the "worst" process during window w.

The elastic method has to control the average number of processes replicated during the whole execution. This is needed for mainly two reasons. First, because we want to be able to compare – in terms of overhead – elastic versus fixed for different budgets [12]. The second reason is related to the fact that, when using the rate ζ_i to discriminate processes, there seems to be an upper limit in the size of the budget above which recall can not be improved. This occurs either because sharp data changes never happen in more than a particular number of processes, or because they are very concentrated in time. In these situations, and if the budget allows, it may be advantageous to replicate more processes even when they do not "behave bad" (i.e., when lightweight SDC detectors have good performance).

Given this, the following strategy is used in order to keep the average number of processes replicated as close to a defined budget as possible. First, a total budget is defined that is computed every time a new decision about replication needs to be made. This is shown in the following Equation:

$$\text{Total\_Budget} = \frac{\text{iter}}{w} \times B, \tag{4.6}$$

where iter is the current iteration (i.e., time step) in the execution and B is the desired final average number of replicated processes. This Total_Budget can be thought of

[12] A *a budget* in the elastic case is an upper limit in the *average* number of processes replicated during the whole execution.

as a "replication income" that is increased every w steps. An accumulator Acc that keeps track of the actual total number of processes replicated from the beginning of the execution is also defined. This can be thought of as "incurred expenditures". Therefore, the available savings can be calculated using Equation (4.7).

$$\text{Saved\_Budget} = \text{Total\_Budget} - \text{Acc}. \tag{4.7}$$

As long as we have that Saved_Budget ≥ 0, we know that we will never go above the budget. The problem now, however, is that it is impossible to know how much budget is needed to save and when it is wise to expend it. If it is never expended, it is possible to miss opportunities to increase recall. On the other hand, if it is expended too early or before a period of massive sharp data changes, recall gains can get penalized. The only way to perfectly solve this problem would be to know the future. Since it is not possible to do that, some heuristics are used to know when it is acceptable to spend the saved budget. The full approach is presented in Algorithm 5.

Algorihtm 5 is divided into two main parts. In the first (lines 5–15), the elastic budget and the vector S of scores (see Section 4.3.2 for details) are calculated. The elastic budget is calculated differently depending on whether we are in the very first iteration (lines 5–7) or we are at the end of a window of w time steps (lines 8–10). For the former, the function *calcEB()* is called (which calculates the elastic budget, described in Algorithm 6) with the parameter $w = 1$ (there is only one row in matrix MPE). The accumulator is initialized in line 7. For the latter, R is calculated using the vector of scores S (the same way as it is done in line 13 of Algorithm 4) and then the elastic budget using the new R and the full matrix MPE are calculated (line 10). The vector S is computed during normal iterations (lines 11–14).

Algorithm 5 Set selection with an elastic budget

```
1: for (iter ∈ iterations) do
2:     ... /* APP computation */
3:     MPE[iter % w] ← maxErrorsPerRank()
4:     R ← sort(ranks(),MPE[iter % w])
5:     if (iter = 0) then
6:         elastic_budget ← calcEB(MPE,R,1,n)
7:         Acc ← 0
8:     else if (iter % w = 0) then
9:         R ← sort(ranks(),S)
10:        elastic_budget ← calcEB(MPE,R,w,n)
11:    else
12:        for (i = 0; i < n; i++) do
13:            S[R[i]] ← S[R[i]] +i
14:        end for
15:    end if
16:    if (iter % w = 0) then
17:        Saved_Budget ← (iter/w × B) − Acc
18:        α ← 1
19:        if (Saved_Budget ≥ 2 × n) then
20:            α ← Saved_Budget/n
21:        end if
22:        elastic_budget ← elastic_budget × α
23:        if (elastic_budget > n) then
24:            elastic_budget ← n
25:        end if
26:        REPLICATE TOP elastic_budget RANKS FROM R
27:        S ← newArray(n × [0])
28:        Acc ← Acc + elastic_budget
29:    end if
30: end for
```

The second part of the algorithm (lines 16–29) deals with the replication of processes. In lines 17–21 the saved budget and the factor α are calculated. The factor α is used to enlarge the elastic budget in the event that the saved budget grows to be greater than or equal to two times the total number of processes (this is the heuristic used to expend the extra saved budget). If that is the case, the factor will be the integer division of the saved budget divided by the number of processes. Lines 23–25 checks that the elastic budget is never above the total number of processes. Finally, the top elastic_budget processes from R are replicated (line 26), S reset (line 27) and the accumulator increased (line 28).

Algorithm 6 Calculation of the elastic budget

```
1: FUNCTION calcEB(MPE,R,w,n) BEGIN
2:     elastic_budget ← 0
3:     for (i = 0; i < n; i++) do
4:         ζ_i ← (Σ_{j=0}^{w-1} (MPE[j][i])) / (Σ_{j=0}^{w-1} (MPE[j][R[0]]))
5:         if (ζ_i > Ψ) then
6:             elastic_budget ← elastic_budget + 1
7:         end if
8:     end for
9:     return elastic_budget
10: END FUNCTION
```

Algorithm 6 shows the function *calcEB()*, which implements the calculation of the elastic budget. The function computes, for each process, its rate ζ_i (line 4) as defined in Equation (4.5). If the rate is above the threshold Ψ (more on this in Section 3.5) then it is said that this process' data behavior is similar enough to the worst behaving process and, therefore, it should be counted for the elastic budget.

4.4 Probabilistic Evaluation Metric

There is no standarized way to evaluate SDC detection methods. Reporting overall single-bit precision and recall rates are commonly adopted. However, it can be

argued that this may not be enough to understand how well applications are actually protected. Consider the case where we have a mechanism with perfect detection recall for the 22 most significant bits of 32-bit numbers. What is the probability that, in this particular example, a corruption will evade the detector? Using the recall results and assuming 1-bit-flip corruptions only, we could say that $10/32 = 0.3125$ (31% of corruptions will pass undetected). But what if 2-bit-flip corruptions are possible? The probability in this case would be $(10/32) \times (9/31) = 0.0907$ (9.07% corruptions will pass undetected). For our hypothetical detector to be unaware of a 2-bit-flip corruption in this case, all flips would always need to hit bits in the 10 less significant positions of the mantissa. In other words, a single bit-flip on position 20 is equivalent to a double bit-flip in positions 20 and 25. It is possible to continue with the case for 3-bit-flips, for 4, 5, and so on. An interesting observation from this example is that, generally, the fewer bits that can get "flipped" in a system, the harder it is to detect corruptions using software mechanisms. Furthermore, another interesting question appears: What is the distribution of corruption sizes (in terms of the number of bits) in the system? Is a corruption affecting a large number of bits more or less common than one affecting just a few?

The key idea is that protecting the numerical data of simulations at this level (e.g., by using replication, algorithm-based fault tolerance (ABFT) [81, 92], approximate computing [82], or DABFT) is not so much protecting against particular bit-flips as it is protecting against numerical corruptions in the form of deviations from the original data. A corruption could happen anywhere in the system: CPUs, memories, buses, interconnections, file systems, operating systems, libraries, and so forth. Assuming that corruptions in the system, which can spread and affect numerical data in a large number of ways, will always manifest themselves as single-bit corruptions is a big assumption. Nevertheless, single-bit injection studies are still useful since injecting all possible corruptions (2^{32} for single precision and 2^{64} for double precision)

is prohibitively expensive.

In this work an evaluation metric based on the probability that a corruption will pass unnoticed by a particular detector is used. Since the goal is to design general SDC detectors, it cannot be assumed that bit-flips in the less significant bits of the mantissa are harmless. For example, a a sensitivity study was performed where different corruption sizes on multiple applications were injected. Two of those applications were from the Nek5000 code package [93]: Vortex, which solves an inviscid vortex propagation problem [94], and Eddy, a 2D solution to the Navier-Stokes equations with an additional translational velocity [95]. It was found that while Vortex is able to absorb all corruption sizes, producing an impact of less than 0.0002 (0.02%)[13] at the end of the execution, Eddy gets affected by even the smallest of corruptions, producing an impact of more than 0.4 (40%) in all cases.

The evaluation metric, called the *probability of undiscovered corruption*, is defined as

$$P_{\mathrm{f}} = \sum_{i=1}^{N} \Big[P(\#\mathrm{bits} = i) \times (1 - \mathrm{aveRec})^{i} \Big], \tag{4.8}$$

where N is the number of bits per data point (i.e., 64), aveRec represents the average recall rate for all bit positions collected during an injection study (see Equation (4.9)), and $P(\#\mathrm{bits} = i)$ represents the probability that the corruption is exactly i bits long.

$$\mathrm{aveRec} = \frac{1}{N} \times \sum_{j=1}^{N} r_j. \tag{4.9}$$

[13]Impact is defined as the rate of deviation over the variable's total data range during the execution. For example, a deviation of 10 on a [0,200] range produces an impact of 0.05.

The distribution $P(\#\text{bits} = x)$ depends on how corruptions in the whole system ultimately affect the numerical data of simulations. Because of the impossibility of calculating this distribution for a system as massive as a supercomputer, four distributions are assumed representing the following four cases (shown in Figure 4.5): (1) the number of bits affected is usually small, with 1 bit being the most common size (for this case, we use a Poisson distribution with $\lambda = 1.0$); (2) all bit sizes are equally probable (i.e., $P(\#\text{bits} = x) = 1/N$); (3) all possible corruptions (2^N) are equally probable (e.g., P$\sim \mathcal{N}(32.5, 13.05)$ for $N = 64$); and (4) the number of bits affected is usually big, with N bits being the most common size (for this case, we use the inverse of distribution (1)).

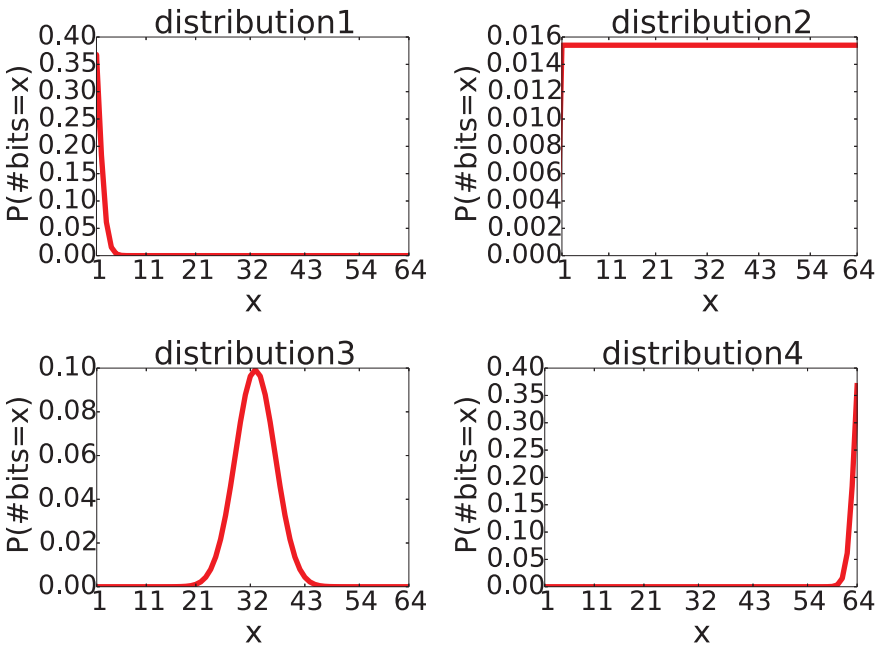


Figure 4.5. Four distributions $P(\#\text{bits} = x)$ for double-precision numbers ($N = 64$)

4.5 Evaluation

This section shows the experimental setup and discusses then the results obtained using the algorithms presented in Section 4.3.

4.5.1 Experimental Setup. For the experiments, four applications from the

FLASH code package [88] are used —Sedov [90], BlastBS [96], Sod [97] and DM-Reflection [98]—representing four different types of explosions. These applications are excellent candidates for testing different SDC detection methods on applications dealing with explosions or collisions. For these applications, the *smoothness* property [86], which is essential to guarantee the success of DAB solutions, is completely lost on the data of those processes through which the different explosions are passing through at that particular moment. Implementations of MPI allowing replication at the process level, such as RedMPI [79], do not yet support partial replication; they support only full replication (i.e., 2x, 3x, etc.); partial replication is simulated by considering precision and recall to be 100% for those processes that are part of the *replication set*.

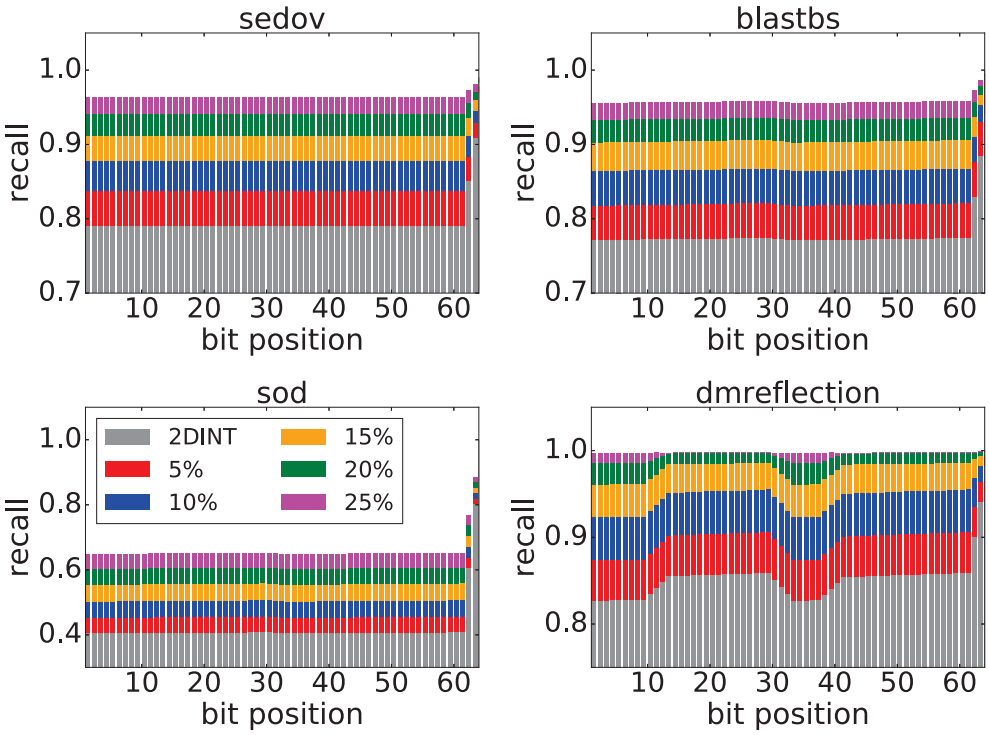


Figure 4.6. Single-bit detection recall results from the injection study. Four applications (Sedov, BlastBS, Sod and DMReflection) running 256 processes are used, setting $w = 100$. Five partial replication rates (5, 10, 15, 20 and 25%) using the fixed budget algorithm are compared with DAB-only nonreplication 2DINT (2D linear interpolation)

Detection recall for each bit position is calculated by averaging the results over hundreds of random injections on the pressure variable in every process over thousands of time steps. For our experiments λ , which controls our dynamic *detection range* δ (see Section 4.2) is set to have exactly zero false positives. In addition, the applications are run using 256 processes, and the data domain is configured to be a two-dimensional grid.

The next two subsections will use only a fixed budget algorihtm. In Section 4.5.2 only spatial overhead (i.e., % of replicated processes) is considered in order to understand how each of the four distributions $P(\#\text{bits}= x)$, as well as different values of the parameter w, affect the detection results. In Section 4.5.3 both temporal and spatial overhead are included, and results for different values of w are again compared. Finally, experiments related to the elastic budget algorithm are presented in Section 4.5.4.

4.5.2 Detection Results. Only the results of those experiments using linear interpolation (spatial) as our predictor are reported. Similar results were obtained using a temporal predictor (QCF), so they are omitted here.

Figure 4.6 presents the results of the injection study. Here, the window w (see Section 4.3.2) is fixed to be 100 time steps. It is possible to see the benefit of using partial replication for improving single-bit detection recall rates. For example, an overall improvement for Sedov from 5% for 5% replication to 20% for 25% replication is observed. For BlastBS it is possible to see an improvement of 6% for 5% replication and of 23% for 25% replication. In the case of Sod, significant gains are obtained, with improvements from 9% for 5% replication to 53% for 25% replication. Finally, for DMReflection one can see an improvement of 6% for 5% replication and of 18% for 25% replication.

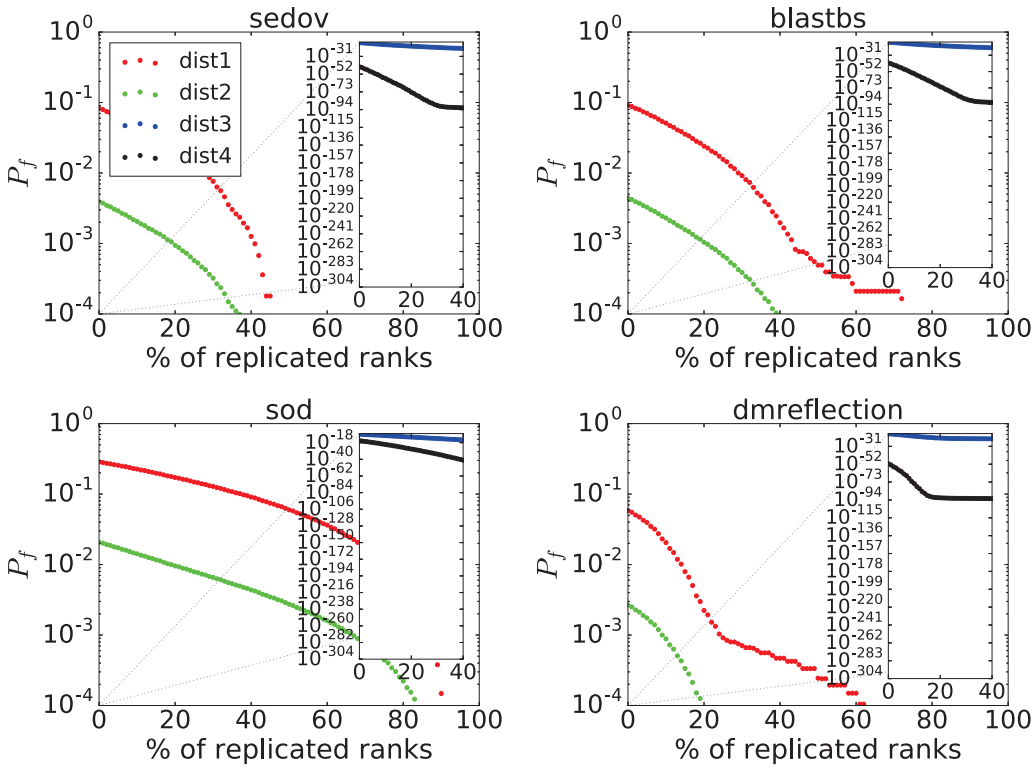


Figure 4.7. Probability of undiscovered corruption when replicating a particular percentage of processes (fixed budget) using the four distributions $P(\#\text{bits}= x)$ described in Section 4.4. Note that the y-axis is plotted using logarithmic scale. The small subplots represent the data zoomed below 10^{-15}

Figure 4.7 presents the results when using the probabilistic evaluation metric P_f with the four distributions presented in Section 4.4. Please note that the y-axis is plotted by using a logarithmic scale. It is possible to categorize the distributions in a spectrum from *difficult* (dist1) to *easy* (dist4) (as discussed in Section 4.4, the fewer the number of bits that can get corrupted, the harder it is to detect corruptions at the software level). In this case, only distributions 1 and 2 are of further interest, since distributions 3 and 4 represent easy detection cases; that is, the probability of undiscovered corruption using the DAB detectors is already below 10^{-15} without even considering replication. For distribution 2, replication of over 20% of the processes is needed in order to achieve a 99.9% protection (i.e., $P_f < 0.001$) level in the case of Sedov, 21% in the case of BlastBS, 67% for Sod, and 9% for DMReflection application.

In the case of distribution 1, over 41% of the processes replicated are needed in order to achieve a 99.9% protection level in the case of Sedov, 44% in the case of BlastBS, 87% for Sod, and only 25% in the case of DMReflection. Recall that distribution 1 is the most *difficult* one, representing an upper bound in the number of replicated processes needed. The author believe that distribution 2, or some distribution between 1 and 2, may be closer to the true one.

Figure 4.8 shows the sensitivity of P_f to the window parameter w. Since we are interested in getting an upper bound on the rate of replication for a given detection recall, only results using distribution 1 for $P(\#\text{bits}= x)$ are shown (again, the hardest for detection). As it is possible to see, the larger the window w, the higher the *probability of undiscovered corruption* P_f. This result is not surprising given that the larger the window w, the higher the probability that the current replication set does not include the ranks with the highest data variability.

In general, partial replication would only be interesting if, for a particular window w, we could get a good enough detection recall (i.e., $P_f < 0.001$) with a tolerable overhead (i.e., at least strictly lower than that of full duplication). In the next subsection it is shown that this is in fact the case for the applications evaluated in this study.

4.5.3 Performance Overhead. Apart from the obvious performance overhead incurred by using replication (i.e., extra hardware needed to run extra processes, or spatial overhead), there is also an overhead introduced by extra messages sent throughout the network, which ultimately enlarges the runtime of applications. The DAB detector has a temporal overhead as it needs to run on every iteration and check all the data points for all the protected variables. From the system's point of view, both dimensions—temporal and spatial—contribute equally to the overall overhead, so both should be included. In partial replication we also need to consider the extra

temporal overhead introduced by process migration when changing the replication set. The total overhead, then, is calculated using the following model:

$$O(r, w) = T(r) \times (r + 1) + M(r, w) \tag{4.10}$$

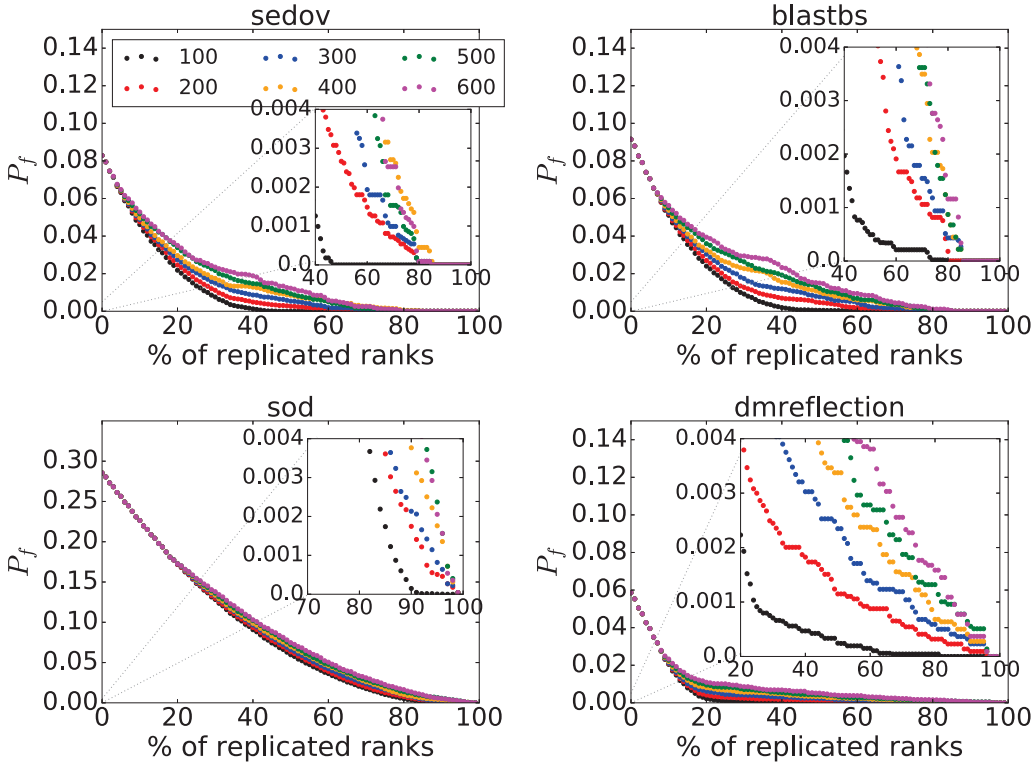


Figure 4.8. Sensitivity of P_f (Figure 4.7) to the window parameter w. In this case, only distribution 1 is used (see Figure 4.5); the small subplots represent the data zoomed between $[0.0, 0.004]$

where r is the replication rate (e.g., 0.5 when replicating half of the processes), $T(r)$ is the runtime overhead introduced by the DAB detector and partial replication when running with a replication rate equal to r (e.g., $T(r) = 1.1$ if there is a 10% increase in running time), and $M(r, w)$ is the overhead introduced by changing the replication set every w steps. Wang et al. [99] show that calculating process migration time as $process\_memory/network\_bandwidth$ is a fairly good estimate. Given this, $M(r, w)$ is calculated as

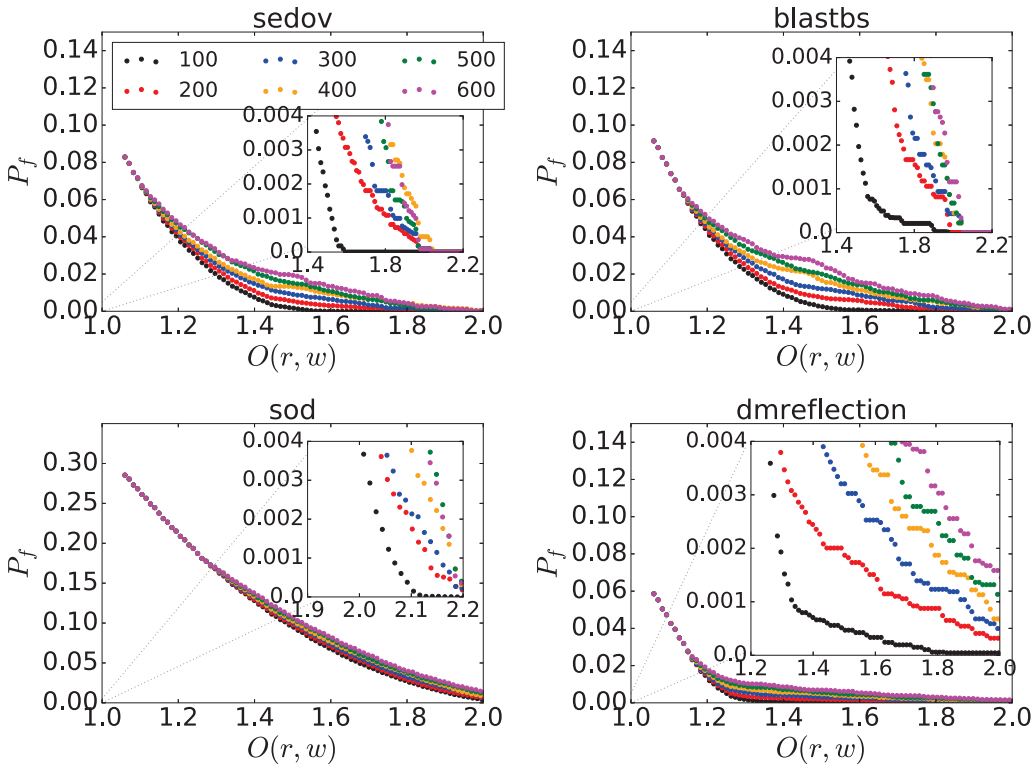


Figure 4.9. Total overhead introduced by partial replication (fixed budget) using different values of w. Distribution 1 is used for $P(\#\text{bits}= x)$; the small subplots represent the data zoomed between $[0.0, 0.004]$

$$M(r, w) = \frac{mem \times r \times n}{W \times T_w}, \tag{4.11}$$

where mem is the memory used per process, $r \times n$ is the replication budget (n is the number of processes), W represents the aggregate network bandwidth in the system, and T_w is the time taken to run w steps in the original application. Note that this is an upper bound, since in some cases the number of processes to replicate is less than the budget, namely, when some processes replicated in the previous window are chosen again for the current one. It is found that in only a few cases does the replication set change completely.

The temporal overhead $T(r)$ may vary depending on the communication-to-

computation ratio of the application. For those cases where computation dominates communication, the extra overhead is usually small. Fiala et al. [79] show that temporal overhead for full duplication (i.e., $r = 1.0$) is not a concern (around 1–2%) for those applications that can maintain a well-balanced communication-to-computation ratio as they scale (applications exhibiting weak scalability). On the other hand, temporal overheads can reach 30% for network-bound applications and kernels. Since partial replication is simulated, it is not possible to measure exactly the value of $T(r)$ for the applications used. In this case, the temporal overhead introduced by the extra network messages is assumed to never be above 5%, given that the stencil codes evaluated are not network-bound. Moreover, the experiments indicate that the temporal overhead introduced by using the DAB detectors is never above 6%[14]. Thus, $T(r = 1.0)$ is set to $T(r = 1.0) = 1.11$ (i.e., 5+6=11% temporal overhead introduced by replicating all processes and using the DAB detector on every process). $T(r)$ for $r < 1.0$ is estimated assuming a balanced communicaton pattern between processes (which is the case in the stencil codes evaluated, where processes communicate mainly with their neighbors): $\hat{T}(r < 1.0) = 1 + r \times 0.05 + 0.06$.

In order to get an idea of how much overhead would be introduced by partial replication, the values of P_f in Figure 4.9 are computed based on $O(r)$, instead of just spatial overhead (i.e., % of replicated processes), for different values of the parameter w. Moreover, distribution 1 is assumed for $P(\#\text{bits}= x)$ (see Figure 4.5) since it is the hardest with respect to detection (see Section 4.4), getting an upper bound on the overhead needed for a particular desired protection level. All the injection experiments are run on the Fusion cluster at Argonne National Laboratory [100], which has an InfiniBand QDR network with a bandwidth of 4 GB/s per link, per

[14]The memory overhead of the DAB detectors is practically 0% given that only spatial-based predictors are used in this study. For that reason, extra memory usage is not included in the overhead calculation.

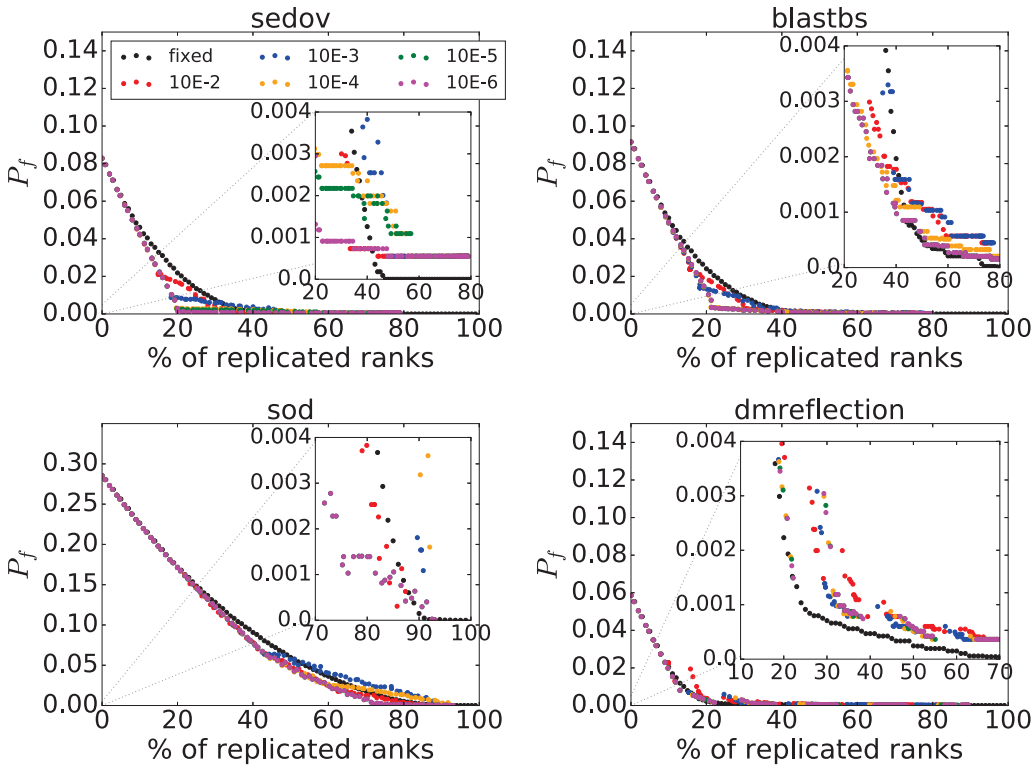


Figure 4.10. Sensitivity of P_f (Figure 4.7) to the parameter Ψ in the elastic budget case, compared with the best solution of the fixed budget algorithm. Distribution 1 is used for $P(\#\text{bits}= x)$; the small subplots represent the data zoomed between $[0.0, 0.004]$

direction, arranged on a fat tree topology. Since network contention issues are not taken into account in the overhead model, W is set to the lowest possible aggregate bandwidth in order to get an upper bound on the effect that the network bandwidth has on the overhead. That is, W is set to W=4 GB/s.

As it is possible to see, the trend between the different curves stays similar to that of Figure 4.8. A window of a 100 time steps is the best choice among all the considered possibilities. The reason is the accuracy loss incurred when using larger window sizes. Even if the temporal overhead can be reduced significantly, the loss in accuracy implies that a bigger rate of replication is needed to cover those ranks for which the lightweight detectors perform poorly. The bigger rate of replication means a much bigger spatial overhead, which overshadows the reduction in temporal overhead

by the smaller frequency of memory transfers throughout the network. The analyses show that it is possible to have a 99.9% protection (i.e., $P_f < 0.001$) with $w = 100$ with a total overhead of around 1.53 (53%) for Sedov, 1.56 (56%) for BlastBS, 2.07 (107%) for Sod, and 1.34 (34%) for DMReflection, with a replication rate of 41%, 44%, 87% and 25% respectively. This is an improvement, over full duplication (considering 5% in temporal overhead due to the extra network messages, full duplication has a total overhead of 2.1, or 110%), of 52% for Sedov, 49% for BlastBS, only 3% for Sod, and 69% for DMReflection, with a detection recall close to 100%.

4.5.4 Elastic Budget. The first step in the evaluation of Algorithm 5 is to set a proper value for the parameter Ψ, which controls how close the MPE of a particular process should be to the MPE of the "worst behaving process" in order to be considered for the elastic budget (as shown in Algorithm 6). The values for Ψ are set to $1/2$, 10^{-1}, 10^{-2}, and so forth, until it is no longer possible achieve better recall results (or recall gets worse). This gives us a value of $\Psi = 10^{-6}$.

Figure 4.10 shows the sensitivity of P_f to different values of Ψ, comparing them to the best solution obtained using the fixed budget algorithm (represented as black curves). We can see from these plots that the added benefit of using elastic varies greatly depending on the application. For example, we can see that it is possible to achieve a $P_f < 0.001$ for the Sedov application replicating only 23% of the processes (on average) instead of the 41% needed in the fixed budget case, a 45% improvement. In the case of BlastBS, however, the gain is smaller: 40% replication instead of 44%. The gain is also modest in the case of Sod, with a 83% replication in the elastic case compared to a 87% in the fixed case. With DMReflection we even lose some recall, needing 33% of replication in the elastic case versus only 25% in the fixed case for the same recall of 99.9% (this may be due to a more uniform distribution of sharp data changes along the time dimension in DMReflection, which may also explain the

good performance of the fixed budget algorithm in the first place).

Nevertheless, and as it will shown in the overhead experiments (Figure 4.12), the gains optained using the elastic budget algorithm in some applications are greater for smaller values of recall. The suitability of one algorithm versus the other depends on the desired protection level.

Figure 4.11 shows the real size of the elastic budget at each time step (and its average) during the full run of the four applications studied in this work. The allocated budget is carefully chosen in order to have at least a 99.9% recall in each application. As we can see, the algorithm is able to use almost all the allocated budget, which means that the mechanism described in Algorithm 5 works well at keeping the actual average close to the allocated budget.

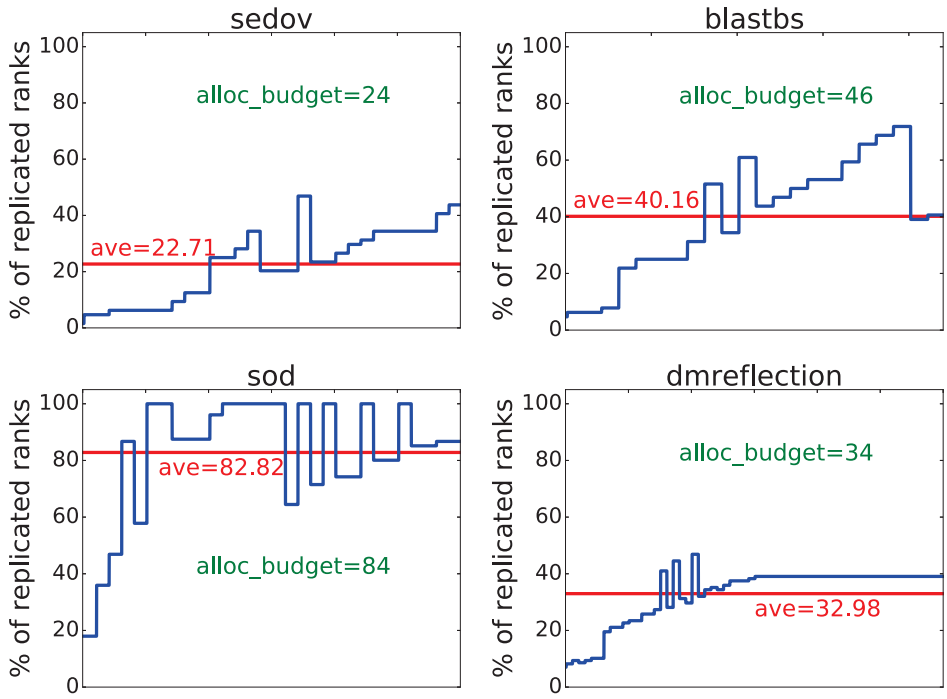


Figure 4.11. Real size of the elastic budget (and average) during the whole execution. The allocated budgets are chosen, per application, in order to have $P_f < 0.001$

Finally, Figure 4.12 compares the best results of fixed budget versus the best results of elastic budget in terms of total overhead as described in Equation (4.10).

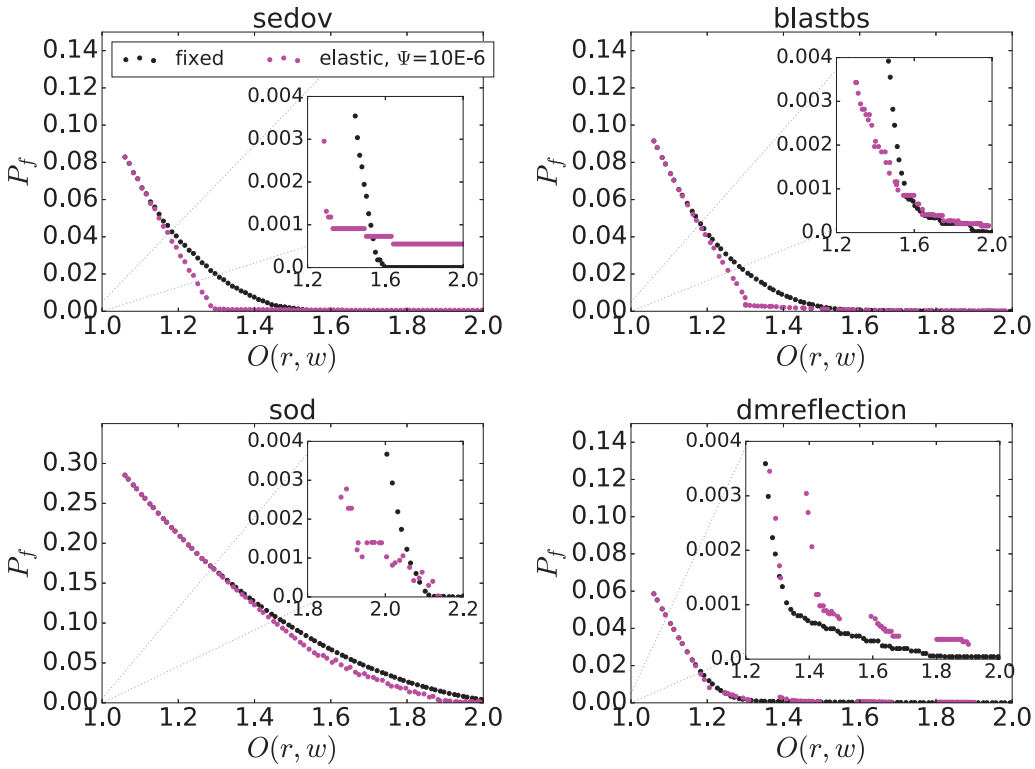


Figure 4.12. Total overhead introduced by partial replication, comparing the fixed budget algorithm with the elastic budget one with $\Psi = 10^{-6}$. Distribution 1 is used for $P(\#\text{bits}= x)$; the small subplots represent the data zoomed between $[0.0, 0.004]$

Again, we see that the curves follow a similar trend to that of Figure 4.10, where results vary greatly by application. If we focus solely on overheads for $P_f < 0.001$, for example, the Sedov application can achieve such a level of recall in the elastic case with an overhead of just 1.33 (33%) instead of 1.53 (53%) in the fixed case; in BlastBS we go from a 1.56 overhead using a fixed budget, to a 1.51 using an elastic one; in the case of Sod, we obtain 2.01 in elastic versus 2.06 in fixed; and in DMReflection we get penalized and the overhead goes from 1.34 in fixed to 1.43 in elastic.

For situations where we do not have such a high demand in recall, however, the elastic-based algorithm can dramatically reduce overhead as compared to the fixed-based one. For example, in the case of a 99% recall ($P_f < 0.01$), there is a reduction in overhead from 1.38 to 1.27 for Sedov, a reduction from 1.4 to 1.3 for BlastBS,

from 1.93 to 1.86 in the case of Sod, and a reduction from 1.23 to 1.21 in the case of DMReflection. For easy comparison, these results are listed in Table 4.3.

Table 4.3. Detection recall and overhead for DAB-only detectors, 2x replication, and the adaptive solution. In the latter, an **f** indicates results using the fixed budget while an **e** indicates results using the elastic budget. Also for the adaptive solution, two cases are shown corresponding to two protection levels: 97% and 99.9% recall, respectively

		DAB-only	**Duplication**	**Adaptive (case 1)**	**Adaptive (case 2)**
Sedov	*Overhead*	6%	110%	**f**=25% **e**=21%	**f**=53% **e**=33%
	Recall	92%	100%	97%	99.9%
BlastBS	*Overhead*	6%	110%	**f**=26% **e**=23%	**f**=56% **e**=51%
	Recall	91%	100%	97%	99.9%
Sod	*Overhead*	6%	110%	**f**=78% **e**=72%	**f**=107% **e**=102%
	Recall	71%	100%	97%	99.9%
DMReflection	*Overhead*	6%	110%	**f**=15% **e**=15%	**f**=34% **e**=43%
	Recall	94%	100%	97%	99.9%

4.6 Related Work

Replication mechanisms for fault tolerance have been studied extensively in the past, especially in the context of aerospace and command and control systems [101, 102, 103, 104]. Traditionally, the HPC community has considered replication to be too expensive to be applicable; and, to the best of my knowledge, it has not been implemented in any real production system. Elliott et al. [105], Engelmann et al. [106], and Stearly et al. [107] show that future exascale HPC systems can benefit from full replication (2x and 3x) because it can reduce the probability of hard failures substantially (i.e., for an application to fail, there needs to be a failure in a rank and all its replicas in a very short window of time). This, in turn, improves utilization by lowering the frequency of checkpointing. Nevertheless, other recent works on improved C/R mechanisms, such as hierarchical C/R, seem to solve this problem without having to duplicate—or triplicate—the resources needed to run an application. In any case, the problem of dealing with hard failures is out of the scope of this work.

Liu et al. [108] propose partial replication *in time* by taking advantage of the fact that soft errors in the first 60% of iterations of some iterative applications are relatively tolerable. The idea is to duplicate all processes only during the last 40% of iterations. Nakka et al. [109], Subasi et al. [110], and Hukerikar et al. [111]—by introducing new programming language syntax—propose to make the programmers responsible for identifying those parts of the code or data that are critical and need to be replicated. In contrast to these solutions, which are application dependent, our work is more general in the sense that we do not require any specific knowledge of tolerability to errors of particular iterations, variables, or code regions.

Partial replication in HPC where processes are chosen at random has also been investigated. Research has shown, however, that such an approach does not pay off [107]. In this work the processes to replicate are chosen based on their data

behavior.

In the realm of GPGPUs, Tan and Fu [112] propose using idle cycles in stream multiprocessors to re-execute wraps. The authors compare their results to discover corruptions. This work aims at protecting application data no matter where it is processed. For example, if the application uses GPGPUs for specific computations, this scheme can be used along with DABFT solutions, since the data computed in the GPGPUs will, at some point, end up affecting the application's state stored in main memory. This same property applies to any other protection mechanisms for specific hardware devices.

4.7 Discussion

In this work I have presented adaptive methods, namely a fixed budget method and an elastic budget method, for general software level silent data corruption detection of parallel applications. These adaptive methods combine partial replication along with DAB detectors to get SDC protection levels that are close enough to those achieved by pure duplication (2x replication) at a lower overhead price. In addition, it has been demonstrated that using an elastic budget for partial replication can be useful for situations where sharp data changes are concentrated not only in a particular place in space but also in time.

An extensive evaluation has also been conducted of the adaptive detection methods against existing data analytic based detection (DAB) and replication in terms ofdetection accuracy and overall overhead. The evaluation is based on four applications dealing with different types of explosions, which are excellent candidates for testing partial replication on applications dealing with explosions or collisions. Moreover, to effectively evaluate various detection methods, an evaluation metric based on the probability that a corruption will pass unnoticed by a particular detector has

been proposed. Instead of evaluating detectors solely on overall single-bit precision and recall rates, which are not enough to understand how well applications are actually protected, this single metric allows us to directly compare SDC detectors against mechanisms with perfect precision and recall, such as full duplication. Results show that the adaptive approach is able to protect the applications analyzed (99.9% detection recall) replicating between 23–83% of all the processes with a maximum total overhead of 33–102% depending on the application (compared with 110% for pure duplication).

While an elastic budget solution provides several benefits over a fixed budget solution in some situations (i.e., when sharp data changes are concentrated not only in a particular place in space but also in time), the elastic solution cannot be directly implemented in the current state-of-the-art HPC systems due to the lack of dynamic resource allocation support on existing HPC platforms. Nevertheless, one can image a scenario in future exascale computing where systems will have spare resources, in our case nodes, which will be allowed to be requested "on the fly" by applications and libraries in order to perform fault tolerance tasks.

CHAPTER 5

CONCLUSION AND FUTURE WORK

The work presented in this thesis has been focused on solving two classical problems that the HPC community has been dealing with for quite some time, although they are turning more pressing as supercomputers continue scaling inexorably toward exascale. The first is the projected dramatic increase [2, 1] in total failures that these systems will experience which will, under the classic C/R model, also increase the frequency of checkpointing. All things being the same, there is the possibility of reaching a point where the drop in utilization rate will make these systems almost useless (see Figure 1.1), i.e., the system will spend more time doing fault tolerance tasks than doing actual valuable computing. The second problem is related to the projected increase in SERs due to the demands imposed upon hardware manufacturers to decrease transistor size as well as energy consumption [15, 16]. When soft errors are not detected and corrected properly, either by hardware or software mechanisms, they have the potential to corrupt the applications' memory state, producing SDCs.

As introduced earlier, solutions to the first problem can be divided into (1) reactive and (2) proactive. Reactive measures deal with the problem *after the fact*; research in this case focuses on what are the best actions to perform in order to decrease the penalty imposed upon utilization when the errors do actually happen. The classic example of this type of strategy in the HPC world is C/R. Proactive solutions, on the other hand, try to predict when and where errors will happen and take action accordingly. The work presented in chapter 2 of this thesis is of this second type.

One of the main contributions of this work in the area of proactive solutions—**PULSE**— is the idea of using environmental data (also known as sensor data) in order to predict failures in supercomputers. Environmental data is composed of numerical

values directly read from the hardware sensors spread all over the system. These sensors report environmental conditions such as motherboard, CPU, GPU, hard disk and other peripherals' temperature, voltage, and/or fan speed. Until this work, the totality of existing studies were based on RAS logs which are designed to be read and understood by humans (e.g., system administrators), and to be read/analyzed after a failure occurs in a forensic kind of fashion. Nevertheless, extensive work exists that uses that type of information successfully to predict failures in supercomputers [11, 12, 13, 14].

Initial work performed as part of this research seemed to suggest that using environmental data, along with a newly designed density-based ML algorithm called Void Search for prediction, was in fact superior to the state of the art prediction using RAS logs. This initial work was conducted using environmental data from the 48-rack IBM BG/Q Mira supercomputer at Argonne National Laboratory, corresponding to the last 4 months of 2012. However, in an effort to extend the study, a new analysis with newer data (the whole year of 2014) from the same system was conducted. In that analysis, it was discovered that the proposed methodology was not performing as good as it was initially thought and, in fact, after careful examination of the sensor data itself, it was discovered that the VS algorithm was only predicting periods of machine shut down (see the discussion is subsection 2.7 for more details).

In spite of the temporary setback, it is of the opinion of the author that work related to failure prediction should continue along these lines, especially since every new generation of supercomputers come with more and more environmental sensors and profiling capabilities.

With respect to the second critical problem (SDC), solutions can also be split into two: (1) hardware-based and (2) software-based. The most widely adopted today are hardware-based due to their generality, i.e., all applications can be equally

protected by them. For example, ECC protects memory words by correcting single bit-flips and detecting double bit-flips. Nevertheless, ECC is unable to protect all levels of the memory hierarchy (such as L1 caches or processor registers) and it is unclear whether these hardware mechanisms will be able to be extended much further moving forward due to energy constraints. On the other hand, software-based solutions are considered either too specific—such as ABFT or Approximate Computing—or too expensive—such as process replication.

In contrast to those, this thesis (chapter 3) introduces a new software-based solution (DAB) which, using some observations related to the nature of the evolution of the data in a large number of scientific applications (more specifically, its *smoothness* in the space and time dimensions), is able to check for data corruptions without needing too much knowledge about each individual application. The only information needed in this case are the variables representing the state of an application at a particular time. These are the variables used to save a checkpoint, i.e., the essential information needed to recreate a particular execution without the need to start all over. The experiments performed show that this mechanism can detect a large number of possible corruptions with a smaller performance penalty than duplication, which can be considered the most general software-based solution (for *software-deterministic applications* only, of course).

Although DAB detection works well with a large number of scientific applications, there are some exceptions—such as applications dealing with explosions or collisions—where the smooth assumption does not hold for some of the variables. Nevertheless, it has been observed that not all processes of parallel applications experience the same level of data variability at exactly the same time; hence, one can *smartly* choose and use more expensive detection methods (such as replication) in only those processes for which DAB detectors would perform poorly. Work is pre-

sented (see chapter 4) that combine the merits of DAB with that of partial replication through an adaptive algorithm that chooses, at particular moments in time, which subset of processes should be replicated. Results show that the adaptive approach is able to protect the applications analyzed (99.9% detection recall) with less overhead than full duplication in all the applications analyzed.

Another open issue is the need to standardize the evaluation of SDC detection methods. Currently, evaluation results are mainly reported in the form of single-bit precision and recall rates. However, it can be argued that this may not be enough to understand how well applications are actually protected. For example, software-based detection performs better the more bits that can get corrupted, as opposed to hardware-based detection (see section 4.4 for a detailed discussion). Having this in mind, this work introduces a single metric to take such observations into account. This single metric gives us a probability that a corruption—no matter in what particular bit(s), and how many—will pass unnoticed by a particular detector. Furthermore, this single metric allows for the comparison with perfect recall (i.e., 100%) methods such as duplication.

The issue of detecting SDCs is a hard, and by far still, open, problem. This thesis contributes to the community by exploring previously uncharted territory in the SDC literature such as DAB detection and partial replication. One of the issues raised by this work is that of the impossibility (for being very hard to calculate analytically) of knowing the distribution of the number of bits that can get corrupted in a memory word, assuming that corruptions can happen anywhere in the system and spread chaotically.

A recent paper [18] tried to approach this issue and calculate this distribution by disabling all hardware-based memory protections during more than a year in 1,080 nodes of an experimental cluster in Barcelona, Spain. They then recorded all the

bit-flip corruptions occurring in the system. Although the study suggests that the majority of the corruptions affect only one or two bits (which would indicate that hardware-based mechanisms are indeed the way to go), it also shows that corruptions affecting a large number of bits, although rare, can not be predicted using detectable errors. In contrast, corruptions affecting just one or two bits can be easily predicted looking into error patterns (usually by looking at previous errors in the same memory module). It is clear then that hardware-based protection is not the only answer moving forward and that general software-based solutions, such as the ones presented in this work, will need to be taken into account if we want to effectively protect HPC applications in extreme scale systems.

BIBLIOGRAPHY

[1] "Addressing failures in exascale computing," *ANL report*, March 2013.

[2] B. Schroeder and G. Gibson, "Understanding failure in petascale computers," *Journal of Physics Conference Series: SciDAC*, vol. 78, p. 012022, June 2007.

[3] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, September 1974.

[4] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," in *41st International Conference on Parallel Processing*, 2012, pp. 148–157.

[5] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. D. Supinski, and R. Eigenmann, "Mcrengine: A scalable checkpointing system using data-aware aggregation and compression," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[6] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka, "Fti: high performance fault tolerance interface for hybrid systems," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–12.

[7] A. Moody, G. Bronevetsky, K. Mohror, and B. R. D. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2010, pp. 1–11.

[8] G. Aupy, Y. Robert, F. Vivien, and D. Zaidouni, "Impact of fault prediction on checkpointing strategies," *Technical Report, INRIA*, vol. 1, no. RR-8023, July 2012.

[9] ——, "Checkpointing algorithms and fault prediction," *Technical Report, INRIA*, no. RR-8237, February 2013.

[10] M. S. Bouguerra, A. Gainaru, F. Cappello, L. B. Gomez, N. Maruyama, and S. Matsuoka, "Improving the computing efficiency of hpc systems using a combination of proactive and preventive checkpointing," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2013.

[11] Z. Lan, J. Gu, Z. Zheng, R. Thakur, and S. Coghlan, "A study of dynamic meta-learning for failure prediction in large-scale systems," *Journal of Parallel and Distributed Computing (JPDC)*, 2010.

[12] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, and S. Ma, "Critical event prediction for proactive management in large-scale computer clusters," in *International Conference on Knowledge Discovery and Data Mining*, 2003, pp. 426–435.

[13] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman, "A practical failure prediction with location and lead time for blue gene/p," in *Proc. of the 1st Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS) in conjunction with DSN*, 2010.

[14] Y. Liang, Y. Zhang, H. Xiong, and R. Shaoo, "Failure prediction in ibm bluegene/l event logs," in *7th International Conference on Data Mining*, 2007.

[15] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *SC'12*, 2012, pp. 57:1–57:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389074

[16] E. Normand, "Single event upset at ground level," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[17] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design," in *ASPLOS'XVII*, 2012, pp. 111–122.

[18] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing : A large-scale study of dram raw error rate on a supercomputer," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[19] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[20] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[21] C. Engelmann, G. R. Vallée, T. Naughton, and S. L. Scott, "Proactive fault tolerance using preemptive migration," in *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009.

[22] G. Lakner and B. Knudson, "Ibm system blue gene solution: Blue gene/q system administrator," *[Online] Available at http://www.redbooks.ibm.com/redbooks/pdfs/sg247869.pdf*, May 2013.

[23] "Managing system software for cray xe and cray xt," *[Online] Available at http://docs.cray.com/books/S-2393-31/S-2393-31.pdf*, 2010.

[24] M. C. Neyrinck, "Zobov: a parameter-free void-finding algorithm," *Monthly Notices of the Royal Astronomical Society*, vol. 386, pp. 2101–2109, 2007.

[25] J. Aikio and P. Maehoenen, "A simple void-searching algorithm," *Astrophysics Journal*, vol. 497, p. 534, April 1998.

[26] E. Platen, R. V. D. Weygaert, and B. J. T. Jones, "A cosmic watershed: the wvf void detection technique," *Monthly Notices of the Royal Astronomical Society*, vol. 380, no. 2, pp. 551–570, 2007.

[27] F. Hoyle and M. S. Vogeley, "Voids in the 2df galaxy redshift survey," *Astrophysics Journal*, vol. 607, pp. 751–764, 2004.

[28] "Mira information at argonne leadership computing facility," *[Online] Available at http://www.alcf.anl.gov/mira*.

[29] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim, "The ibm blue gene/q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, March-April 2012.

[30] V. Icke and R. V. D. Weygaert, "Fragmenting the universe," *Astronomy and Astrophysics*, vol. 607, pp. 751–764, 2004.

[31] J. H. Holland, "Adaptation in natural and artificial systems," *Univ. of Michigan Press*, 1975.

[32] M. Srinivas and L. M. Patnaik, "Genetic algorithms: a survey," *Computer*, vol. 27, no. 6, pp. 17–26, June 1994.

[33] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan, "Practical online failure prediction for blue gene/p: Period-based vs event-driven," in *Proc. of PFARM workshop*, 2011.

[34] H. Hotelling, "Analysis of a complex of statistical variables into principal components," *Journal of Educational Psychology*, vol. 24, pp. 417–441 and 498–520, 1933.

[35] E. Berrocal, L. Yu, S. Wallace, M. E. Papka, and Z. Lan, "Exploring void search for fault detection on extreme scale systems," in *IEEE Cluster (Best Paper Award)*, 2014.

[36] T. Hastie, R. Tibshirani, and J. Friedman, in *The Elements of Statistical Learning, Springer, Second Edition.*

[37] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys*, vol. 41, no. 3, July 2009.

[38] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 28, p. 100, 1979.

[39] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[40] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Computing Surveys*, vol. 42(3), no. 10, March 2010.

[41] A. Oliner and J. Stearley, "What supercomputers say: A study of five systems logs," in *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2007.

[42] R. Ren, X. Fu, J. Zhan, and W. Zhou, "Logmaster: Mining event correlations in logs of large scale cluster systems," in *Proc. of IEEE Symp. On Reliable Distributed Systems*, 2010.

[43] B. Schroeder and G. Gibson, "A large-scale study of failures in hpc systems," in *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2006.

[44] N. Taerat, N. Naksinehaboon, C. Chandler, J. Elliott, C. Leangsuksun, G. Ostrouchov, and S. L. Scott, "Using log information to perform statistical analysis on failures encountered by large-scale hpc deployments," in *High Availability and Performance Computing Workshop (HAPCW)*, 2008.

[45] M. Gabel, A. Schuster, R. Bachrach, and N. Bjorner, "Latent fault detection in large scale services," in *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2012.

[46] M. Kasick, J. Tan, R. Gandhi, and P. Narasimhan, "Black-box problem diagnosis in parallel file systems," in *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[47] Z. Lan, Z. Zheng, and Y. Li, "Toward automated anomaly identification in large-scale systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 2, 2010.

[48] G. Bronevetsky, I. Laguna, S. Bagchi, B. de Supinski, D. H. Ahn, and M. Schulz, "Automatic fault characterization via abnormality-enhanced classification," in *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2012.

[49] L. Yu, Z. Zheng, Z. Lan, T. Jones, J. M. Brandt, and A. C. Gentile, "Filtering log data: Finding the needles in the haystack," in *International Conference on Dependable Systems and Networks (DSN)*, 2012.

[50] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, pp. 10–16, Nov. 2005. [Online]. Available: http://dl.acm.org/citation.cfm?id=1108266.1108285

[51] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design," in *ASPLOS'XVII*, 2012, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150989

[52] S. Krishnamohan and N. R. Mahapatra, "Analysis and design of soft-error hardened latches," in *GLSVLSI'05*, 2005, pp. 328–331. [Online]. Available: http://doi.acm.org/10.1145/1057661.1057740

[53] S. Di, E. Berrocal, L. Bautista-Gomez, K. Heisey, R. Guptal, and F. Cappello, "Toward effective detection of silent data corruptions for hpc applications," ser. SC '14 - poster, 2014.

[54] L. A. Bautista-Gomez and F. Cappello, "Detecting silent data corruption through data dynamic monitoring for scientific applications." in *PPoPP'14*, 2014, pp. 381–382.

[55] X. Xu, "Large eddy simulation of compressible turbulent pipe flow with heat transfer," *Doctorate Thesis, Iowa State University*, 2003.

[56] J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland, "Speeding up nek5000 with autotuning and specialization," in *ICS'10*, 2010, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810120

[57] A. Dubey, K. Antypas, A. Calder, B. Fryxell, D. Lamb, P. Ricker, L. Reid, K. Riley, R. Rosner, A. Siegel, F. Timmes, N. Vladimirova, and K. Weide, "The software development process of flash, a multiphysics simulation code," in *SE-CSE 2013: The 2013 International Workshop on Software Engineering for Computational Science and Engineering*, 2013, pp. 1–8. [Online]. Available: http://flash.uchicago.edu/site/publications/HostedPapers/icsews13secse-id3-p-16586-preprint.pdf

[58] S. Habib, V. A. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. A. Insley, D. Daniel, P. K. Fasel, N. Frontiere, and Z. Lukic, "The universe at extreme scale: Multi-petaflop sky simulation on the bg/q," in *SC'12*, 2012, pp. 1–11.

[59] B. Fryxell, K. Olson, P. Ricker, F. Timmes, M. Zingale, D. Lamb, P. MacNeice, R. Rosner, J. Truran, and H. Tufo, "Flash: an adaptive mesh hydrodynamics code fro modeling astrophysical thermonuclear flashes," *ApJS*, vol. 131, no. 273, 2000.

[60] L. Sedov, "Similarity and dimensional methods in mechanics. Translated from 4th Russian edition by M. Friedman. Edited by M. Holt." New York-London: Academic Press. 363 p. (1959)., 1959.

[61] Paul Ricker, "FLASH Code for Sedov Explosion," http://flash.uchicago.edu/ ricker/research/codes/flash/test_gallery/sedov.html.

[62] G. A. Sod, "A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws," *Journal of Computational Physics*, vol. 27, no. 1, pp. 1–31, 1978.

[63] Paul Ricker, "FLASH Code for Sod Problem," http://flash.uchicago.edu/ ricker/research/codes/flash/test_gallery/sod.html.

[64] T. Semiconductor, "Soft errors in electronic memory - a white paper," 2004.

[65] Cataldo, "Mosys, iroc target ic error protection," 2002. [Online]. Available: http://www.eetimes.com/story/OEG20020206S0026

[66] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *HPCA'05*. IEEE, 2005, pp. 243–247.

[67] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," *IBM Microelectronics Division*, pp. 1–23, 1997.

[68] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *SC'12*, 2012, pp. 78:1–78:12.

[69] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *ISCA'02*, 2002, pp. 99–110.

[70] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 518–528, 1984.

[71] Z. Chen, "Online-abft: an online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *PPoPP'13*. ACM, 2013, pp. 167–176.

[72] A. R. Benson, S. Schmit, and R. Schreiber, "Silent error detection in numerical time-stepping schemes," *International Journal of High Performance Computing Applications*, pp. 1–20, 2014.

[73] N. W. Hengartner, E. Takala, S. E. Michalak, and S. A. Wender, "Evaluating experiments for estimating the bit failure cross-section of semiconductors using a colored spectrum neutron beam," *Technometrics*, vol. 50, no. 1, pp. 8–14, Feb. 2008.

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

[75] S. Borkar, "Major challenges to achieve exascale performance," *Intel Corp.*, April 2009.

[76] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *HPCA'05*, 2005.

[77] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," in *IBM Microelectronics Division*, 1997, pp. 1–23.

[78] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 397–404, September 2005.

[79] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *SC'12*, 2012, pp. 78:1–78:12.

[80] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in *ISCA'02*, 2002, pp. 99–110.

[81] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 518–528, 1984.

[82] A. R. Benson, S. Schmit, and R. Schreiber, "Silent error detection in numerical time-stepping schemes," *International Journal of High Performance Computing Applications*, pp. 1–20, 2014.

[83] K. S. Yim, "Characterization of impact of transient faults and detection of data corruption errors in large-scale n-body programs using graphics processing units," in *IPDPS'14*, 2014, pp. 458–467.

[84] T. Chalermarrewong, T. Achalakul, and S. C. W. See, "Failure prediction of data centers using time series and fault tree analysis," in *ICPads'12*, 2012, pp. 794–799.

[85] L. A. Bautista-Gomez and F. Cappello, "Detecting silent data corruption through data dynamic monitoring for scientific applications," in *PPoPP'14*, 2014, pp. 381–382.

[86] S. Di, E. Berrocal, and F. Cappello, "An efficient silent data corruption detection method with error-feedback control and even sampling for hpc applications," in *CCGRID'15*, 2015.

[87] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for hpc applications," in *HPDC'15 (short paper)*, 2015.

[88] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo, "Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes," *The Astrophysical Journal Supplement Series (ApJS)*, vol. 131, pp. 273–334, 2000.

[89] L. A. Bautista-Gomez and F. Cappello, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," in *1st International Workshop on Fault Tolerant Systems (part of Cluster'15)*, 2015, pp. 595–602.

[90] L. I. Sedov, *Similarity and Dimensional Methods in Mechanics (10th Edition)*. New York: Academic Press, 1959.

[91] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Exploring partial replication to improve lightweight silent data corruption detection for hpc applications," in *EuroPar'16*, 2016.

[92] Z. Chen, "Online-ABFT: an online algorithm based fault tolerance scheme for soft error detection in iterative methods," in *ACM PPoPP'13*, 2013, pp. 167–176.

[93] "Nek5000 project," *[online] Available at https://nek5000.mcs.anl.gov*.

[94] P. Fisher, "Nek5000 user guide," *[online] Available at http://www.mcs.anl.gov/˜fischer/nek5000/examples.pdf*.

[95] O. Walsh, "Eddy solutions of the Navier-Stokes equations," in *Proceedings of the Navier-Stokes Equations II - Theory and Numerical Methods*, 1991, pp. 306–309.

[96] A. L. Zachary, A. Malagoli, and P. Colella, "A higher-order godunov method for multidimensional ideal magnetohydrodynamics," *SIAM Journal of Scientific Computing*, vol. 15, no. 2, pp. 263–284, 1994.

[97] G. A. Sod, "A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws," *Journal of Computational Physics (JCP)*, no. 27, pp. 1–31, 1978.

[98] P. Colella and P. R. Woodward, "The piecewise parabolic method (ppm) for gas-dynamical simulations," *Journal of Computational Physics (JCP)*, no. 54, pp. 174–201, 1984.

[99] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Proactive process-level live migration in hpc environments," in *SC'08*, 2008.

[100] "Fusion cluster at Argonne National Laboratory," *[online] Available at http://http://www.lcrc.anl.gov/guides/Fusion*.

[101] D. Briere and P. Traverse, "AIRBUS A320/A330/A340 electrical flight controls – a family of fault-tolerant systems," in *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, 1993, pp. 616–623.

[102] H.-P. D. Company, "HP integrity nonstop computing," *[online] Available at http://h20223.www2.hp.com/nonstopcomputing/cache/76385-0-0-0-121.aspx*.

[103] M. Pignol, "How to cope with SEU/SET at system level," in *Proceedings of the IEEE International On-Line Testing Symposium*, 2005, pp. 315–318.

[104] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," in *Proceedings of the IEEE Aerospace Applications Conference*, 1996, pp. 293–307.

[105] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining partial redundancy and checkpointing for hpc," in *32nd IEEE International Conference on Distributed Computing Systems*, 2012, pp. 615–626.

[106] C. Engelmann, H. Ong, and S. L. Scott, "The case for modular redundancy in large-scale high performance computing systems," in *PDCN*, 2009, pp. 189–194.

[107] J. Stearly, K. Ferreira, D. Robinson, J. Laros, K. Pedretti, D. Arnold, P. Bridges, and R. Riesen, "Does partial replication pay off?" in *DSN'12*, 2012.

[108] J. Liu, M. C. Kurt, and G. Agrawal, "A practical approach for handling soft errors in iterative applications," in *Cluster'15*, 2015, pp. 158–161.

[109] N. Nakka, K. Pattabiraman, and R. Iyer, "Processor-level selective replication," in *DSN'07*, 2007, pp. 544–553.

[110] O. Subasi, J. Arias, O. Unsal, J. Labarta, and A. Cristal, "Programmer-directed partial redundancy for resilient hpc," in *CF'15*, 2015.

[111] S. Hukerikar, P. C. Diniz, R. F. Lucas, and K. Teranishi, "Opportunistic application-level fault detection through adaptive redundant multithreading," in *HPCS'14*, 2014.

[112] J. Tan and X. Fu, "Rise: Improving the streaming processors reliability against soft errors in gpgpus," in *PACT'12*, 2012.