Chapter 2
Errors and Faults

Ana Gainaru and Franck Cappello

Abstract Understanding the behavior of failures in large-scale systems is important in order to design techniques to tolerate them. Reliability knowledge of resources can be used in numerous ways by scientist of systems administrators: (1) it can be used to improve the quality of service of the machine; (2) to reduce performance loss due to unexpected failures either by reliability-aware scheduling or by reliability-aware checkpointing; (3) to design more resilient applications, programming models or machines in the future. This chapter focuses on offering an overview of failures observed in real large-scale systems and their characteristics, with an emphasis on modeling, detection, and prediction.

2.1 Introduction

As large-scale systems evolve toward post-Petascale computing to accommodate applications' increasing demands for computational capabilities, many new challenges need to be faced, among which fault tolerance is a crucial one.

The number of system components in current and future supercomputers increases faster than component reliability. In the future, projections show larger systems with even more failure-prone hardware and more complex system and application codes. Near threshold logic is considered as a candidate technology for future system. It has advantages in power consumption but it increases error rates. Even in classic CMOS technology, soft errors can cause one or multiple bits to spontaneously flip to the opposite state, due to multiple reasons such as alpha particles from package decay or cosmic rays. Although techniques such as error correcting codes (ECCs) have been implemented in memory, in reality, some bit flips still manage to pass undetected.

A. Gainaru (✉)
NCSA, University of Illinois at Urbana-Champaign, Champaign, USA
e-mail: againaru@ncsa.illinois.edu

F. Cappello
Argonne National Laboratory, Lemont, USA
e-mail: cappello@mcs.anl.gov

© Springer International Publishing Switzerland (outside the USA) 2015

T. Herault and Y. Robert (eds.), *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks,
DOI 10.1007/978-3-319-20943-2_2

Moreover, processor caches are not protected by ECC in general. In addition, the constant need to reduce component size and voltage, limits the use of soft-error mitigation techniques. The overall consequence is a decreasing Mean Time Between Failures (MTBF) for future extreme-scale systems.

Failures in supercomputers are assumed to be uniformly distributed in time. However, recent studies show that failures in high-performance computing systems are partially correlated in time, generating periods of higher failure density. Understanding the inter-arrival patterns of failures is crucial in optimizing current fault tolerance approaches and decreasing the impact of failures on execution time to a minimum. This chapter provides characterization of failures and their pattern in extreme-scale computers with a focus on modeling, detection, and prediction. We present currently used detection mechanisms for several national laboratories in the US as well as state-of-the-art research for more sophisticated mechanism in Sect. 2.3. Failure detection is valuable for system management, replication, load balancing, and other maintenance services. The inter-arrival distribution of failures has been the study of many research programs. Failure modeling is an important research direction used in guiding reliability-aware resource allocation and optimizing fault-tolerant protocols in order to minimize the performance loss due to failures. We present the most recent findings and their impact on resiliency protocols in Sect. 2.5.

There are important lessons to be learned form the statistical information of failures and events generated by current large-scale systems (Sect. 2.4). These lessons will guide the design of future extreme-scale platforms and can be used to predict the direction of improvements in technological solutions for future system and application software development. Finally failure prediction is a field that complements current resiliency methods and has the potential of improving the performance of fault-tolerant protocols. A survey of prediction methods (starting from methods that assess the future reliability of a system to methods that pinpoint the exact time and space occurrence of the next failure) and their impact on checkpointing is presented in Sect. 2.6.

2.2 Definitions

The absence of consistent definitions and metrics for supercomputer reliability, availability and serviceability has hindered meaningful collaborations in the community [72]. In order to avoid this problem, the workshop organized by the Institute for Computing Sciences on August 2012 proposed a taxonomy of terms to be used as standard. The definitions that were proposed are based almost entirely on [5]. We will use the terms as defined in the workshop's report [71]. This section enumerates the most important ones.

Resilience is defined as the ability of a system to keep applications running and maintain an acceptable level of service in the face of transient, intermittent, and permanent faults. The term "fault tolerance" refers to the ability of a system to continue performing its intended function properly in the face of faults.

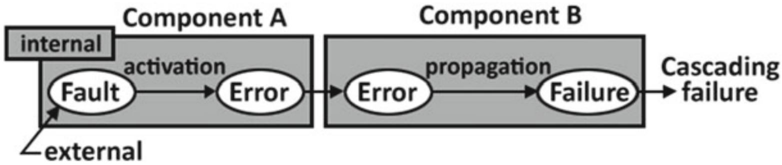

Fig. 2.1 Error propagation and cascading failures

Figure 2.1 shows the propagation chain from faults to failures in a system. A fault represents the cause of an error, like a bit flip due to an alpha particle. An error is the part of total state that might lead to a failure and the failure is a transition to incorrect service. Faults can be active or inactive, depending on whether they cause errors or not. For example, a software bug that is never exercised is called inactive while a bit flip in the processor cache that leads to an application crash is called an active fault. In general, a fault is local to one component, either software or hardware, while errors and failures may propagate from one component to another. In case of failures, this propagation is called cascading failures.

More generally, a failure can be defined as the event that occurs when the service delivered deviates from the correct service operation or when at least one external state of the system deviates from the correct service state. Faults may be caused by complex combinations of internal states and external conditions that occur rarely and are difficult to reproduce.

By error identification, we mean the process of discovering the presence of an error but without necessarily identifying which part of the system state is incorrect, and what fault caused this error. By definition, every fault causes an error. Almost always, the fault is detected by detecting the error the fault caused. Therefore, fault detection or error detection often refers to the same thing. Latent or silent errors are errors that are not detected.

There are several means of dealing with faults divided in four separate classes:

- Tolerance is used to avoid failures in the presence of faults
- Removal is used to reduce the fault number and severity
- Forecasting is used to estimate the present number, future incidence and likely consequences of faults
- Prevention is used to prevent fault occurrences

The term “Time to Failure” or TTF represents the interval between the end of the last failure and the beginning of the consecutive failure. Time between Failures (TBF) represents the interval between the beginnings of two consecutive failures and the Time to Repair (TTR) is synonymous with the Downtime and it defines the interval between the beginning of a failure and its end. Formally:

MTBF (Mean time between failures) = $\frac{TotalTime}{NumFailures}$

MTTF (Mean time to failure) = $\frac{Uptime}{NumFailures}$

MTTR (Mean time to repair) = $\frac{UnscheduledDowntime}{NumFailures}$

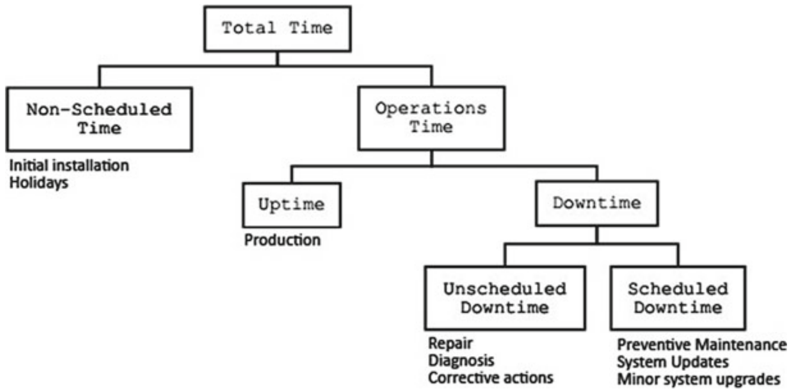

Fig. 2.2 Diagram of possible states in a supercomputer's life

An incredibly dense state diagram would be required to characterize all possible states for a supercomputer lifespan and its workload. For the purpose of resiliency, we will consider the diagram in Fig. 2.2. The diagram is a simplified version of the one proposed by [72].

The nonscheduled time represents the interval when the system is not scheduled to be utilized by production or system engineering users. Otherwise, the system is considered in its operation times. This interval includes production time or unscheduled and scheduled downtime. The unscheduled downtime occurs when a system is not available due to unplanned events, like failures, while the scheduled downtime occurs when the system is not available due to planned events, for example system testing in order to verify that a component is functioning properly.

Prediction is defined as the activity of estimating the presence of a failure. A prediction where the predicted failures occurred in the given time interval and on the given location (or set of locations) is called a true positive. A false positive happens when the predicted failure either does not occur in the given time frame or the predicting location is wrong. Failures that occur without being predicted are false negatives. These three values define the three metrics that we use to measure the performance of a failure predictor: precision, recall and F-measure. Precision defines the quality of all the predictions made by the method and it is equal to the ratio of true positives to the total of number of predictions made. Recall represents the coverage of a predictor and defines the ratio of failures predicted to the total failures in the system. A measure that combines precision and recall is the harmonic mean of the two, called F-measure.

2.3 Detection

In order for management systems to be able to analyze the characteristics of failures, to try to reduce the fault number and severity or to implement forecasting and prevention mechanisms, the system needs to be able to detect failures. In general, HPC

system vendors integrate management systems that aim at ensuring job completion and providing system administrator with a way to check the sanity of the machine. Examples of such management systems are Cray's Node Health Checker or IBM's Cluster Systems Management.

Cray's node health checker is automatically invoked by the scheduler upon the termination of an application. The scheduler gives the monitoring system a list of compute nodes associated with the terminated application on which the node health checker performs specified tests to determine if the given compute nodes are healthy enough to support running subsequent applications. If a node fails one of the tests, it gets removed from the available resource pool. In addition, subsequent monitoring systems are deployed for other components as well. For example, the Gemini interconnect technology used by Cray, has hardware and software support that allows the system to handle certain types of failures without requiring a system reboot. Another example is the CLFS Lustre Monitor tool that implements detection and fault-tolerant methods in order to keep the file systems available in the event of a Meta Data Server (MDS) and/or Object Storage Server (OSS) failures.

Similarly, IBM's Cluster System Management (CSM) provides automatic error detection through heartbeats that is implemented in conjunction with problem avoidance, resolution, and recovery. Disk-heartbeat networks work by exchanging heartbeat messages on a reserved portion of a shared disk. Moreover, current HPC systems implement various optimization methods that provide a way of reducing the time it takes for a node failure to be sent throughout the cluster. For example, IBM's CSM uses disk-heartbeat so that the node that fails puts a departing message on a shared disk so its neighbors will be immediately aware of the node failure (without waiting for missed heartbeats).

There are some failure situations in which heartbeat monitoring cannot determine what exactly failed. For example, lets imagine a cluster node failure due to a problem in a critical hardware component such as a processor. The whole machine can go down without giving the cluster resource service on that node an opportunity to notify other cluster nodes of the failure. The other nodes can only see a failure in the heartbeat monitoring. They are unable to know if it was due to a node failure or a failure in some part of the communication path (for example a router or an adapter).

More advanced node failure detection methods are provided outside of the vendors management systems. These solutions are implemented by system administrators and researchers and are used to reduce the number of failure scenarios which result in system partitions. In general, failures that are not detected at the system level but that propagate and crash an application, are analyzed post-mortem by system administrators. Rules and alerts are afterwards created in order to capture future occurrences of the same problem. This process is, in general, manual and requires a considerable effort.

There are several general methods used in the literature in order to detect failures in an automatic way, from modeling the hardware system and finding outliers in this model, to implementing fault detection at the application and programming model levels. We will briefly present, in this section, a few examples from each method. Most studies are not integrated in any production machine, but some are used as external tools by several national laboratories.

Firstly, there are external approaches for detecting a node failure by using a similar method offered by HPC vendors, namely based on the use of heartbeats, as a way of constantly monitoring a system. These are, in general, hardware health monitoring methods (e.g., IPMI an open standard hardware management specification that defines a set of common interfaces to hardware and firmware). For example, some network failures partition the filesystem into two or more groups of nodes that can only see the nodes in their group. These types of failures can be easily detected through a hardware heartbeat protocol. Software health monitoring [61] systems are also implemented on several large-scale systems by using timeouts to detect node problems.

One method used extensively in the past was to measure each node's behavior and compare it to all other nodes executing similar workloads. An event is categorized as a failure in case of a significant deviation [73, 88]. The method can be applied on performance metrics as well as log entries. After recording performance metrics at a fixed time interval from various components in the system, this information can then be aggregated into a single large matrix. Similarly, for system logs, homogeneous nodes correctly executing similar workloads will tend to generate almost identical logs. The same matrix as for performance metrics can be created by indexing the logs and using nonzero values in the matrix to indicate how many times word i occurs during hour j. By normalizing and performing PCA (principal component analysis), the methods are able to determine anomalies in these matrices. Such a method has been used by researchers at the Sandia National Laboratory in order to complement the available software integrated on their Spirit cluster. The results are in general good and show that the method is able to detect several known fault conditions. On Sandia's 512-node "Spirit" Linux cluster, the detection algorithm was able to localize 50 % of faults with 75 % precision.

Another similar type of method models the components and their interactions and then monitors the model. Most examples are using pattern recognition algorithms [65, 83] to model the system. Others include context free grammars [13] and mathematical equations [2]. Some methods are better suited for analyzing performance metrics (like regression models), while other can be applied on both log files and performance metrics. Some analyze the entire system, while others focus on a specific component. For example, Markov models can be used to implement network failure symptom detection and event correlation discovery. The failure detection's results, when applying the method on the entire system, are decent but less impressive than using the previous method. However, when focusing on one component, specifically networks, path analysis contributes to the discovery of the system structure and the detection of subtle behavioral differences across system versions. Path analysis is an extension of regression methods that estimates the magnitude and significance of causal connections between sets of variables. The method complements existing failure detection methods offered by vendors. Similarly, several studies use, as their core abstraction, runtime paths followed by requests as they move through the system. Based on this, they characterize component interactions. Automated statistical analysis of multiple paths allows to detect and put a diagnosis on complex application level failures.

In the Clouds field, online failure detection is done by using entropies [80]. The algorithm works in three phases: metric collection, entropy time series construction, and entropy time series processing like spike detection, signal processing, or subspace methods, in order to find anomaly patterns. The method uses performance metrics by collecting values from all components on each physical level of the system's hierarchy. A leaf component collects raw metric data from its local sensors. A nonleaf component collects not only its local metric data but also entropy time series data from its child nodes. The method is not integrated in production at this point, but the preliminary results on smaller systems are extremely good showing a 90 % recall with an 80 % precision.

More specific methods focus on the software stack of an HPC system. One example is Rani et al. [61], where the authors propose a fault-tolerant approach that provides the ability to detect and self-recover the parallel runtime environment in cases of compute node failure. Their solution consists of a lightweight heartbeat protocol (BHB) that addresses the scalability issues in system monitoring and failure detection. Their focus is common fault tolerance issues in large scale systems, especially due to permanent component failure.

Application level failure detection is in general used for large servers or for large web application. For example, in [42], the authors present a generic framework for using statistical learning techniques to detect and localize likely application-level failures in component-based Internet services. In the HPC field, Kharbas et al. [41] propose a study on generic fault detection capabilities at the MPI level. The authors implement multiple detectors at various layers of the software stack: at the MPI communication layer and a separate one as stand-alone processes across nodes.

2.4 Observations

The design of extreme-scale platforms that are expected to reach Exascale in the next several years will represent an improvement in technological solutions and will push the boundary of algorithm and application software development. The precise details of these new designs are not yet known. However, there are numerous papers that look at the configurations and properties of existing systems and make predictions regarding the future of HPC systems. There are important lessons to be learned form the statistical information of failures and events by analyzing generated log files and performance/environmental metrics from current systems.

There are several Exascale/Petascale reports that focus on presenting directions for resiliency and programming models for future Exascale systems. The DARPA white paper on system resilience at extreme scale from 2008 [21] points out that current high-end systems waste in average 20 % of their computing capacity on failure and recovery. The paper outlines possible research in order to bring this number down to 2 %, but current methods are not yet at this point. The DOD/DOE report issued in 2009 [16] identifies resilience as a major emerging issue for HPC. It proposes research in five thrust areas: theoretical foundations, enabling infrastructure, fault

prediction and detection, monitoring and control and end-to-end data integrity. The paper published in the International Journal of High Performance Computing Applications in 2009 [11] describes the challenges resiliency faces in the Exascale era and possible directions in order to address these needs. The DOD/DOE report [81] issued in 2012 identifies six high priorities: fault characterization, detection, fault-tolerant algorithms, fault-tolerant programming models, fault-tolerant system services and tools. The DOE workshop from 2012 [78] describes the required HPC resilience for critical DOE mission needs and details what HPC resilience research is already being done at the DOE national labs and what is expected to be done by industry and other groups. Also, the workshop focused on determining what fault management research is a priority for DOEs Office of Science and NNSA over the next five years. The Exascale report from March 2013 [71] gathered the main points discussed at the workshop organized by the Institute for Computing Sciences on August 2012. The report analyzes the state of resiliency for HPC and proposes three designs for approaches in this field: (1) business as usual where the global checkpoint/restart is used; (2) system-level resilience where vendors do not provide sufficiently low SDC rates at an acceptable acquisition and operation cost and a combination of hardware and software technologies is needed to hide the increased failure rates from the application; (3) application-level resilience for which there is an assumption that application codes will need to be modified in order to handle the increased failure rate. The paper makes a couple of recommendations for each design in order for them to become solutions for future systems.

The International Exascale Software Project (IESP) Workshop [19], held in Kobe, Japan on April 12–13, 2012 discussed what will be the major obstacles that the climate community will face at Exascale and proposed and evaluated possible ways to overcome these obstacles. The focus of the workshop was on node-level performance, scalability and resilience. The European Exascale Software Initiative EESI2 [62] is a collaborative project that aims to build and consolidate a vision and roadmap at the European level including applications both from academia and industry to address the challenge of performing scientific computing on the new generation super-computers. In September 2013, the project released the report on the first technical workshop where experts in the areas of software development, performance analysis, applications knowledge, funding models and governance aspects in High Performance Computing provided recommendations and roadmaps for the future of HPC in Europe.

Continuous availability of HPC systems has become a primary concern with the continuous increase of system size to thousands of components. Understanding the behavior of failures in current systems is increasingly important in order to design more reliable systems. To this extent, failure data analysis of current HPC systems can serve three purposes. Firstly, it can highlight dependability bottlenecks and might serve as a guideline for designing more reliable systems in the future. Moreover, real data can be used to develop performance models and simulations, which are an essential part of reliability engineering. As we will see in the following sections, these models can be used to predict node availability, which is useful for resource characterization and scheduling. Reliability knowledge of resources can reduce per-

formance loss due to unexpected failures, and can improve QoS (Quality of Service) either by reliability-aware scheduling, where the systems allocate a priority job to get maximum reliability or by reliability-aware checkpointing where the optimal checkpointing interval can be computed based on the reliability of a set of nodes (see Chap. 1, and Sect. 1.3 in particular).

There are several papers that study the statistics of the data, including the root cause of failures, the mean time between failures, and the mean time to repair. Work on characterizing failures in computer systems differs in the type of data used; the type and number of systems under study; the time of data collection; and the number of failure or error records in the data set. Most of these statistics are based on reliability, availability and serviceability (RAS) data mainly provided by major HPC laboratories and centers in the USA: Los Alamos National Laboratory (LANL), National Energy Research Scientific Computing Center (NERSC), Pacific Northwest National Laboratory, Sandia National Laboratory (SNL), Laurence Livermore National Laboratory (LLNL), and the National Center for Supercomputing Applications (NCSA).

Most studies divide failures into two major categories: software and hardware, each having separately different subcategories. Reliability monitoring and analysis considers failures that affect a single node and also failures affecting a group of nodes. The studies also look at failures that may affect applications or important services. Table 2.1 presents an overview of the categories used in literature when looking at the broad overall image of a system.

Table 2.2 presents a summary of the current studies presenting failure statistics for different machines. Depending on the study and on the analyzed system, the table shows a wide range of results. There is not a consistent main root cause of failures among all systems, nor a consistent MTBF or mean time to repair. However, by looking at the overall view including all systems there are several observations to be made.

On average, for the biggest Petascale systems, there are failures of any type once every 7–10 h, while the systems suffer system-wide outages (SWO), in general once every week. The MTBF has continued to decrease from one failure every couple of months in very small systems (LANL systems in the Table) to a failure every several hours for the Blue Waters system. Moreover, the time to restart the machine and the applications after a system-wide outage is taking longer times for larger machines. The frequency of failures and the system complexity is making the task of failure detection and prediction much harder.

Table 2.1 Types of failures

Hardware failures	Software failures
Failures that affect group of nodes	
Switch; Power supply	Scheduler; FS; Cluster Management Software
Individual node failure	
Processor; Mother Board; Disk	OS; Client Daemon

Table 2.2 Different failure characterization studies and their results

System	MTBF	Root cause analysis	Citation
-A cluster of 12 SGI Origin 2000 (1500 CPUs) -A PC cluster (1000 CPUs) -A cluster of 162 Itanium dual CPUs	MTTI of 1 day, less than 1 h and about 6 h respectively	Software was at the origin of most outages (59–84 %)	Lu et al. [51]
Blue Gene/L during 6 months	More than 10 h	–	Leangsuksun et al. [45]
- Blue Gene/L (131k CPUs) - Red Storm (11k CPUs) - Thunderbird (9k CPUs) - Spirit (1k CPUs) - Liberty (512 CPUs)	–	Software caused 64 % of failures, while hardware only 18 %	Oliner et al. [57]
22 different systems at LANL, mostly large clusters of SMP and NUMA nodes, over a period of 10 years	–	Hardware is the main cause of failures with percentages ranging from 30 % to more than 60 %. Software is the second largest contributor, with percentages from 5 to 24 %	Schroeder et al. [68]
Blue Gene/P	The job level MTBF is about three times larger than that system level MTBF	–	Zheng et al. [87]
Blue Gene/L, Blue Gene/P, SciNet, Google	–	A large fraction of DRAM errors can be attributed to hard errors	Hwang et al. [38]
Beowulf-style PC clusters: Platium, Titan	6 h	Software represents the cause of most outages with 84 % for Platium and 60 % for Titan	Lu et al. [50]
Blue Waters	6–8 h	Software is the cause of almost all storage failures, more than 50 % of the node failures and more than 50 % of system wide outages	Di Martino [53]

2.4.1 Location Propagation

In general on all systems, over 15 % of failures affect more than one node (without considering system wide outages). For example failures in the voltage converter module, or problems with the cabinet controller on the Blue Waters system, affect a whole blade consisting of four nodes.

Large-scale systems contain a large number of nodes that are organized in a hierarchy. For example for the BlueGene systems, nodes are gathered into mid-planes and multiple mid-planes form a rack. The propagation path for different error types follows closely the way components are connected in the system. For example, if a fan breaks, all nodes sharing the same rack will be affected. In general, sequences of non faulty events, like warnings, following a failure do not propagate on different locations and if they propagate they appear on a small number of nodes: only around 22 % for Mercury and 25 % for Blue Gene/L show some kind of propagation. This phenomenon is consistent with all the systems presented in Table 2.2.

A very small number of failures appear on locations that do not follow the topology of the machine. An example of such a failure can be seen on the Mercury machine, when it experiences NFS (Network File System) problems. The event "`rpc: bad tcp reclen d+ (nonterminal)`" indicates network file system unavailability to any requests for a machine. In applications using the network file system this could cause file operations to fail and the application to quit. Also all nodes from which the application tries to access the network file system will be affected by this problem. This failure usually occurs nearly simultaneously on a large number of nodes depending on where the application was running.

In general some failure types are more likely to create cascade failures than others. This is the case, for example, of network and filesystem failures. In general, errors in memory or processor caches do not show the same behavior.

2.4.2 Failure Statistics

We divided all failures in 5 main categories that can be encountered in all systems: hardware, software, network, facility and unknown. Table 2.3 presents the percentage of failures representing each considered type for several HPC systems.

Table 2.3 Percentage of different failure types

Category	Blue Waters (%)	Blue Gene/P (%)	LANL systems (%)
Hardware	43.12	52.38	61.58
Software	26.67	30.66	23.02
Network	11.84	14.28	1.8
Facility/Environment	3.34	2.66	1.55
Unknown	2.98	–	11.38
Heartbeat	12.02	–	–

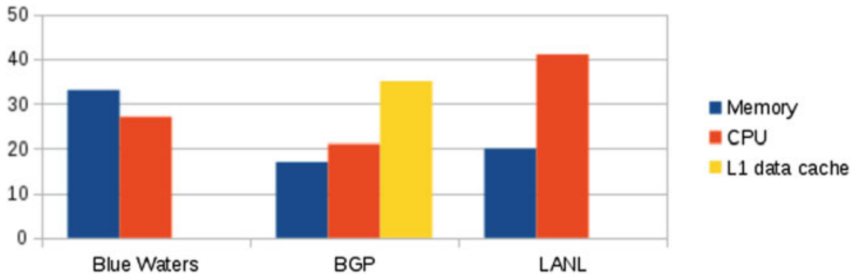

Fig. 2.3 Percentage of main hardware failures

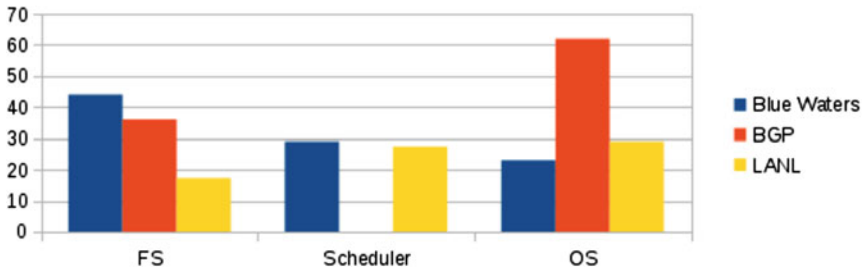

Fig. 2.4 Percentage of main software failures

Hardware represents the majority of failures for all systems, with the lowest percentage of 43.12 % for the Blue Waters system and 61.58 % for the LANL systems. As shown in Fig. 2.3, the majority of hardware failures were memory and processor errors. Moreover, failures with hardware root causes were limited to a single node in 96 % of the cases, or a single blade consisting of 4 service nodes in 99.3 % of the cases.

Software errors represent over 30 % of total failures for the Blue Waters system, while for the LANL system they represent only 23 %. In general, as the system increases in size and complexity, the number of software failures has continued to increase while the hardware failures represent a smaller percentage.

Figure 2.4 presents the main causes of software failures. The main ones are filesystem problems (Lustre for the Blue Waters system, GPFS for BGL and several for LANL: Cluster File System, Parallel File System, NFS, Scratch FS and Vizscratch FS), failures of the job scheduler and operating system problems. On the Blue Waters system, 12 % of total software failures caused system-wide outages (SWO) and represent over 75 % of all causes that triggered SWO. Moreover, software failures, when not causing an SWO, propagate to more than one node in 15 % of the cases.

Environmental failures include power-outages, failures related to temperature, cooling hardware problems and others. Table 2.4 presents the main failure types for each main category for each system.

Table 2.4 Main specific failure types

Blue Waters	Blue Gene/P	LANL systems
Hardware		
RAM 33.12 %	L1 data cache parity error 35.27 %	CPU 41.35 %
CPU 27.04 %	CPU 21.81 %	DIMM 20.08 %
	Memory 16.72 %	
Software		
FS (Lustre) 27.2 %	OS 62.11 %	Other software 21.89 %
Scheduler 18.9 %	FS 36.02 %	OS 20.99 %
		DST 21.02 %
		FS 12.33 %

2.4.3 Additional Information

Some of the studies give additional information to what is presented in the Table 2.2. Schroeder et al. [68] demonstrate that the number of failures per socket in different systems is rather stable from 1996 to 2004. They also find that average failure rates differ wildly across systems, ranging from 20–1000 failures per year.

In a paper from 2011, Zheng et al. [87] analyze Blue Gene/P at Argonne National Lab and by using both RAS and job logs, they filter out failures that do not affect any jobs. By characterizing only the failures that lead to application crashes, they make a couple of interesting observations which might influence the fault-tolerant protocol used by different applications. Another observation their study reveals is that the probability of job interruption is high if there exist historical records of application-related interruptions. Moreover, most errors due to bugs in the application tend to be reported in the first hour. Therefore it is not recommended to introduce checkpointing early in the execution period if the job has historical records of application-related interruptions. Another interesting find is that the MTBF after filtering all failures that do not lead to application crashes is about three times larger than that without applying the filtering.

Tsai et al. [77] present a study that uses data collected from a population of over 50,000 customer deployed disk drives to examine the relationship between disk soft errors and failures, in particular failures manifested as hard errors. They observe that soft errors alone cannot be used as a reliable predictor of hard errors. However, in the cases where soft errors do accurately predict hard errors, sufficient warning time exists for preventive actions. Disk failures will be inspected in more detail in the next section.

In [38], Hwang et al. analyze data on DRAM errors collected on a diverse range of production systems in total covering nearly 300 terabyte-years of main memory. The authors provide a detailed analytical study of DRAM error characteristics, including both hard and soft errors.

Table 2.5 presents high-level statistics on the frequency of correctable and uncorrectable errors per year of operation, broken down by the type of hardware platform.

Table 2.5 Memory errors per year

Platform	Technology	Time (days)	Scrubbers	Corrected errors	Uncorrected errors
Google—platform 1	DDR1	2.5 years	no	45.80 %	0.17 %
Google—platform 2	DDR1	2.5 years	yes	22.30 %	2.15 %
Google—platform 3	DDR2	2.5 years	yes	19.60 %	2.65 %
Google—platform 4	Fully-Buffered DIMM	2.5 years	no	N/A	0.27 %
Blue Gene/L	N/A	214	no	5.32 %	N/A
Blue Gene/P	N/A	583	no	3.55 %	1.34 %
SciNet	N/A	211	no	2.51 %	N/A

For the Google platforms the last two columns represent the percentage of machines affected by at least one error, while for the HPC system the percentage refers to nodes. As we see, memory errors are not rare events. For example, about a third of all machines at Google experience at least one uncorrectable memory error per year and the average number of correctable errors per year is over 22,000. These numbers vary across platforms, with some system experiencing nearly 50 % of correctable errors, while in others it represents only 15–30 %. For the platforms with a low percentage of machines affected by correctable errors, the average number of correctable errors per machine per year is the same or even higher than for the other platforms.

In general, we have seen that in all studies that analyze memory failures, the number of errors is highly variable depending on the machine under study. Some systems develop a very large number of correctable errors compared to others. The overall image of memory errors shows that for all platforms, 20 % of the nodes with errors make up more than 90 % of all observed errors for the corresponding system. This can be explained by the observation that memory errors are highly correlated.

While correctable errors, depending on their number, typically do not have an immediate impact on applications, uncorrectable errors usually result in a crash. Uncorrectable errors are less common than correctable errors (as seen in Table 2.5); however, they do happen at a significant rate. Memory failures will be inspected in more detail in the next section.

Recently, systems have started to have heterogeneous nodes, which may have different failure rates. The study from 2013 [75] has shown that even homogeneous nodes present different rates both in the failure rate as well as the reliability.

2.4.3.1 Time to Repair

There are several studies that analyze the mean time to repair [45, 53, 68, 69]. In general, the studies conclude that hardware type has a major effect on repair times. While systems of the same hardware type exhibit similar mean and median time to repair, repair times vary significantly across systems of different type. All studies make an analysis of the relative impact that different types of failures have on the

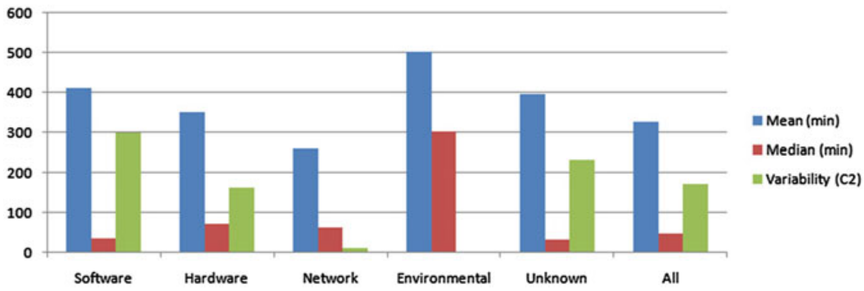

Fig. 2.5 Time to repair for different failure types

total number of node repair hours (hours required to repair failures due to the same root cause, multiplied by the number of nodes involved in the failure). Figure 2.5 shows the median and mean of time to repair as a function of the root cause, and as an aggregate across all failure records. The figure summarizes the information given by all studies presented in Table 2.2.

Both the median and the mean time to repair vary significantly depending on the failure type that is being analyzed. The mean time to repair ranges from less than 3 h for failures caused by human error, to nearly 10 h for failures due to environmental problems, while the repair time across all failures (independent of root cause) has an average close to 6 h. The most frequent type of failures affecting the systems are hardware and software.

After merging the results from all the studies, we observed that failures with software root causes are responsible for the largest percentage of the total node repair hours. This is surprising considering that hardware failures represent the majority of failures for all analyzed systems. Together, hardware and software node repair hours represent over 90 % of the total downtime. For larger system this percentage is even higher. For example, for the Blue Waters system, over 98 % of the total node repair hours are represented by software and hardware problems. The rest of 2 % represents network, facility, environmental, and unknown failures.

In general, hardware problems are closely monitored and are well managed by the vendor management system. In addition, hardware is easier to diagnose than software. There is also a difference in the distribution of the number of nodes involved in failures with different types of root causes. Failures with hardware root causes propagate outside the boundary of the smallest unit of node aggregation in a very small percentage of cases. Conversely, software failures propagate in a much larger percentage of the cases (for the Blue Waters system this number is 20 times more often than the hardware propagation percentage).

One reason for the high variability in repair times of software and hardware failures might also be the diverse set of problems that can cause these failures. For example, the root cause information for hardware failures spans on a very large list of different categories, compared to only two (power outage and A/C failure) for environmental problems on the LANL smaller systems. Breaking down the hardware

problems in specific categories, the average time to repair for each still presents a high variability. Moreover, even the same hardware problem can have different times to repair depending on when it occurred. For example, the variability for repair times of CPU, memory, and node interconnect problems, as expressed with the squared coefficient of variation, are 36, 87, and 154, respectively on the LANL system. This indicates that there are other factors as well contributing to the high variability of hardware failures. Software failures have a similar behavior.

Another important observation from these studies is that the time to repair for all types of failures is extremely variable, except for environmental problems. These failures are better understood and the solution policies are already in place and it usually requires changing a component in the system, which explains the long average times to repair such problems.

2.4.4 Silent Errors

All the characteristics and statistical properties extracted so far are covering only a subset of all actual faults, namely those that can be detected by software and hardware monitoring systems. These are the faults that have a fail-stop behavior or degrade the performance of the systems and/or applications. Silent data corruptions (SDC) are undetected faults that are usually materialized as bit flips in storage (both volatile memory and nonvolatile disk) or even within processing cores.

In general, a single bit flip in memory can be detected and even corrected if the system is using an Error Correcting Code (ECC). Double bit flips, however, even though they are detected, often force an instant reboot of the node since ECC cannot correct them. For smaller systems, the frequency of double bit flips was considered to be seemingly low. Current studies [24] on the Oak Ridge National Laboratory's Cray XT5 system have shown that the density of DIMMs causes uncorrected but detected errors to occur on a daily basis (at a rate of one per day for 75,000+ DIMMs).

Single bit flips in the processor cores mostly remain undetected since few processor structures are protected. For this reason, the sensitivity of register files and ALUs for SDC is significantly higher. The Blue Gene/L architecture uses an unprotected L1 cache. Due to the high number of silent corruptions, the next generation system, the Blue Gene/P implemented ECC in L1. However, since hardware redundancy still remains extremely costly, not all storage systems afford to implement ECC.

It is believed that in today's systems, the frequency of bit flips is no longer dominated by single-events caused by radiation from space but it is increasingly attributed to fabrication miniaturization and aging of silicon given the increasing likelihood of repeated failures in DRAM after a first failure has been observed [38].

Bit-flips in stale data or in the instruction caches do not impact the system and application's execution. Only those in active data or the system's code can have extremely big effects and potentially render computational results invalid without ever being detected. This creates a severe problem for today's science that relies increasingly on large-scale simulations.

There are several solutions for overcoming the effects of silent corruption on systems and applications. Redundant computing is the method of choice when the frequency of silent errors is not extremely high since it can detect silent data corruption that impacts the results. Detection requires dual redundancy, while correction requires at least triple redundancy. Such high levels of redundancy are costly. However, with the current systems and depending on the application, it might be preferable to flawed scientific results. For the Exascale era redundancy at the process level, as currently defined, would not be feasible. Thus, the state of research for HPC requires urgent investigation to level the path for the Exascale computing.

The research direction in this field spans two separate directions: (i) more efficient application level redundancy and (ii) application level detection.

The study in [38], analyzes the potential for redundancy to detect and correct soft errors in MPI message-passing applications. For this purpose, the authors investigate the challenges inherent to detecting soft errors within MPI applications by providing transparent MPI redundancy. Their theoretical model assumes that corruption in application data manifests itself by producing differing MPI messages between replicas. With this model, the authors study the best suited protocols for detecting and correcting corrupted MPI messages.

In scientific applications that involve dense matrices, checksum encodings have yielded "Algorithm-Based Fault Tolerance" (ABFT) in the event of data corruption from either hard or transient (soft) errors in the hardware. The second research direction in dealing with SDC for Exascale systems deals with optimizing or finding new ways of detecting the corruption at the application level. In [70], the authors developed a new sparse checksum encoded algorithm that can be applied to all the key operations in the Preconditioned Conjugate Gradient method (PCG), including sparse matrix-vector multiplication, vector operations and the application of a preconditioner through sparse triangular solution. Their method detects a single error in the matrix and vector elements and in the metadata representing the sparse matrix row or column indices.

In [17] the authors convert the detection problem to a one-step look-ahead prediction issue. By modeling the values taken by different variables used by an algorithm based on their history, one can predict the future state of each of them. A running HPC application often iteratively operates on a set of data, whose values thus change over time. As shown in Fig. 2.6, at each iterative time-step, the detector dynamically predicts the possible range for the next-step data value. The detector will consider a

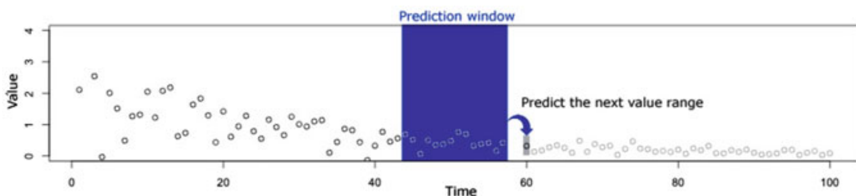

Fig. 2.6 Value prediction model for SDC detection

value as an outlier if it falls outside this range. The study explores the most effective prediction methods for different HPC applications (the Auto Regressive model, the Auto Regressive Moving Average Model, Linear Curve Fitting, and Quadratic Curve Fitting). Experiments show that this method can obtain an F-measure around 80 % for silent bit-flip errors.

2.5 Modeling

Failures and downtime intervals have a severe impact on the performance of applications in large scale HPC environments. Research efforts have been deployed to understand the failure behavior on such computing systems. Failure modeling is an important research direction used in guiding reliability-aware resource allocation and optimizing fault-tolerant protocols in order to minimize the performance loss due to failures. The failure's distribution is also used by fault-tolerant protocols, like checkpointing, to decide an advantageous trade-off between frequently creating checkpoints, which takes resources away from completing execution of the application but reduces the amount of lost calculation, and infrequent checkpoints, which diverts less resources but incurs greater losses when a fault occurs. Prediction can also be built when knowing the failure distribution and later used for marking suspicious components and monitoring them more frequently.

In general, studies that analyze failures on different HPC systems, like the ones presented in the previous section, are also extracting the failure distribution. Most research characterizes an empirical distribution by using three import metrics: the mean, the median, and the squared coefficient of variation (C^2). The squared coefficient of variation is a measure of variability and is defined as the squared standard deviation divided by the squared mean which has the advantage of allowing comparison of variability across distributions with different means.

Another used method is the empirical cumulative distribution function (CDF) that studies how well the data is fit by several probability distributions commonly used in reliability theory, like the exponential, the Weibull, the gamma, and the log-normal distributions. The method uses the maximum likelihood estimation to parameterize the distributions and evaluate the goodness of fit either by visual inspection and the negative log-likelihood test. The primary problem with using goodness-of-fit measures is that usually they do not take into account the number of free parameters in a model; with enough free parameters, any model can precisely match any dataset. For example, a phase-type distribution with a high number of phases would likely give a better fit than any of the above standard distributions, which are limited to one or two parameters. The standard distribution is preferred whenever the quality of fit allows it.

There are several goodness-of-fit tests that are currently used from Anderson-Darling and Kolmogorov–Smirnov to the chi-square test. The chi-square goodness-of-fit test can be applied to discrete distributions such as the binomial and the Poisson, while the Kolmogorov–Smirnov and Anderson–Darling tests are restricted to continuous distributions.

2.5.1 Randomness Testing

Before investigating the distribution fitting for failures, several tests of randomness need to be run in order to identify whether the generated failures have a truly random behavior. A random data series exhibits trends of periodicity, autocorrelation, or nonstationarity. Fitting probability distribution to nonrandom data is not statistically relevant since data with such properties do not respect the basic assumption of all standard statistical tests. For this purpose, when fitting distribution to a dataset, randomness needs to be tested first.

The methodology used in the literature focuses on two classes of randomness tests: parametric and nonparametric tests. In the first case the algorithm has information about the distribution of the data and only parameters need to be found. The second class refers to tests where the distribution of the observed data is unknown. In this category, there are tests of randomness based on runs or trends, such as the Mann-Kendall test, the Bartels' rank test, or the Wald-Wolfowitz test, as well as tests based on entropy estimators. A commonly used example of run-based randomness test is the Wald-Wolfowitz test in which each interval of time is compared with the mean in order to determine the mutually independent property of the intervals. Another frequently used test is the autocorrelation that is used to discover repeated patterns that differ only by a lag in time.

The autocorrelation function describes the correlation between values in the data at different times. Plotting the autocorrelation values makes it easy to visualize the lags that offer correlation. Examples of the auto-correlation function, for a periodic and random data set, can be seen in Fig. 2.7. Random data sets have only one peak for lag 0, which means that the signal has a high similarity only with itself. Periodic data sets have multiple peaks, visible in Fig. 2.7b. Thresholds can be chosen to decide when a data set is periodic either by setting it manually or in automatic using different heuristics [29].

Since no method is perfect, in general it is recommended to run multiple tests on the same data set and compare the results. The runs test and the up/down test return one value called a probability value and noted p-value. This value is used to either reject the null hypothesis about the randomness of the data, if the p-value is smaller than or equal to the significant threshold, or to confirm that the data is truly random otherwise. Depending on how many samples are available in the data set, the statistical threshold might have different values, between 1 and 15 %. For large data sets, a significance level is chosen before data collection and is usually set by all research to 5 %. Other significance levels, for example 1 %, may be used, depending on the field of study. For the autocorrelation function, in general a confidence interval of 95 % is considered. Thus, if the value of the autocorrelation test is out of this confidence interval, the sample contains a high correlation of order 1, which implies the nonrandomness of the data.

In general, most HPC systems pass the randomness test. By analyzing studies that looked at different HPC systems from several national laboratories, around 85 % of systems presented a random time between failures. Interestingly, after filtering out

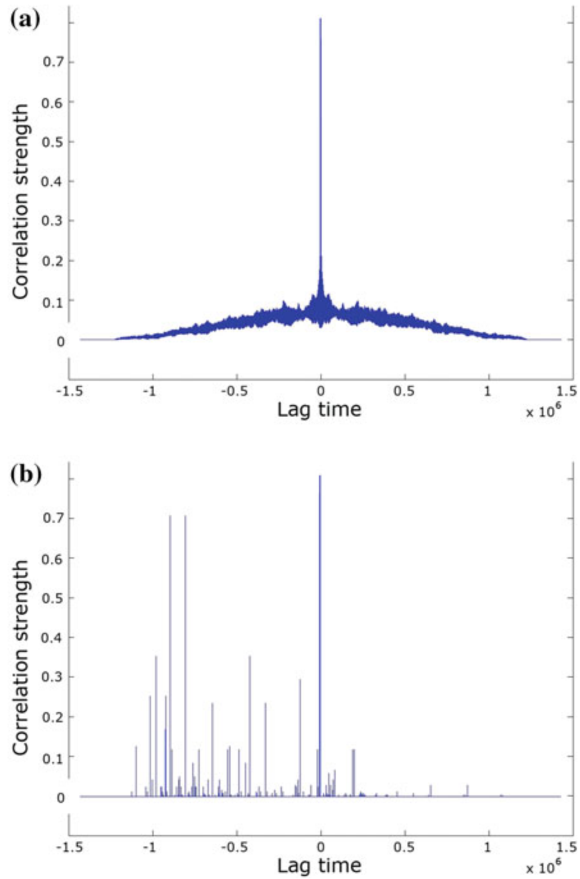

Fig. 2.7 Auto-correlation plots for different signals. **a** Random signals. **b** Periodic signals

the failures that can be predicted with a failure prediction method, the remaining failures pass the randomness test for 75 % of the systems that had nonrandom time intervals initially. This means that, now, fault-tolerant protocols can be optimized for these systems as well, since failures can be fitted to a distribution.

2.5.2 Fitting Distributions

The principle behind fitting a data set with a distribution function is to find the type of distribution (for example, exponential, normal, log-normal, gamma) and the value of its parameters (mean, variance, etc.) that give the highest probability of producing the observed data. The objective of a fitting distribution algorithm in the fault tolerance field is to find the mathematical model that best describes the inter-arrival time between failures. As mentioned before, only traces with a truly random behavior can be used to find a good probability distribution that is a good match to the empirical distribution.

2.5.2.1 Fitting Methodology

There are several available methods that can be used to fit the empirical data to probability distribution functions. In general, the methodology used has three distinct steps: (1) select a set of candidate distributions either by domain knowledge about the given data set or by including as many distribution functions as possible; (2) estimate the parameter values for each empirical distribution; and (3) choose the best fit with the most likely similarity either by manual inspection or by using automatic thresholds.

Many distributions could be used as input candidates in the first step of the fitting methodology, but in general, in the HPC community there are several commonly used distribution functions to model failures [9, 39] namely, the Poisson, exponential, Weibull, log-normal, normal, and gamma distributions.

The second step of the methodology deals with computing the best parameter values for each candidate distribution. Specifically, the maximum likelihood estimates (MLE) method is being used [46] to chose the values that are the most likely to fit the empirical data. Several older studies also use the moment matching method. However, since this methods has been shown to be sensitive to outliers [23], recent work has mainly used the MLE method. This method aims to maximize the logarithm of the likelihood function that corresponds to the closest distance between the empirical distribution and the samples resulting from the distribution function with certain parameters. The negative log likelihood value produced by the MLE is being used to rank different distributions. This method will give a list of ordered distributions, however, without giving an indication of how good the distributions actually fit the empirical data.

In order to check if a distribution is actually a good model for the given data, we must check also the goodness-of-fit between the data sample and the synthetic sample generated by the distribution. In most cases, the number of inspected distributions is relatively low so a visual inspection of the fitting is desirable. Figure 2.8 shows

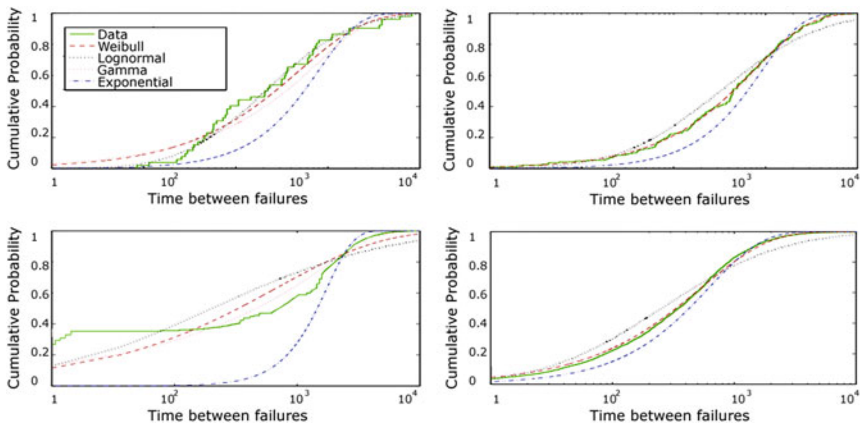

Fig. 2.8 The CDF, fitted with a Poisson, normal and log-normal distribution

four cumulative distribution functions of four separate data sets that represent the measured number of failures per compute node in different year intervals, with four different distributions fitted to it: the Weibull, log-normal, gamma, and exponential distributions. It is visible that, in general, the distribution between failures is well modeled by a Weibull or gamma distribution, for some year intervals better than for others. Both distributions create an equally good visual fit and the same negative log-likelihood. In general, the simpler exponential distribution is a poor fit. This can also be seen by looking at its C^2, which is equal to 1 for the exponential distribution, for example for the first two figures. This value is significantly lower than the data's C^2 of 1.9. Using the C^2 goodness-of-fit method, choosing the distribution can be made in an automatic way.

There are many goodness-of-fit tests in the literature, but most of them are not used in practice in the HPC community. The Kolmogorov–Smirnov test is a nonparametric test and one of the most popular methods to date. Through this test, samples are being compared with a reference probability distribution. The Kolmogorov–Smirnov test quantifies a distance between the empirical distribution function of the sample and the cumulative distribution function of the reference distribution. What makes this test attractive is that it also rejects the true randomness hypothesis.

Another popular method for visualizing the fitness of a distribution is the standard probability-probability plot (P-P plot) and quantile-quantile plot (Q-Q plots). In a Q-Q Plot, the cumulative distribution function associated with the empirical measure of the sample and the CDF from the theoretical distribution are plotted against one another. If the extracted distribution would fit exactly the given data observations, the resulting Q-Q plot would be very nearly a line intersecting the origin and having slope of 1. Figure 2.9 presents examples of Q-Q plots for failure data sets from two wide-area distributed computing environments after fitting them on a Weibull distribution.

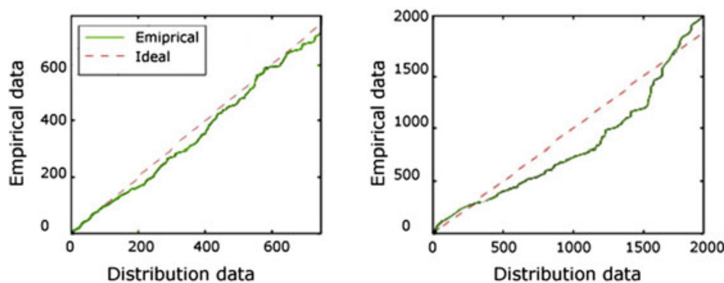

Fig. 2.9 The Q-Q plot for two data sets and Weibull distribution samples

2.5.2.2 Failure Distribution for HPC System

There are several studies that analyze different HPC systems. Some even investigate the relationship between the distribution of failures without considering prediction and the probability distribution of the false negative alerts when prediction is performed.

Table 2.6 presents the results obtained by different studies when fitting the failure distributions for several HPC systems.

There was an assumption in the past that the failure rate at all nodes follows a Poisson process with the same mean. In this case the distribution of failures across nodes would be expected to match a Poisson distribution. The large majority of studies for current HPC systems have found that the Poisson distribution is a poor fit, the Weibull and log-normal distributions being a much better fit, visually as well as measured by the negative log-likelihood.

Overall, we observed that the failure rate varies widely across systems, ranging from as low as 17 failures per year to about 1200 failures per year depending on the size and the architecture used by each system. In fact, variability in the failure rate is high even among systems of the same hardware type. Moreover, the same system might experience different distributions depending on when in the system's lifecycle the failures are inspected. For example, some systems present a complete different distribution that best fits their results when analyzing the first on second half of the system life.

For failure inter-arrival distributions, it is useful to know how the time since the last failure influences the expected time until the next failure. This notion is captured by the distribution's hazard rate function. An increasing hazard rate function predicts that if the time since a failure is long then the time lag until the next failure will be short. And a decreasing hazard rate function predicts the reverse, i.e., not seeing a failure for a long time decreases the chance of seeing one in the near future. In general, the analyzed systems are well-fit by a Weibull distribution, in most cases with a shape parameter of less than 1, indicating that the hazard rate function is decreasing.

Table 2.6 Failure distribution for several systems

System	Failure distribution	Citation
20 systems at LANL	Weibull distribution with decreasing hazard rate	Schroeder et al. [68]
O2K	Weibull distributions	Lu et al. [50]
Titan	Exponential distribution	Lu et al. [50]
Platium	Exponential distribution	Lu et al. [50]
Blue Gene/L	Weibull distribution	Taerat et al. [74]
Blue Gene/P	Weibull distribution	Harper et al. [34]

Another important overall observation after inspecting the studies, is that some nodes experience a significantly higher number of failures than other nodes in the same cluster. In some cases, it was observed that nodes that make up only about 5 % of the entire cluster account for over 20 % of all the failures. One explanation is that these nodes run different workloads. For example, nodes that are used for visualization, as well as computation, thus resulting in a more varied and interactive workload compared to the other nodes, experience a higher failure rate. Similar observations are made for other systems as well. For example, it was observed that front-end nodes, which run a more interactive and varied workload have a different, higher failure rate than all other nodes. Another interesting observation is that, while the whole system best fits the Weibull and gamma distributions, individual node failures are best fit by the log-normal distribution, followed by the Weibull and the gamma distribution.

As the failure distribution varies depending on the node's workload, as well as for other reasons, it is important to characterize how this phenomenon influences the applications running on a system. The study from [75] uses the failure trace obtained from prominent HPC platforms to study and compare different distributions, Exponential, Weibull, and Log-Normal for fitting the failures that affect applications running on k nodes. Their results indicate that Weibull distribution results in the better reliability model in most of the cases for the given data.

Table 2.7 shows the distribution that fits the time to repair for several systems. The distributions, as well as their parameters are very different depending on the system. This is to be expected since the process is dependent on the failover mechanisms offered by the vendor, the policy of the center and the policies used by system administrators.

Not all failures propagate to the application level and crash the job. Indeed, as seen in the previous section, a large percentage of failures either do not affect the application at all or degrade its performance without causing it to crash. All failure distribution functions presented thus far deal with modeling all system level failures without making a distinction between the fail-stop ones. Moreover, job-related redundancy is not negligible. Some studies have shown that over half of the resubmitted jobs were allocated to the same failed nodes by the scheduler. When analyzing application crashes it is important to separate the redundant failures, either due to the same faulty node or because users keep submitting the same buggy code, thereby leading to the same type of application errors at different locations.

Table 2.7 Failure distribution for several systems

System	Time to repair distribution	Citation
20 systems at LANL	Log-Normal distribution	Schroeder et al. [68]
O2K	Inverse normal distribution	Lu et al. [50]
Titan	Gamma distribution	Lu et al. [50]
Platium	Truncated Weibull distribution	Lu et al. [50]

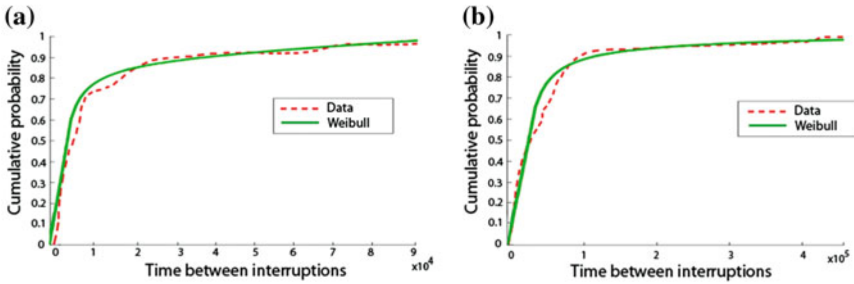

Fig. 2.10 Experimental CDF for inter-arrival times of interruptions. **a** Due to system failures and **b** due to application errors

In general, application interruption distribution can be fitted by a Weibull distribution with a shape parameter of less than 1, indicating that the hazard rate function is decreasing as in the case of system failures. In general, over 65 % of job interruptions are caused by system failures, and the rest are caused by application errors. Figure 2.10 presents the distribution fitting results of interruption inter-arrival times for the Blue Waters system at the National Center for Supercomputing Applications for a period of 4 months. Weibull distribution still gives a best fit for both interrupts caused by system failures and interrupts caused by application errors, having a shape parameter of less than 1. Another observation is that the hazard rate of interruptions caused by system failures is less than the one for interruptions caused by application errors. The main reason is that application errors generally need more time for fixing, whereas some system failures can be easily solved by rebooting.

2.5.3 Including Prediction

The distribution of failures is in general known for most of the current HPC system. However, when prediction is used, and preventive actions are taken for the failures that are known in advance, the distribution of the predicted failures, as well as that of the false negatives is, in general, unknown. In other fields [52], the analysis of false negatives has given new insights into combining a predictor with fault tolerance actions, but since prediction in the HPC field is still an area of research that is rarely used in practice, such studies are rare.

In [9], the authors present such a study for several systems that were deployed at the Los Alamos National Laboratory in the past several years. Firstly, they check the randomness hypothesis of failure intervals and identify whether it is possible to fit the entire failure set to any classic probability distribution functions. After which, by studying the randomness of the false negatives alerts, they investigate the impact of the prediction process on the randomness of the data and on the final distribution fit of the predicted failures.

Interestingly, the authors report several systems with nonrandom behavior that become random after prediction is applied and only false negatives are analyzed. In this case, after prediction, the unpredicted failures can be fitted by one of the classical distribution functions. In general, the best fit distributions for each system is different, with most systems having a Weibull, other log-normal and rarely some systems are fitted by exponential distributions, all with different parameters. While no pattern can be extracted, the results are still important since after prediction fault-tolerant protocols can be optimized for the rate of false negative alerts.

Table 2.8 shows the best fitted distribution for all failures and false negatives for several systems. The figure investigates the relationship between the initial failure distribution and the false negative distribution function. In general, it was found that the best fitted distribution for the false negative alerts is the same as for the initial failure data set, but having different parameters. Interestingly, this means that a failure predictor does not change the initial distribution and affects only the scale parameters of the initial distribution.

Moreover, when the best fitted distribution is exponential, the ratio between the parameter μ_u for the initial distribution and the parameter μ_y for the false negative distribution is given by $\mu_y/\mu_u \approx 1 - r$, where r represents the recall value for the given predictor. Similarly for the Weibull distribution, the systems that fit this distribution have approximately the same shape parameter for both distributions, and the scale parameters follow the same pattern as before: $a_u/a_y \approx 1 - r$. Basically, the failure prediction mechanism acts as a scaling filter affecting only the time scale. Therefore the distribution of the false negative alerts can be estimated from the initial failure distribution, by using the recall value to scale the parameters.

The analysis of false negatives and the impact of its distribution on fault-tolerant protocols is a current area of research that has been applied on a relatively small number of systems. Moreover, there is no study on current Petascale systems that have shown more complexity and different behaviors than previous generation machine. Thus, the results of false negative distribution analysis still need to be validated on current HPC system in order for a general conclusion to be drawn.

Table 2.8 Distribution fitting for all failures and for false negatives

System	All failures		False negatives		Param ratio
	Dist. fit	Parameters	Dist. fit	Parameters	
Blue Gene/L	Exp.	$\mu_x = 62431.3$	Exp.	$\mu_y = 113289$	0.55
LANL system 3	Exp.	$\mu_x = 215705$	Exp.	$\mu_y = 393538$	0.54
LANL system 4	Exp.	$\mu_x = 204544$	Exp.	$\mu_y = 371218$	0.54
LANL system 5	Exp.	$\mu_x = 197671$	Exp.	$\mu_y = 382671$	0.51
LANL system 6	Exp.	$\mu_x = 1007800$	Exp.	$\mu_y = 1912690$	0.54
LANL system 23	Weib.	$a_x = 509380$ $b_x = 0.84$	Weib.	$a_y = 895274$ $b_y = 0.85$	0.56

2.5.4 Per Component Failure Distribution

Large-scale applications using large numbers of processors and memory in parallel are relatively more sensitive to individual component failures. It is important to understand the failure distribution of each component and how this might affect the application's tolerance to failures. Application-centric models provide more accurate reliability estimates compared to general models. This can, for example, improve the efficiency of fault-tolerant algorithms by tuning the application checkpointing strategies to the tailored model.

The components (such as memory, CPU, disk, and the network) of current HPC systems have different failure dynamics in terms of time and space. Some components fail randomly and frequently while others fail in a correlated manner. Moreover, modern computing systems use multiple heterogeneous types of processors, networks or storage systems. This makes the failure dynamics even more diverse.

As mentioned previously, the failure data of the entire system is best fitted by the log-normal and Weibull distributions with p-values of around 0.4–0.5 respectively, by using the standard method of maximum likelihood estimation of the Kolmogorov–Smirnov test to evaluate these distributions.

In [36], the authors analyze the distribution of the most frequent failures on a particular HPC system (Mercury at the National Center for Supercomputing Applications). They found that the best fit for each failure type is still a Weibull or log-normal distribution, but depending on the analyzed type the parameters that describe the distribution might vary considerably.

For certain storage failures the distribution that best fits the failure rate is the log-normal and Weibull with average p-values around 0.52–0.61. The Network File System failures have irregular sharp spike shapes and are in general difficult to model. However, the data still fits reasonably to an exponential, log-normal or Weibull distribution (p-values 0.33, 0.22 and 0.12 respectively). Network availability failures are best fitted by a log-normal distribution with p-values around 0.38. Memory and processor cache errors are the most similar and are both fitted by log-normal distributions (p-values 0.82 and 0.68), Weibull distributions (p-values 0.68 and 0.58) and exponential distributions (p-values around 0.55).

The parameters for these distributions are shown in Table 2.9. In addition to having heterogeneous scale and shape parameters, the hazard rates are different as well depending on the failure type. As mentioned previously, a decreasing hazard rate means that when a component has been without failure for longer, the probability of the component failing in the future becomes lower. A shape parameter with value less than 1 indicates a decreasing hazard rate.

Storage and network errors show a clear decreasing hazard rate since the shape parameter is below one. For memory and processor cache errors the shape parameter has values above or slightly below 1, which means that the hazard rate is relatively constant. Overall the hazard rate is decreasing for the dominant failures in the system, namely storage, memory, and filesystem. This indicates that the presence of a failure in the system, in most cases, is followed by a period of increased failure rate.

Table 2.9 Failure distribution parameters for NCSA's Mercury system

Inter-event time in days				
Failure type	Distribution fit	Time interval 1	Time interval 2	Time interval 3
All failures	Weibull	$\lambda = 0.26$	$\lambda = 0.16$	$\lambda = 0.17$
		$k = 0.66$	$k = 0.61$	$k = 0.58$
Storage errors (F1)	Weibull	$\lambda = 3.16$	$\lambda = 7.57$	$\lambda = 10.68$
		$k = 0.84$	$k = 0.55$	$k = 0.65$
NFS errors (F2)	Weibull	$\lambda = 1.83$	$\lambda = 13.08$	$\lambda = 8.07$
		$k = 0.53$	$k = 0.92$	$k = 1.41$
Network unavailability (F3)	Log normal	$\mu = -1.71$	$\mu = -2.76$	$\mu = -2.62$
		$\sigma = 2.03$	$\sigma = 2.24$	$\sigma = 2.12$
Memory errors (F4)	Weibull	$\lambda = 7.18$	$\lambda = 5.5$	$\lambda = 2.27$
		$k = 0.84$	$k = 0.85$	$k = 0.7$
Processor cache errors (F5)	Weibull	$\lambda = 2.52$	$\lambda = 4.78$	$\lambda = 4.54$
		$k = 1.32$	$k = 1.45$	$k = 1.09$

The differences between the characteristics of different failure types are also visible when looking at the effect of node failure distributions on job failure probability. The weighted sum of the probabilities that the failure affects one or more of the nodes the job is running on can be used in order to compute the node failure distribution. We are assuming that all nodes are equally likely to experience a given failure.

Figure 2.11 shows this function for different error types. The results are for the Mercury system, but are general and they describe the behavior of other HPC systems. When looking at all failures at once, the distribution that fits the data is Weibull. For

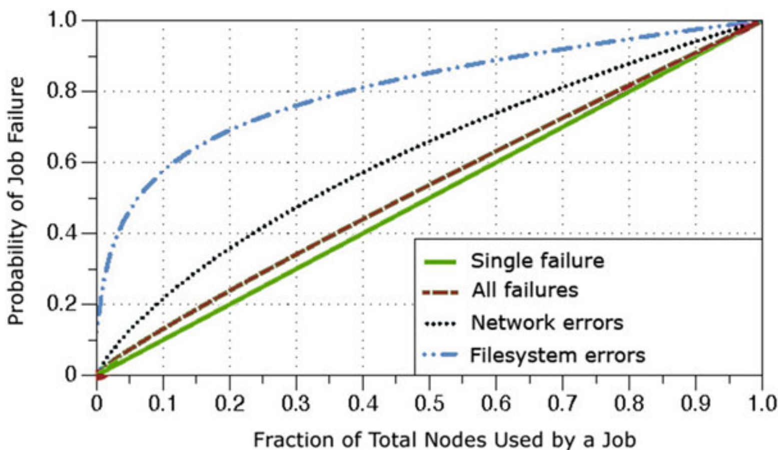

Fig. 2.11 Effect of node failure distribution on job failure probability. More details in [36]

network failures the exponential distribution offers the best fit, while for filesystem failures, the log-normal fits better. The black solid line represents the case when failures affect a single node. In this case the job failure probability rises linearly because the probability of a single node failure affecting a job rises linearly with the number of nodes used by the job. When looking at different failure types that are likely to affect a large numbers of nodes, the probability of having a job failure is high even when using a few nodes. When looking at all failures at once, the job failure probability is similar to that of a single node, but slightly higher due to the chance of a failure affecting multiple nodes (depending on the system this number can be as low as 10 %).

These differences in characteristics show that, when analyzing a system, it is important to also look at individual failure types and extract their distribution and probabilities.

Recently, studies [6, 38, 77] have focused on certain components and analyzed them separately and extensively. This is the case with DRAMs on today's large-scale systems, since main memory is currently one of the leading hardware causes for application crashes. Designing, evaluating, and modeling systems that are resilient against memory errors requires a good understanding of the underlying characteristics of errors in DRAM in the field. Studies in this field provide a detailed analytical study of DRAM error characteristics, including both hard and soft errors.

In general, failure rates are observed to increase with age, even when the early stage of the disk's lifecycle is included. The analysis of the failure distribution shows moderate degree of spatial correlation and a high temporal correlation between successive failures. Depending on the study, between 40 and 80 % of errors arrive within one minute of the previous error. This is visible when inspecting the arrival-rate distributions that present very long tails in all the studies. This observed locality implies that the errors are detected close in time, even though they may have developed long before they were detected.

The data collected by these studies cover over 1 million disks that were analyzed and their behavior investigated. Patterns were extracted by using latent sector errors, for both *nearline* storage and enterprise class disks. The correlation between latent sector errors (hard errors) and recovered errors (soft errors) have also been analyzed.

In general, about one-third of all hard errors occur within a very short time, less than one second, after an initial soft error. However, a large portion of hard errors has a time delay between the soft and hard error of over one hour, sometimes reaching even several days. In the study that gives the most pessimistic results, only about one third of the hard errors have a lead time of one hour or greater, while in the most optimistic results this number is closer to 80 %. These results are very promising considering that such a lead time could allow preventive actions to finish before the hard errors occur.

2.6 Prediction

Over the years, approaches on prediction have been developed in a variety of fields, from astrology, meteorology to stock analysis and politics to computer science and engineering. High reliability and availability are important requirements for many systems, such as switches and track circuits for railroad networks [18], liquid storage tanks during earthquakes [58] or routers and batteries for self adapting sensor networks [1]. For this purpose, several methods have been used depending on the environmental variables gathered by each system. For example, multivariate statistical models have been developed to improve the ability to predict the occurrence of broken rails; Data mining classification models are used to create predictive models on a combination of hourly temperature readings with fire reports in order to build the context and model environmental variability for sensor networks.

Fatigue prediction has also been an area of increasing research in several fields. The term "Fatigue" refers to a failure of a component as a result of cyclic stress and it occurs following the same patters no matter the system analyzed. There are three phases that characterize fatigue failures: initiation, propagation, and catastrophic overload failure. The duration of each of these three phases depends on many factors including fundamental raw material characteristics, magnitude, and orientation of applied stresses or processing history. In the past, predicting fatigue life has been one of the most important problems in design engineering for reliability and quality. Holmgren et al. present an overview in their 1996 paper [37] of different methods for fatigue life prediction. Even though their study focuses on bogie beams, the presented methods are general and can give a good background for understanding the evolution of aging predictors in computer systems.

The fatigue life prediction methods use, in general, three primary steps. First, a theoretical or constitutive equation is defined, which forms the basis for modeling. Depending on the type of system that needs to be modeled, appropriate assumptions need to be made in constructing the constitutive equation. Second, the constitutive equation is translated into a model. The model considers the predicted stress–strain values for the system under study and returns stress values for the simulated conditions. Third, the model is tested and validated by measurement data.

In general, environmental metrics are used as the input data that creates the model and triggers predictions. Out of all environmental metrics, the most used is the load history, since it is uniaxial and proportional. Fatigue can then be evaluated with the S–N curve, also known as the Wöhler curve, which represents a graph of the stress amplitude against the logarithmic scale of cycles to failure. In many applications we deal with multi-axiality and non-proportional loading. In this instance, the S–N curve is insufficient for fatigue prediction. Weibull distribution and modified Goodman diagram modeling are two other frequently used methods. Moreover, critical plane models examine stress state in different orientations in space and can therefore incorporate some effects of multi-axiality and non-proportionality. Because they can accurately predict the fatigue failure phenomenon for many structural applications, they have gained a wide acceptance among the engineering community.

In computer science, prediction methods are used in various areas. For example, branch prediction in microprocessors tries to prefetch instructions that are most likely to be executed; similarly, memory or cache prediction tries to forecast what data might be required next. In the fault-tolerance community, the focus is on predicting computer system failures, a topic that has attracted interest for more than 30 years. However, what is understood by the term "failure prediction" varies among research communities and has also changed over the decades. In reliability theory, the goal of reliability prediction is to assess the future reliability of a system from its design or specification. For clouds [31, 80], run-time information can be used to identify the current execution state, and to check whether the design-time model will satisfy a set of wanted/unwanted properties in the future.

As computer systems are growing more and more complex, they are also changing dynamically due to the mobility of devices, updates and upgrades, changing execution environments, online repairs, the addition and removal of system components and the systems/networks complexity itself. Classical prediction methods do not work online and are therefore not capable to reflect the dynamics of run-time systems and failure processes. Such methods are typically useful in design for long term or average behavior predictions and comparative analysis. New methods have been developed specifically for short-term predictions on rapidly changing computing systems.

These new methods are almost entirely based on data mining, by using either classification, for predicting the outcome from a set of finite possible values; regression, for predicting a numerical abnormal value; clustering, for summarizing data and identifying groups of similar data points; association analysis, for finding relationships between attributes; or deviation analysis, for finding exceptions in major trends or structures.

Over the years, approaches on failure prediction in computer science have been developed in relation to reliability theory and preventive maintenance [30, 56, 68], by using the lifetime distribution or the component aging rate. Models evolved by trying to incorporate several factors into the distribution, for example the manufacturing process [79] or code complexity [22]. As the methods from other fields, these solutions are tailored to long-term predictions and, in general, do not work appropriately for online failure prediction. We will look at these methods next and in the following sub-section we will analyze methods for short term prediction.

2.6.1 Long-Term Prediction

In general long-term predictions use deterministic models for approximating the aging indicators. In addition some studies use an automated procedure for statistical testing of their correctness in order to find the optimal rejuvenation schedule under utility functions [2]. Aging in this context of long-term prediction refers, in general to software aging. The software aging phenomenon is defined by an increase in the failure rate or in the performance degradation of a system, which can be induced due to unreleased resources, the accumulation of errors in the system state, filesystem degradation or to the consumption of resources such as physical memory.

While a priori the most straightforward solutions to fix the software bugs that cause software aging, in practice this can be rarely applied due to many reasons from application complexity to budget constraints.

Software rejuvenation is the most used solution for the aging phenomenon, by restarting the software continuously after certain time frames or at a specific performance degradation level. This approach has been investigated in context of application replication in order to avoid service outages [2].

In general, in order to apply software rejuvenation techniques effectively, the aging process needs to be modeled. These models allow to estimate the current or future progress of the performance degradation or failure state. Moreover, they can facilitate the schedule of optimal rejuvenation times or the management of system administrator tasks, such as alerting operators of anticipated crashes. These approaches are known as adaptive or proactive software rejuvenation.

In [14], the authors present a comprehensive analysis of Software Aging and Rejuvenation literature, by reviewing almost 500 papers that were published in the fields of software engineering and software dependability. The aim of the paper is to provide an overall picture of the state of the art in this field. For this purpose, the paper surveys relevant studies that have been used to forecast the software aging phenomenon and to apply software rejuvenation. It also presents a study of the kind of systems and aging symptoms that have been studied, and the techniques that have been proposed to rejuvenate complex software systems.

In general, the work addressing software rejuvenation, though rich in research studies, often lacks experimentation on real systems. Most of the studies are validated by numerical examples and by simulations instead of real scenarios. In order for these studies to be useful for practical scenarios it is not enough to study these techniques in simulated environments since the models presented in the survey make assumptions about the system being modeled, which can be validated only by comparing the actual behavior of the system with the prediction of models, and because the deployment of software rejuvenation on real systems can reveal practical issues that would be neglected otherwise.

A lot of research has been focused on software rejuvenation techniques as well, however most studies do not specify the particular rejuvenation technique analyzed. This phenomenon denotes that the focus of the community is more on finding the theoretical optimal scheduling rather than on the design of the actual rejuvenation action. Rejuvenation techniques are useful to reduce the performance decrease by keeping the cost of software rejuvenation low. Most studies analyze approaches that are independent from the application and that involve a restart of the corresponding software. At the same time, other approaches are application-depended by using specific features of the system to improve the availability and efficiency of the software rejuvenation technique. The best results in the literature are given by these application-specific methods, especially in the context of embedded systems and distributed systems. Another interesting result of current research is given by rejuvenation techniques that are selective and can restart parts of the system instead of the whole software.

Another long-term prediction method category is using failure models in order to predict the reliability of a component in the system. Most studies use the Weibull and other failure distributions discussed in the previous section in order to define the state of components. This prediction can be combined with short-term predictions in order to increase their coverage [9].

2.6.2 Short-Term Prediction

If the system knows about a critical situation in advance, it can try to trigger preventive measures in order to mitigate the effect of a failure, or it can prepare repair mechanisms for the upcoming failure in order to reduce time-to-repair. For this purpose, short-term predictions need to be accurate and leave enough time for the preventive measure to be taken.

Recent methods for short-term failure prediction are typically based on run-time monitoring as they take into account a current state of the system. The taxonomy we will be using is presented in Fig. 2.12.

There are two levels of online failure prediction in the literature: component level and system level failure prediction. The first level assumes methods that observe components (hard drive, mother board, DRAM, etc.) with their specific parameters and domain knowledge and define different approaches that give best prediction results for each [38]. One example of this type of approach is to compare the execution

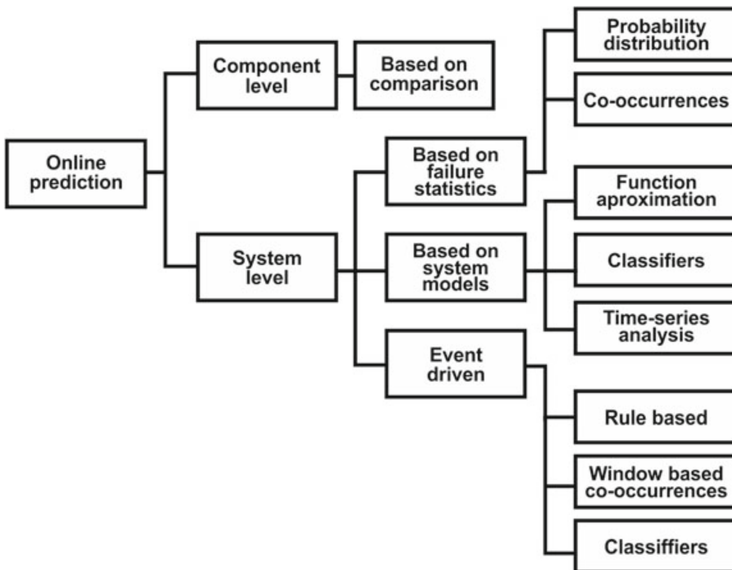

Fig. 2.12 Online failure prediction taxonomy

of good components with failed ones. A couple of studies from different fields that fit in this category are [8, 60]. For the HPC community, one example is [88] in which matrices are used to record system performance metrics at every interval. The algorithm afterwards detects outliers by identifying the nodes that are far away from the majority. One example is [44], where the authors implement their own data collection module that gathers relevant data across the system and assembling them into a uniform format. In the second step they apply two feature extraction techniques: PCA and ICA to generate matrices with lower dimensionality and in the last step the nodes that are far away from the majority are determined and considered potential anomalies. Their data mining algorithm is specifically designed for HPC systems and the results are different than previous studies.

The second level is represented by system level failure prediction, in which monitoring daemons observe different system parameters (system log, scheduler logs, performance metrics, etc.) and investigate the existence of correlations between different events. In the last couple of years, a significant number of papers have been proposed that focus on providing predictions by analyzing different HPC systems. However, most predictors are able to use the information extracted in the training phase for only short prediction span after which a new training phase is required. For example, [89] is using almost 3 months of training for predicting only half a month of execution. When dealing with real long time execution of a HPC system, the results of this type of prediction are unknown and can become unusable for real large-scale applications.

System level failure prediction has several categories:

Prediction Based on Failure Statistic The basic idea of failure prediction based on failure statistics is to draw conclusions about upcoming failures from the aggregated occurrences of previous failures. This may include the time of occurrence as well as the types of failures that have occurred. The two sub-categories includes: Probability Distribution Estimation and Co-Occurrence. Prediction methods belonging to the first category try to estimate the probability distribution of the time to the next failure from the previous occurrence of failures. The second type of failure predictors use the fact that system failures can occur close together either in time or in space (e.g., at proximate nodes in a cluster environment). This can be exploited to make an inference about failures that might come up in the near future.

Prediction Based on System Models The motivation for analyzing periodically measured system variables such as the amount of free memory or CPU usage in order to identify an imminent failure is the fact that some types of errors affect the system even before they are detected. The key notion of failure prediction based on monitoring data is that some errors can be grasped by their side-effects on the system such as exceptional memory usage, CPU load, disk I/O, or unusual function calls in the system. These side-effects are called symptoms. Symptom-based online failure prediction methods frequently address non-fail-stop failures, which are usually more difficult to grasp. The following subcategories are included in this category: function approximation that refers to mimicking a target value,

which is supposed to be the output of an unknown function of measured system variables as input data (this includes stochastic models, regression and machine learning); classifiers where failure prediction is achieved by classifying whether the current situation is failure-prone or not (this includes for example Bayesian networks); and time-series analysis where sequences of monitored system variables are treated as time series and time-series analysis is used in order to predict outlier moments in the series.

Event-Driven Failure Prediction Failure prediction approaches that use error reports as input data have to deal with event-driven input data. This is one of the major differences to system model failure prediction that uses symptom monitoring-based approaches, which in most cases operate on periodic system observations. Furthermore, symptoms are in most cases represented by metrics that are real-valued while error events are mostly discrete, categorical data such as event IDs, component IDs, log messages, etc.

2.6.2.1 Prediction Based on Failure Statistics

The predictors in this category assume the correlation between failures either in time or in space. In order to estimate the probability distribution of the time to the next failure, nonparametric methods as well as Bayesian predictors have been applied. Figure 2.14 presents how failures occur in a cluster, where the horizontal axes represents time and $M_1, M_2, \ldots, M_J$ denote the J nodes in use. ALL_{CL} refers to the set of failure event times over the entire cluster. ALL refers to the set of unique failure event times over the entire cluster. Several papers [36, 38, 84] study the failure inter-arrival time distribution as well as the failure co-occurrence properties for different systems. Their results are later used by predictors in this category so it is important to understand their results. The next section gives a brief overview of their most important findings that can later be used by failure predictors.

Cumulative Distribution of Failure Inter-Arrivals

Figure 2.13 presents a cumulative distributions (CDF) graph of the time between failures for an entire cluster over different epochs of time. This study by Heien et al. [36] analyzes a large scale system from the National Center for Supercomputer application, but their results are general and have been seen on other systems as well. In this figure, on the horizontal axes F_1, F_2 represent different failure types, while on the vertical axes each failure is analyzed for different time intervals. The solid lines indicate inter-event times for the cluster as a whole.

These time intervals between consecutive events correspond to the time line labeled ALL_{CL} in our previous figure (Fig. 2.14). It is visible that some failures are correlated and so occur nearly simultaneously on multiple machines. Current studies have shown that network or filesystem failures are a good example of such a spatial co-occurrence. In the logs generated by the cluster, a correlated failure appears as sequence of time-clustered events of the same failure type across machines. For this purpose, different studies assume different time thresholds for determining when

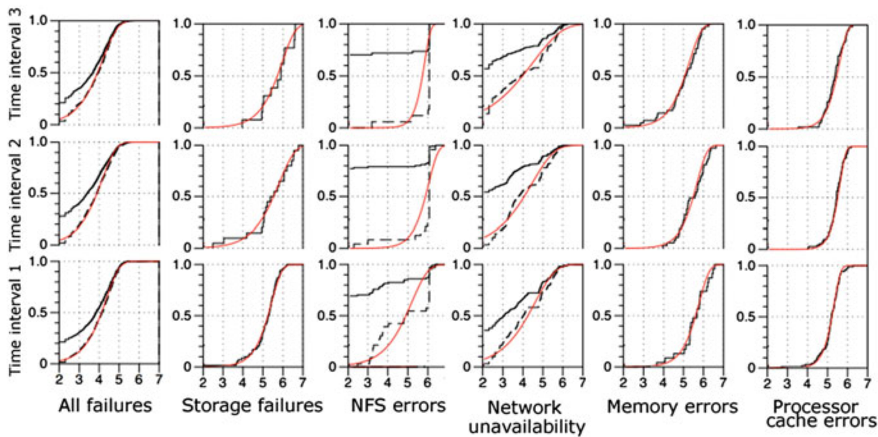

Fig. 2.13 Failure inter-event cumulative distributions for different epochs (*Solid line* cluster as a whole, *dotted line* discounts simultaneous failures, *red line* best fit)

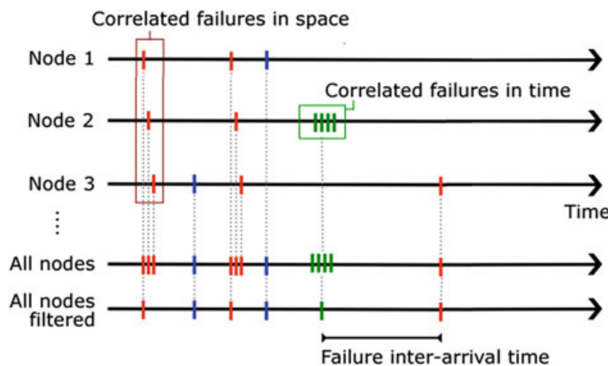

Fig. 2.14 Example of failure co-occurrence

failures are correlated. Depending on the system administrators expertise, one needs to assume that a correlated failure occurs when the time separating two consecutive failure events is less than a given threshold. Usual values for this threshold can range between tens of seconds, 1 min and several minutes.

After merging all simultaneous failures as a single failure, the resulting cumulative distribution is shown as the dotted line in Fig. 2.13. These time intervals between failures correspond to the time line labeled ALL in our previous figure (Fig. 2.14). The red line indicates the line of best fit. The best fit is found with any of the methods presented in Sect. 2.5.

Failure Correlation. All studies have found that some of the failures are correlated across different machines while other do not present this behavior. The failures that have a strong space correlation are shown in Fig. 2.13 as a large cumulative distribution at the start of the plot. Depending on the study, between 10 and 40 % of all the failures on different machines occur within 30 s of each other. One example

of this phenomenon is when multiple nodes are being restarted simultaneously after a shared power failure.

Besides space correlation between failures, it is important to understand and investigate whether different failure types are correlated, such as memory and processor failures being correlated due to common causes like overheating or motherboard failure. To determine whether different failures are time-correlated between machines, current research is dividing the life interval of a machine into time intervals of different lengths and noting the existence of failures in each period. Thus, the value of each period is 1 if a specified failure type occurred and 0 otherwise. In general, each study chooses the time interval length of a couple of hours since most failure events are separate by several hours. The cross-correlation is then calculated between all combinations of failure types, with a cross-correlation of 1 indicating exact correlation (at some time delta) between two failure patterns. When looking at the entire lifetime of a cluster and on broad failure categories, studies have shown that the average cross-correlations between different failures over all time intervals ranged from 0.04 to 0.11, none of which indicate strong positive or negative correlation. However, when analyzing shorter periods of time and/or specific categories, there is a visible correlation between different failure types. For this purpose, we present the results of a study that investigates the daily failure probability of certain general types of failures following another failure.

The authors looked at the probability that a node will fail within 24 h following a failure of a particular type. At the same time they are also looking at the percentage of cases when a node failure of any type follows a particular type of failure within an hour window. The percentage of cases when a failure of a particular type follows any failure within a one hour time window is also investigated. The results are presented in Fig. 2.15.

Figure 2.15a shows what types of failures are good precursors for other failures and Fig. 2.15b shows the types of failures that have precursors. In general, many failures seem to follow environmental and network failures. Also, by looking at Fig. 2.15c, we observe that these failures in general affect a large number of nodes which suggest they propagate not only in time but also space. All the results seem to indicate a strong correlation in space and time for failures affecting a cluster. These studies have encouraged the development of predictors based on failure statistics and co-occurrences.

Statistical Prediction

In order to estimate the probability distribution of the time to the next failure, non-parametric methods as well as Bayesian predictors have been applied. In [20], the authors investigate reliability prediction by analyzing a decade of field data made available by Los Alamos National Lab. They focus on investigating the impact of factors such as the power quality, temperature, fan and chiller reliability, system usage and utilization, and external factors, such as cosmic radiation, on system reliability. They observed that some types of failures increase the likelihood of follow-up failures more than others and that this information can be used for creating effective failure prediction models based on root cause distribution.

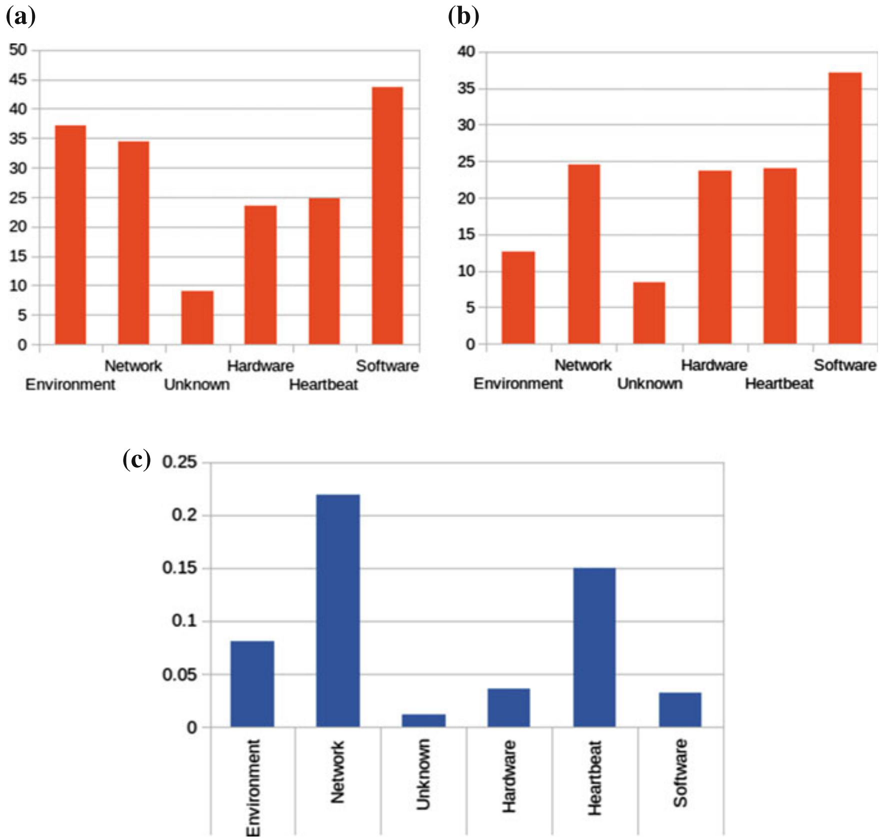

Fig. 2.15 Correlations between failures. **a** Probability of having a failure of any type after a failure of type X. **b** Probability of failure of type X following another failure of any type. **c** The probability that any node-failure follows a failure of type X

Bayesian failure prediction has the goal of estimating the probability distribution of the next time of failure by benefiting from the knowledge obtained from previous failure occurrences in a Bayesian framework [15, 35]. In [35], the authors use a mixture model of naive Bayes clusters trained by using expectation-maximization algorithm in order to predict disk failures.

Another paper [59] uses Bayesian statistics to develop an anomaly detection/ prediction system that employed naive Bayesian networks to perform intrusion detection on traffic bursts. Their model has the capability to potentially detect distributed attacks in which each individual attack session is not suspicious enough to generate an alert.

Due to sharing of resources, system failures can occur close together either in time or in space (at a closely coupled set of components or computers). As mentioned in the previous subsection, it has been observed several times that failures

occur in clusters in a temporal as well as in a spatial sense. Liang et al. [48] choose such an approach to predict failures of IBM's Blue Gene/L from event logs containing reliability, availability and serviceability data. The key to their approach is data pre-processing employing first a categorization and then temporal and spatial compression: temporal compression combines all events at a single location occurring with inter-event times lower than some threshold, and spatial compression combines all messages that refer to the same location within some time window. Prediction methods are rather straightforward: using data from temporal compression, if a failure of type application I/O or network appears, it is very likely that a next failure will follow shortly. If spatial compression suggests that some components have reported more events than others, it is very likely that additional failures will occur at that location. Fu and Xu [25] further elaborate on temporal and spatial compression and introduce a measure of temporal and spatial correlation of failure events in distributed systems. A different approach is given in [49, 82], where the authors investigate parameter co-occurrences between different application log messages for extracting dependencies among system components. The authors mine dependencies from the tuple-form representations of the log messages looking for patterns that could indicate a failure in the system that prevented tasks from completing.

Statistical methods do not have extremely good results when looking at the entire set of failures affected by a system, however, they have proven a great insight for some particular failures. The study that uses these types of prediction and that has the best results shows 50 % precision and 48 % recall on a 350 node-based cluster [82]. Location prediction is one of the limitations of these methods, so it is no surprise that the same methods applied on larger and more complex systems give lower results. However, these types of method can be used to predict a state of instability for a node, without being able to give an exact time when the failure will occur. When using large time windows for when the node might fail, the methods give better results.

Another example of using the statistical failure predictor is by combining it with a rule-based method. In [64], the authors use a meta-learning predictor to choose between a rule-based method and a statistical method depending on which one gives better predictions for a corresponding state of the system and their results show that they can obtain better predictors, showing a 90 % prediction and 70 % recall on small clusters.

2.6.2.2 Prediction Based on System Models

One frequently used method is represented by regression techniques where parameters of a function are adapted such that the curve best fits the measurement data, for example by minimizing the mean square error. The simplest form of regression one can use in order to develop a predictor is curve fitting of a linear function. For this purpose, system administrators analyze either performance metrics or the count of particular events in order to predict future failures. The usual performance metrics that represent the input for regression models are temperature and CPU usage. A steady increase in temperature until exceeding a given threshold for a given cabinet

could indicate future problems in several components on the corresponding cabinet. At the same time, an increase in the count or frequency of correctable memory errors usually can indicate a future occurrence of an uncorrected memory error.

In [2], the authors apply deterministic function approximation techniques such as splines to characterize the functional relationships between the target function and input data. Deterministic modeling offers a simple and concise description of system behavior with few parameters. They consider the problem of automated modeling in server-type applications whose performance degrades depending on the "work" done since last rejuvenation. Their input data is for example the number of served requests by a system. In this case a failure can be seen as a type of performance degradation caused mostly by resource depletion. This is common for filesystems for example where failures on the metadata or object target server propagate at the application level as performance diminishes.

Pattern recognition techniques operate on sequences of error events trying to identify patterns that indicate a failure-prone system state. The most used method for pattern recognition is by far the Markov chain model. The approach is based on the assumption that failure-prone system behavior can be identified by characteristic patterns of errors. The most used technique by current system administrators as well as research in this field is to use a hidden Markov model.

An HMM is mathematically equal to a stochastic finite automaton defined by 5 tuples (Q, $\sum$, Δ, π, O), where Q $= \{q_1, q_2, \ldots, q_N\}$ is a finite set of states , $\sum$ is an alphabet of output symbols, $\Delta = \{a_{ij}$ with $1 \leq$ i, j $\leq$ N, $\sum_{j=1}^{N} a_{ij} = 1\}$ is a state transition probability distribution and $\pi = \{\pi_i$ with $1 \leq$ i $\leq$ N, $\sum_{j=1}^{N} \pi_i = 1\}$ is an initial state distribution and O is the set $\{e_j$(x) with $1 \leq$ j $\leq$ N, $\sum_{j=1}^{N} e_j(x) = 1\}$ of output symbol probabilities.

HMMs are called hidden because only the outputs can be observed from outside and the actual state q_i is hidden from the observer. The objective of this type of predictors, as before, is to assess the risk of failure for some time in the future. Similar to the previous methods, here failures are predicted by analysis of error events that have occurred in the system by using the property of systems that the frequency of error occurrence increases before a failure occurs. Given a sequence of observations (events generated by the system in the past) a Hidden Markov Model is successfully developed from a probabilistic finite state automata. The overall probability of the given sequence can be afterwards found by sequence likelihood.

Building a failure predictor from a sequence of error events takes two steps: (i) firstly the number of states in the automata needs to be fixed and (ii) secondly the probability that leads to failure needs to be computed. All methods used in the literature construct the best automaton governing the given data. There are a number of variations on HMM problems depending on how many states a system administrator would take. The simplest model has one state, the most complex model has a state for each and every symbol of the data but certainly neither extreme is justified. Figure 2.16 presents two models, one with two states and one with five that both fit the input data AAACACBBBCCBBAACAAACB. The number of states for

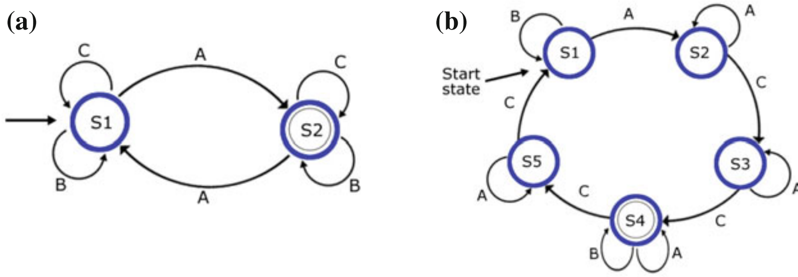

Fig. 2.16 State models. **a** Two states automaton. **b** Five states automaton

the Markov model is either chosen empirically based on the previous knowledge of the failure and its behavior or by a hierarchical algorithm that fits the data for increasing number of states and the best one is chosen. In general, current work is using between 5 and 15 states depending on the analyzed failure and system.

In [66] the authors propose to use hidden semi-Markov models (HSMM) in order to add one additional level of flexibility to the theoretical method. Two HSMMs are trained from previously recorded log data: one for failure and one for non-failure sequences. Online failure prediction is then accomplished by computing the likelihood of the observed error sequence for both models and by applying Bayes decision theory to classify the sequence (and hence the current system status) as failure-prone or not.

The second step implemented by [33] uses two semi Markov models that quantify the reliability of a node in the overall system. In the process the method identifies nodes that tend to be the source of a large number of failures and predicts the reliability of these nodes. The first discrete-time semi-Markov model is built for each system where state transitions are driven by functions derived from the distributions fitted to the result of the neural-gas filtering analysis. The second semi-Markov process computes transaction probabilities and event arrival rates directly from event observations.

Other methods include covariance models with an adjustable timescale to quantify the temporal correlation and a stochastic model to describe the spatial correlation. In [25], the authors build a neural network to approximate the number of failures in a given time interval. The set of input variables consists of a temporal and spatial failure correlation factor together with variables, such as CPU utilization or the number of packets transmitted by a computing node.

Support vector machines (SVM) are another popular modern pattern recognition and regression algorithm that is used by current research for offering model-based failure prediction. The principle of the SVM classifier is to project the data into a higher dimensional space where the classes are separated by a linear hyperplane which is defined by a small set of support vectors. There are open source or free tools

already developed that implement a wide range of SVM algorithm, like the MySVM package developed by Ruping [63]. Other researchers develop their own, depending on the needs of their analysis. Murray et al. [55] have used the MySVM package on their data, gathered with the Self-Monitoring and Reporting Technology (SMART) system in order to predict failures of hard disk drives.

Depending on the analyzed data set, predictors based on system models can be accurately used in diverse systems. For disk drives, the predictor detects around 50 % of failures with only a 0.6 % false alarms. If even lower false alarm rates are needed, studies have shown that changing the combination of attributes can offer a prediction for 25.0 % of failures with no measured false alarms. For disk drives it is important to have a very low false alarm rate since it reduces the number of returned good drives, thus lowering costs to manufacturers of implementing improved SMART algorithms.

2.6.2.3 Event-Driven Prediction

Failure prediction methods in this category analyze the events generated by the system and derive a set of rules/patterns/correlations between different events. In general, the rules express temporal ordering of events in the form “if errors A and B occur within x seconds, then error C occurs within y seconds with probability P.” Several parameters such as the maximum length of the data window, types of error messages, and ordering requirements have to be prespecified.

The predictors follow the same work flow (Fig. 2.17): (1) the input data is preprocessed so that it fits the standards required by the analyzing modules; (2) data mining algorithms are applied on the preprocessed data in order to extract patterns and rules between events; and (3) the system is monitored and predictions are triggered based on the extracted rules. In the preprocessing phase, failures are divided into different classes after which rules are being extracted for each class. In general, the groups used are either general groups of failures, like network, hardware or software,

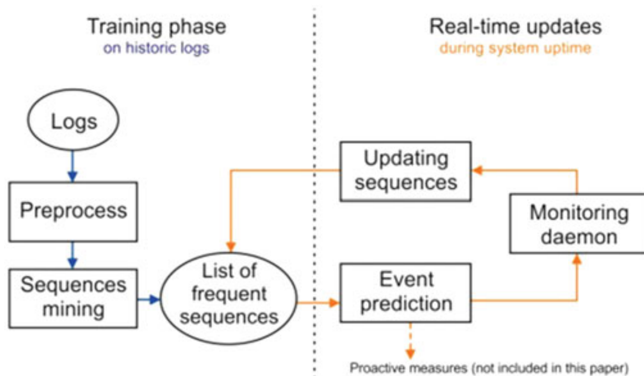

Fig. 2.17 Prediction methodology

or more specific, like memory ECC errors, cpu cache error and so on. Most of the research in this field uses a predefined number of classes, that are usually defined by the system administrator. For analyzing general types of events a predefined number of classes is enough. However, when moving the analysis at a more specific layer, a more flexible method of defining events is necessary. Since systems can change during the course of their lifetime, novel events may appear, thus the number of classes may need to change in time. In [33] the authors propose a clustering method that groups events automatically. The method proposed in [27] is another way of automatically extracting all events generated in the past by a system and keeping them accurate by monitoring everything that is generated by the system. Table 2.10 presents examples of event types found on different systems with their corresponding regular expressions.

Extensive research has been focusing on using system logs, scheduling logs, performance metrics or usage logs in order to extract a correlation between events generated by a system. There are numerous methods, starting with simple brute force extraction of rules between nonfatal events and failures [67] and going to more sophisticated techniques. Event logs are a rich source of information for analyzing the cause of failures in cluster systems. However, the size of these files has continued to increase with the ever growing size of supercomputers, making the task of analyzing log files a hard and error prone process when handled manually.

Table 2.11 presents the number of events generated by systems throughout time, starting with LANL systems from 1996 to the current Blue Water system from 2014. The number of events generated by the Blue Waters system is two orders of magnitude larger than previous generation systems and it keeps increasing as new components

Table 2.10 Examples of event types

System	Template	Event type
BGL	Failed to configure resource mgmt subsystem err = d+	Processor cache error
Blue Waters	* panic - * syncing: *	LBUG
Blue Waters	Lustre: * @@@ Request sent has failed due to network error: n+	MDT failure
BGQ	Component state change: component * is in the * state *	Info notification
BGQ	ECC-correctable single symbol error: DDR Controller d+, failing SDRAM address *, BPC pin *, n+	DDR single symbol error

Table 2.11 Log file statistics

System	Events/Day	Total event types
Blue Waters	11.5 GB (90 mil events)	10,495
Blue Gene/P	8.12 MB (120,000 events)	252
Blue Gene/L	5.76 MB (25,000 events)	186
Mercury	152.4 MB (1.5 mil events)	563
LANL systems	433,490 in 5 years	53

are added in the system. Filtering both in time and space is frequently necessary for several data mining algorithms in order to reduce the input data set and to cluster all failures that belong to the same problem.

Filtering Failures. Failure events from the same location often occur in bursts or clusters of notifications. Some clusters are homogeneous, with their failures having the same type, while others are heterogeneous and their failures usually report different attributes of the same event. For example, a memory failure cluster is heterogeneous, in the sense that every consecutive entry reports the same memory error referring to the same unique system state. Filtering failures in time requires to coalesce a cluster into a single failure record. Identifying such bursts from the log, requires sorting/grouping all the failures according to the associated general/specific category and location. Studies that focus on the application level might also consider the job ID. Failures that occur within the same subsystem and that are reported by the same location (and/or the same job), are filtered into a single entry if the gaps between them are less than a specified threshold. Figure 2.18 presents the usual number of remaining failure records after using this filtering technique with different threshold values. In general, the threshold is chosen by the system administrator and is a fix value between 5 and 10 min depending on the study.

Bursts of messages can occur on multiple locations as well, especially since HPC systems host parallel jobs. For example, all the tasks from a job will experience the same I/O failure if they access the same directory. At the same time, a network or a file system failure is very likely to be detected by multiple locations. As a result, many studies consider it essential to filter across locations. Spatial filtering removes failures that occur closer in time than a given threshold for the same event type and/or from the same job, but from different locations. Similarly, Fig. 2.19 presents the number of remaining failure records after spatial filtering with different threshold values. The values presented in the figure are for the Blue Gene/L system but the filtering percentage is similar for other systems as well.

In general, choosing the appropriate value for spatial filtering is easier than the temporal filtering, seen in Figs. 2.18 and 2.19 by the fact that the resulting failure count is not very sensitive to the threshold value for spatial filtering. Most studies chose the same threshold as for the temporal filtering, between 5 and 15 min.

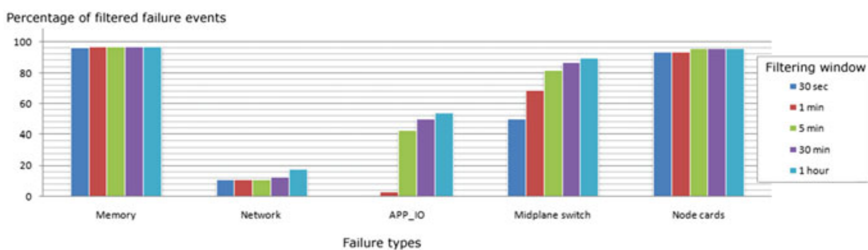

Fig. 2.18 Percentage of failures filtered with temporal filtering for different thresholds

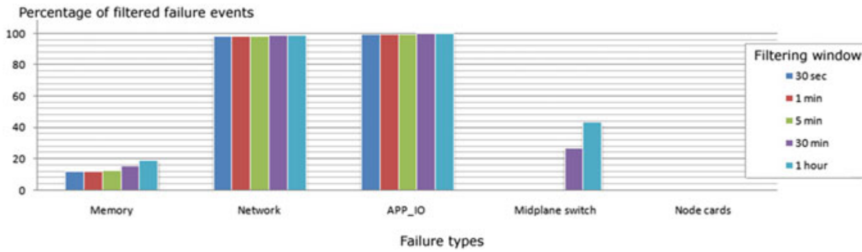

Fig. 2.19 Percentage of failures filtered with spatial filtering for different thresholds

Extracting Events' Behavior. Large scale systems experience a large variety of events during their lifetime and they output notifications for each of them. Once an error is triggered for one component, either software or hardware, there is not a consistent way of registering how the system will behave. For example, in case a node experience a network failure and is incapable of generating log messages, the failure is announced in the log files by a lack of generated messages. Conversely, some component failures may cause logging a large numbers of notifications. For example, memory failures can result in a single faulty component generating hundreds or thousands of messages in less than a day.

At the same time, some errors are notified by a single message. For example on NCSA's Mercury system, NFS related errors that indicate unavailability of the network file system for a machine, need a single instance of the generated message to notify a potentially fatal failure to an application using this resource. However, this is not always the case. Memory errors, for example, are often correctable by the ECC capabilities, so only when the system generates a large numbers of these errors in a short time span, it is likely to have a permanent failure of a component.

Each failure type behaves differently and affects the systems differently. An alternative to the methods that simply apply the same data mining algorithm on the pre-process data is to model the normal behavior of the system for each event that might be generated. By characterizing the way a failure affects these models the input data can be transformed into a unified format that can be used as an input in the data mining algorithm. Such an example is [29], where the authors extract all the event types and then plot the number of occurrences per time step for each event type into separate signals. Each event type has occurrences at different times in a system lifespan. By choosing a sampling rate and mapping the number of messages generated by the system in each sampling slot and for each event type, time series of number of occurrences for each event type can be extracted. The obtained time series are regarded as signals and can be analyzed with signal processing modules. The sampling rate is chosen differently depending on the characteristics of each signal.

Extracting all the signals for different systems has shown that there are three types of events: periodic, silent, and noisy. An example of each of the three types can be seen in Fig. 2.20. Usually, periodic signals are generated by daemons or by events that deal

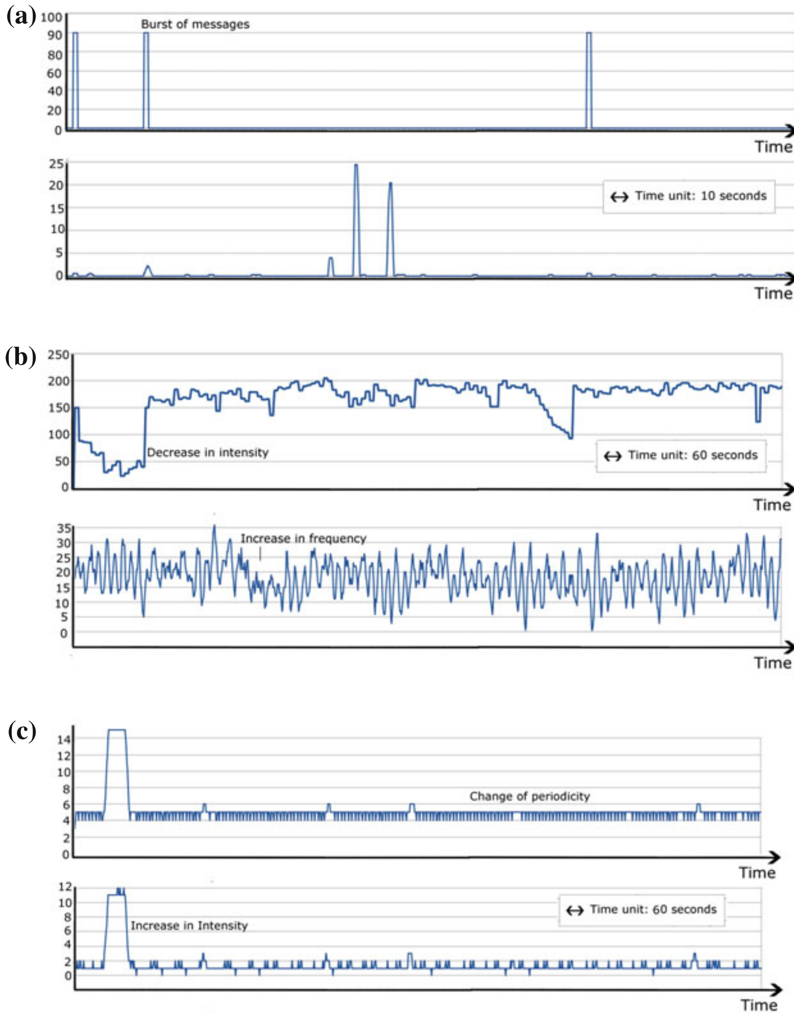

Fig. 2.20 Different signals generated by HPC systems. **a** Silent signals. **b** Noisy signals. **c** Periodic signals

with monitoring information. Examples of these signals are presented in Fig. 2.20c. We call the second type silent signals because most of the signal is a flat line around the zero value, and only from time to time there is a burst of messages. This type is presented in Fig. 2.20a and is usually characteristic for error messages, for example in case of PBS (Portable Batch System) errors. Noisy signals are chatty signals that send notifications very often. Two examples of such events are presented in Fig. 2.20b. These types of signals are usually warning messages that are generated both in case of normal behavior and failures, usually preceding error messages or when a problem

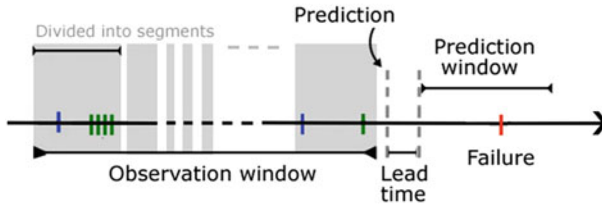

Fig. 2.21 Rule extraction methodology

is corrected. We observed that even some failure events can experience this behavior, for example in the case of memory errors that could be corrected by ECC. Anomaly detection can be applied for each signal and a unified model can be created for each.

Prediction

Event-driven predictors are in general characterized by two main methods: (1) period-based approaches and (2) rule-based approaches with a few variations. Figure 2.21 shows the methodology for the first type of method. In general, there is an observation window, a lead time and a prediction window. The observation window is usually composed of a set of consecutive time intervals I $= I_1, I_2, \ldots, I_n$, either of the same size (like in [64]) or dynamically adjusted (like in [26]). The observation window is used to collect evidence that determines whether a failure will occur within the prediction window. *A priori* based data mining algorithms are used in order to find correlated events that occur frequently together in the same time interval.

The second method is called rule-based prediction and it uses an observation window in order to extract rules between different failures and between failures and events. This is done either by brute force (for smaller systems) by investigating all to all correlations or with more advanced data mining solutions, like the Grite algorithm [40].

The period-based method has been applied in [85]. The authors are using a three-phase failure predictor for the Blue Gene/L systems: event preprocessing where the raw RAS log is cleaned and categorized; the base prediction phase where different base learning methods are applied on the preprocessed log to identify fault patterns and correlations. Similarly, [64] uses a period-based predictor close to [85] in that it uses a fixed time window for creating the time intervals. The method consists of two steps: (1) a preprocessing step that converts sys-logs into a data set that is appropriate for running classification techniques by extracting a set of features. These features can accurately capture the characteristics of failures. (2) In the next step the method applies different classifiers besides the rule-based method in order to compare the results (a rule-based classifier and a Bayesian network that combines both methods). A different approach on the classical period-based prediction is presented in [26] where the time intervals are not fixed, but they are rather defined by the data. For each failure type, the time intervals are defined by the time between two consecutive failures. The same algorithm is then applied to find frequently occurring events for each failure.

In [43], the authors are using a meta-learning predictor to choose between a rule-based method and a statistical method depending on which one gives better predictions for a corresponding state of the system. They show that different prediction methods capture different failures so combining predictors together is beneficial and has the potential to increase the results drastically. Another successful rule-based approach is presented in [28] where the authors combine signal analysis with data mining in order to extract the rules. A modified version of the gradual item set mining algorithm is used for extracting patterns of the form "the more/less $X_1, \ldots,$ the more/less X_n". In our case the algorithm gives us rules of the form "X_1 has anomalies, ..., X_n has anomalies" where X are events in the system. This method has the advantage of extracting multiple event correlations instead of only pairs. The results indicate that this type of prediction has good results on its own and gives enough lead time for different preventive actions.

Table 2.12 presents the prediction results for the most successful studies in literature up to date. While some of the results seem extremely good, at a closer look it is clear that some are obtained either by using long training phases for only a couple of days of prediction, or not considering the lead time between when the prediction is done and when the failure occurs. Moreover, most of the presented methods do not provide any location information. This makes it impossible for proactive methods to know which application processes should be migrated. At the same time, predictions with location information will enable checkpointing data only on those failure-prone components, thereby avoiding system-wide checkpointing which is significantly time consuming. When filtering out the predictor's that have a small lead time or do not offer location prediction, the results are slightly lower, the best study offering around 50 % recall and 80 % precision.

For the second problem, Yu et al. [85] offers a study of the influence that the observation window has on the prediction's results. They look at both period-based and event-based approaches. The accuracy achieved by the period-based approach is growing with the increase of the observation window, while for the event-driven approach it is the opposite. The period-based approach achieves its best performance, in general, when the observation window is bigger than a couple of days, while the

Table 2.12 Prediction results for different state of the art-related work

System	Method	Precision	Recall	Lead time (s)	Citation
BGL	Rule-based method	0.7	0.3–0.4	5 min	[48]
BGL	Statistical method	0.5	0.48	–	[64]
BGL	Multiple methods	0.9	0.7	–	[64]
BGP	Rule-based method	0.4/0.4/0.35	0.8/0.7/0.6	0/300/600	[89]
BGP	Rule-based method	0.5	0.5	30 min	[85]
LANL systems	Signal analysis	0.9	0.5	10 s	[29]
BGP	Signal analysis with rule-based method	0.7	0.6	10 s	[28]

event-driven one reaches its peak when the observational window is as low as a couple of hours. Looking at the entire results, the event-driven approach outperforms the period-based approach significantly.

In general, thresholds, like the observation window, greatly influence the results. Figure 2.22 presents the prediction results for several methods when varying one of the parameters. Event-driven approaches are in general sensitive to the events that occur shortly before a failure. That is the reason why, in general, event-driven predictors will achieve their best performance with small observation windows. On the contrary, the period-based approaches takes more benefits from past statistical information which makes them achieve their best results with larger observation windows. There are also thresholds, like the correlation threshold in the first graph from Fig. 2.22, that offer trade-offs between precision and recall. Depending on how the predictions are used, a larger coverage or a higher accuracy might be desirable.

A failure in the system most of the time does not seem to impact a massive numbers of jobs. At a closer analysis, we observed that only around 44 % of the failures lead to at least one application crash, out of which the most surprising were filesystem failures.

Filesystem failures are one of the main reasons for a low recall for current large-scale system. For example, on the Blue Waters system, the Lustre Metadata failures have very few precursors since most of them occur at the same time with the actual failure. Metadata servers for the Lustre filesystem store namespace metadata, such as filenames, directories, access permissions, and file layout. When applications detect an MDT (Meta Data Target) failure, they connect to the the backup MDT and continue their execution. Just in less than 17 % of the cases, applications having trouble connecting to the back-up MDT fail. During an OST (Object Storage Target) failure, when applications attempt to do I/O to a failed Lustre target, these are blocked waiting for OST recovery. An application does not detect anything unusual, except that the I/O may take longer to complete. Rarely, when an OST is marked as inactive, the file operations that involve the failed OST will return an IO error and the application might be terminated.

In general, prediction from the application's point of view is more complex and differs in results compared to the one for system failures. Moreover, when analyzing the prediction results from the application's perspective, the online methodology is

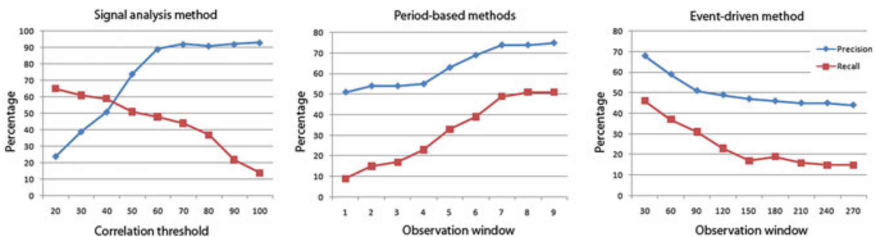

Fig. 2.22 Evaluation of different approaches using different thresholds

highly sensitive to the lead time offered by each prediction. The lead time represents the time interval between when the prediction is triggered and when the failure actually occurs. Location prediction gets a slightly new meaning as well when application crashes need to be predicted instead of system failures. If a given method predicts a failure correctly in time, but the failure occurs on a different node, all methods will give a false negative and a false positive in the final results. However, if an application was running on multiple nodes, one of which corresponds to the predicted node, and the application takes global preventive actions, the mis-predicted failure could be masked. Depending on the fault avoidance strategy, a predictor that only looks at applications as a whole and not as a set of running nodes could increase the recall significantly. By taking the lead time and the new definition of location prediction into consideration one can recompute the results for different methods. We observed that the prediction results have, in general slightly better values than when applying the same method for system failures. For the method presented in [28] and when analyzing the Blue Waters system, the application crash prediction has a higher recall value with 5 % for the same precision.

2.6.3 Checkpointing Challenges

We consider that the prediction performance presented in the previous section has the potential to be used in the future in order to reduce the effects of failures on application. For this purpose, failure prediction is useful only when coupled with a proactive failure management that tries to apply countermeasures. The decision to actually trigger a countermeasure may follow a complex process involving (i) cost of the actions, (ii) the confidence in the prediction and (iii) the effectiveness and complexity of the actions. These promising advances in failure prediction precision and recall open the possibility to reduce drastically the rework time by actually checkpointing right before the failure; a technique know as proactive checkpointing.

However, proactive checkpointing alone, cannot systematically avoid re-executing the application from scratch if failures are not perfectly predicted. Since executions on large scale HPC systems are very expensive (in time and energy), taking the risk of long (potentially near to full) re-executions is unacceptable. Therefore, failure prediction and proactive checkpointing should be combined with periodic checkpointing. Nevertheless, little is known about the benefits of failure prediction and proactive checkpointing when combined with periodic checkpointing.

Most predictions offer small lead time windows for proactive actions to be taken. To be consistent with short term prediction, fault tolerance strategies require significantly improvement. One promising direction is multi-level checkpointing. There are currently two environments providing multi-level Checkpoint/Restart: SCR (Scalable Checkpoint/Restart) [54] and FTI (Fault Tolerance Interface) [7]. Recent results show that a process context of 1GB can be saved in 2–3 s in local SSD (i.e. 2 SSD mounted in RAID0). Such checkpoint speed is orders of magnitude faster than checkpointing on remote file system which requires tens of minutes in current petascale

systems and may require many hours in projected exascale systems. An experiment with FTI on a large scale execution (1/2 million GPU cores) of an earthquake simulation on a hybrid system composed of CPU and GPUs demonstrates very low overhead on the execution time (i.e., less than 10 %) when using such checkpoint strategy compared to no fault tolerance. Other research results demonstrate that checkpointing on remote node memory is even faster than on local HDD or SSD [86]. These results demonstrate that proactive checkpoints can be taken even with a few seconds before the predicted failure happens. However, proactive checkpointing introduces a whole new dimension with several challenges:

- To decrease the checkpoint size and maximize efficiency many applications rely on user-guided checkpointing, in which users specify points in the code where to checkpoint, so that the amount of data that need to be saved is minimal. However, upon a failure prediction, the checkpoint is triggered by the prediction runtime and the application may be in the middle of a complex kernel execution that requires a high memory footprint. Thus, new ways of combining user-guided checkpointing with proactive checkpointing need to be found.
- Furthermore, it is important to remember that the application needs to restart after the failure and still produce correct results. This is the classic checkpointing coordination problem that may imply the use of a fault-tolerant protocol. In application level checkpointing, the coordination is implicit, while in system level checkpointing capturing the state of the execution is explicit and relies on a fault-tolerant protocols. If the approach relies on coordinated checkpointing or on hierarchical fault-tolerant protocols [32], the coordination (global or partial) needs to be fast enough to store the state of the application before the failure occurs.

Any proactive checkpointing implementation that does not provide high performance solutions for these two problems will not be able to work efficiently in combination with a prediction strategy. There are several theoretical studies that propose to combine classic periodic checkpointing with proactive fault tolerance actions in order to study the theoretical benefit of such approaches in case such an implementation will work on future systems. One such example is presented in Aupy et al. [3], where the authors propose a fault-tolerant strategy that uses the prediction alerts to compute an optimal checkpointing interval. In their follow-up work [4], the authors assume that the fault-prediction systems that do not provide exact prediction dates, but instead time intervals during which faults are predicted to strike, with different probabilities at each moment of time. Li et al. [47] consider a different model of prediction mechanism that provides a probability of failure when the application ask for a prediction. Moreover, they consider a specific application model where proactive checkpoints or migration can be performed at a predefined location during the execution. Cappello et al. [12] proposed two proactive fault tolerance strategies, both relying on a perfect prediction mechanism. The perfect prediction mechanism is supposed to have a 100 % recall, 100 % precision and enough lead time to perform either checkpointing or migration. Even though the scenario is not realistic since there is no prediction method that can offer these results, it can show the trade-off of combining prediction either with checkpointing or with migration.

In [10], the authors combine an existing multi-level checkpointing strategy with a short-term event driven prediction. The results show that the benefit of such an implementation can give a decrease in the waste of the checkpointing strategy of around 10 % for the Blue Waters system and over 20 % for Blue Gene/L. Considering the extra 3–4 % overhead induced by this hybrid approach, the overall benefit becomes over 15 % for smaller system, and around 7 % for current Petascale computing.

Long-term predictions can be combined with checkpointing in order to reduce the I/O overhead and compute resource wastage induced by current checkpointing strategies. In [76], the authors propose a couple of methods that place checkpoints by taking advantage of the temporal locality in failures, instead of naively taking periodic checkpoints.

Acknowledgments Ana Gainaru's work is supported by the Blue Waters sustained-Petascale computing project, funded by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. This chapter is build on material from publications co-authored with numerous colleagues. The authors would like to thank Leonardo Bautista-Gomez, Mohamed Slim Bouguerra, Jeremy Enos, Joshi Fullop, Eric Heien, Derrick Kondo, and William Kramer.

References

1. Anaya IDP, Simko V, Bourcier J, Plouzeau N, Jézéquel J-M (2014) A prediction-driven adaptation approach for self-adaptive sensor networks. In: Proceedings of the 9th international symposium on software engineering for adaptive and self-managing systems, SEAMS 2014. ACM, New York, pp 145–154
2. Andrzejak A, Silva L (2007) Deterministic models of software aging and optimal rejuvenation schedules. In: 10th IFIP/IEEE international symposium on integrated network management, IM'07, pp 159–168
3. Aupy G, Robert Y, Vivien F, Zaidouni D (2012) Impact of fault prediction on checkpointing strategies. Rapport de recherche RR-8023, INRIA
4. Aupy G, Robert Y, Vivien F, Zaidouni D (2013) Checkpointing strategies with prediction windows. In: 2013 IEEE 19th Pacific Rim international symposium on dependable computing (PRDC), pp 1–10
5. Avizienis A, Laprie J-C, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE J Dependable Secur Comput 1:11–33
6. Bairavasundaram LN, Goodson GR, Pasupathy S, Schindler J (2007) An analysis of latent sector errors in disk drives. In: Proceedings of the 2007 ACM SIGMETRICS international conference on measurement and modeling of computer systems, SIGMETRICS'07. ACM, New York, pp 289–300
7. Bautista-Gomez L, Tsuboi S, Komatitsch D, Cappello F, Maruyama N, Matsuoka S (2011) FTI: high performance fault tolerance interface for hybrid systems. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis, pp 1–32
8. Bolander N, Qiu H, Eklund N, Hindle E, Rosenfeld T (2009) Physics-based remaining useful life predictions for aircraft engine bearing prognosis. In: Conference of the prognostics and health management society
9. Bouguerra MS, Gainaru A, Cappello F (2013) Failure prediction: what to do with unpredicted failures? In: 28th IEEE international parallel and distributed processing symposium

10. Bouguerra MS, Gainaru A, Cappello F, Gomez LB, Maruyama N, Matsuoka S (2013) Improving the computing efficiency of HPC systems using a combination of proactive and preventive checkpointing. In: Proceedings of IEEE IPDPS 2013. IEEE Press
11. Cappello F, Geist A, Gropp B, Kale L, Kramer W, Snir M (2009) Toward exascale resilience. Int J High Perform Comput Appl 23:374–388
12. Cappello F, Casanova H, Robert Y (2010) Checkpointing versus migration for post-petascale supercomputers. In: 2010 39th international conference on parallel processing (ICPP), pp 168–177
13. Chen MY, Accardi A, Kıcıman E, Lloyd J, Patterson D, Fox A, Brewer E (2004) Path-based failure and evolution management. In: Proceedings of the international symposium on networked system design and implementation, NSDI'04, pp 309–322
14. Cotroneo D, Natella R, Pietrantuono R, Russo S (2014) A survey of software aging and rejuvenation studies. J Emerg Technol Comput Syst 10(1):8:1–8:34
15. Csenki A (1990) Bayes predictive analysis of a fundamental software reliability model. IEEE Trans Reliab 39:177–183
16. DeBardeleben N, Daly J, Scott S, Harrod W (2009) High-end computing resilience: analysis of issues facing the HEC community and path forward for research and development. National HPC workshop on resilience
17. Di S, Berrocal E, Bautista-Gomez L, Heisey K, Gupta R, Cappello F (2014) Toward effective detection of silent data corruptions for HPC applications. In: Proceedings of the 28th ACM international conference on supercomputing, SC'14
18. Dick T, Barkan C, Chapman E, Stehly M (2000) Predicting the occurrence of broken rails: a quantitative approach. In: Proceedings of the American railway engineering and maintenance of way association annual conference
19. Dongarra J, Beckman P, Moore T, Aerts P, Aloisio G, Andre J-C, Barkai D, Berthou J-Y, Boku T, Braunschweig B, Cappello F, Chapman B, Chi X (2011) The international exascale software project roadmap. Int J High Perform Comput Appl 25(1):3–60
20. El-Sayed N, Schroeder B (2013) Reading between the lines of failure logs: understanding how HPC systems fail. In: 2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN), pp 1–12
21. Elnozahy E, Bianchini R, El-Ghazawi T, Fox A, Godfrey F, Hoisie A, McKinley K, Melhem R, Plank J, Ranganathan P et al (2008) System resilience at extreme scale. Technical report for the defence advanced research project agency
22. Farr W (1996) Software reliability modeling survey. Handbook of software reliability engineering. McGraw-Hill, New York, pp 71–117
23. Feitelson DG (2002) Workload modeling for performance evaluation. Performance evaluation of complex systems: techniques and tools. Springer, Berlin, pp 114–141
24. Fiala D, Mueller F, Engelmann C, Riesen R, Ferreira K, Brightwell R (2012) Detection and correction of silent data corruption for large-scale high-performance computing. In: Proceedings of the international conference on high performance computing, networking, storage and analysis, SC'12. IEEE Computer Society Press, Los Alamitos, pp 78:1–78:12
25. Fu S, Xu C (2007) Quantifying temporal and spatial fault event correlation for proactive failure management. In: IEEE proceedings of symposium on reliable and distributed systems
26. Gainaru A, Cappello F, Fullop J, Trausan-Matu S, Kramer W (2011) Adaptive event prediction strategy with dynamic time window for large-scale HPC systems. In: Managing large-scale systems via the analysis of system logs and the application of machine learning techniques, SLAML'11. ACM, New York, pp 4:1–4:8
27. Gainaru A, Cappello F, Trausan-Matu S, Kramer W (2011) Event log mining tool for large scale HPC systems. In: Proceedings of the 17th international conference on parallel processing—volume part I, Euro-Par'11. Springer, Berlin, pp 52–64
28. Gainaru A, Cappello F, Snir M, Kramer W (2012) Fault prediction under the microscope: a closer look into HPC systems. In: Proceedings of 2012 international conference for high performance computing, networking, storage and analysis. IEEE Press

29. Gainaru A, Cappello F, Kramer W (2012) Taming of the shrew: modeling the normal and faulty behavior of large-scale HPC systems. In: Proceedings of IEEE IPDPS 2012. IEEE Press
30. Gertsbakh I (2000) Reliability theory: with applications to preventive maintenance. Springer, Berlin
31. Guan Q, Zhang Z, Fu S (2011) Ensemble of Bayesian predictors for autonomic failure management in cloud computing. In: 20th international conference on computer communications and networks, pp 1–6
32. Guermouche A, Ropars T, Snir M, Cappello F (2012) HydEE: failure containment without event logging for large scale send-deterministic MPI applications. In: 2012 IEEE 26th international parallel and distributed processing symposium (IPDPS), pp 1216–1227
33. Hacker T, Romero F (2009) An analysis of clustered failures on supercomputing systems. J Parallel Distrib Comput 69:652–665
34. Hacker TJ, Romero F, Carothers CD (2009) An analysis of clustered failures on large supercomputing systems. J Parallel Distrib Comput 69:652–665
35. Hamerly G, Elkan C (2001) Bayesian approaches to failure prediction for disk drives. In: Proceedings of the eighteenth international conference on machine learning, pp 202–209
36. Heien E, Kondo D, Gainaru A, LaPine D, Kramer B, Cappello F (2011) Modeling and tolerating heterogeneous failures in large parallel systems. In: Proceedings of 2011 international conference for high performance computing, networking, storage and analysis. ACM, p 45
37. Holmgren M (1996) Comparison between different methods for fatigue life prediction of bogie beams. Rakenteiden Mekaniikka, vol 29
38. Hwang A, Stefanovici I, Schroeder B (2012) Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. SIGARCH Comput Archit News 40(1):111–122
39. Javadi B, Kondo D, Vincent J-M, Anderson D (2011) Discovering statistical models of availability in large distributed systems: an empirical study of SETI@home. IEEE Trans Parallel Distrib Syst 22(11):1896–1903
40. Jorio D, Laurent A, Teisseire M (2009) Mining frequent gradual itemsets from large databases. In: International conference on intelligent data analysis
41. Kharbas K, Kim D, Hoefler T, Mueller F (2012) Assessing HPC failure detectors for MPI jobs. In: Proceedings of the 2012 20th euromicro international conference on parallel, distributed and network-based processing, pp 81–88
42. Kiciman E, Fox A (2005) Detecting application-level failures in component-based internet services. IEEE Trans Neural Netw 16(5):1027–1041
43. Lan Z, Gu J, Zheng Z, Thakur R, Coghlan S (2010) Dynamic meta-learning for failure prediction in large-scale systems: a case study. J Parallel Distrib Comput 6:630–643
44. Lan Z, Zheng Z, Li Y (2010) Toward automated anomaly identification in large-scale systems. IEEE Trans Parallel Distrib Syst 21:147–187
45. Leangsuksun C, Ostrouchov G, Scott SL (2008) Using log information to perform statistical analysis on failures encountered by large-scale HPC deployment. In: Proceedings of the 2008 high availability and performance computing workshop
46. Lehmann EL, Casella G (1998) Theory of point estimation, vol 31. Springer, New York
47. Li Y, Lan Z (2006) Exploit failure prediction for adaptive fault-tolerance in cluster computing. In: Sixth IEEE international symposium on cluster computing and the grid, CCGRID 06, vol 1
48. Liang Y (2006) Blue Gene/L failure analysis and prediction models. In: Proceedings of the international conference on dependable systems and networks, pp 425–434
49. Lou J (2010) Mining dependency in distributed systems through unstructured logs analysis. ACM Spec Interes Group Oper Syst (SIGOPS) 44
50. Lu C-D (2013) Failure data analysis of HPC systems. Technical report CoRR abs/1302.4779
51. Lu C-D, Reed DA (2005) Scalable diskless checkpointing for large parallel systems. Technical report, Ph.D. dissertation, University of Illinois at Urbana-Champain
52. Mane SV (2008) False negative estimation: theory, techniques and applications. ProQuest, Ann Arbor

53. Martino CD, Baccanico F, Fullop J, Kramer W, Kalbarczyk Z, Iyer RK (2014) Lessons learned from the analysis of system failures at petascale: the case of Blue Waters. In: IEEE/IFIP international conference on dependable systems and networks (DSN 2014)
54. Moody A, Bronevetsky G, Mohror K, de Supinski BR (2010) Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis, pp 1–11
55. Murray J, Hughes G, Kreutz-Delgado K (2003) Hard drive failure prediction using non-parametric statistical methods. In: Proceedings of ICANN/ICONIP
56. Nassar FA, Andrews DM (1985) A methodology for analysis of failure prediction data. In: IEEE real-time systems symposium, pp 160–166
57. Oliner A, Stearley J (2007) What supercomputers say: a study of five system logs. In: IEEE international conference on dependable systems and networks
58. Panigrahi PK, Dwivedi M, Khandelwal V, Sen M (2003) Prediction of turbulence statistics behind a square cylinder using neural networks and fuzzy logic. J Fluids Eng 125:385–387
59. Papadogiannakis A, Polychronakis M, Markatos EP (2010) Improving the accuracy of network intrusion detection systems under load using selective packet discarding. In: Proceedings of the third European workshop on system security, EUROSEC'10. ACM, New York, pp 15–21
60. Patra A, Bidhar S, Kumar U (2010) Failure prediction of rail considering rolling contact fatigue. Int J Reliab Qual Saf Eng 17(3):167–177
61. Rani S, Leangsuksun C, Tikotekar A, Rampure V, Scott S (2006) Toward efficient failure detection and recovery in HPC. In: Proceedings of high availability and performance workshop
62. Ricoux P (2013) European exascale software initiative EESI2—towards exascale roadmap implementation. In: 2nd IS-ENES workshop on high-performance computing for climate models
63. Ruping S (2000) MySVM manual. Technical report, University of Dortmund, CS Department, AI Unit
64. Sahoo RK, Oliner AJ, Rish I, Gupta M, Moreira JE, Ma S, Vilalta R, Sivasubramaniam A (2003) Critical event prediction for proactive management in large-scale computer clusters. In: Proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining, KDD'03. ACM, New York, pp 426–435
65. Salfner F (2006) Modeling event-driven time series with generalized hidden semi-Markov models. Technical report 208, Department of Computer Science, Humboldt University
66. Salfner F, Malek M (2007) Using hidden semi-Markov models for effective online failure prediction. In: Symposium on reliable distributed systems, pp 161–174
67. Salfner F, Lenk M, Malek M (2010) A survey of online failure prediction methods. Comput Surv 42:1–42
68. Schroeder B, Gibson G (2010) A large-scale study of failures in high-performance computing systems. IEEE Trans Dependable Secur Comput 7(4):337–350
69. Schroeder B, Gibson GA (2007) Understanding failures in petascale computers. J Phys: Conf Ser 78:012022
70. Shantharam M, Srinivasmurthy S, Raghavan P (2012) Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In: Proceedings of the 26th ACM international conference on supercomputing, ICS'12. ACM, New York, pp 69–78
71. Snir M, Wisniewski RW, Abraham JA, Adve SV, Bagchi S, Balaji P, Belak J, Bose P, Cappello F, Carlson B, Chien AA et al (2013) Addressing failures in exascale computing. Argonne report ANL/MCS-TM-332
72. Stearley J (2005) Defining and measuring supercomputer reliability, availability and serviceability (RAS). In: Proceedings of the Linux cluster institute conference
73. Stearley J, Oliner AJ (2008) Bad words: finding faults in spirit's syslogs. In: The eighth IEEE international symposium on cluster computing and the grid, pp 765–770
74. Taerat N, Naksinehaboon N, Chandler C, Elliott J, Leangsuksun C, Ostrouchov G, Scott S, Engelmann C (2009) Blue Gene/L log analysis and time to interrupt estimation. In: International conference on availability, reliability and security, ARES'09, pp 173–180

75. Thanakornworakij T, Nassar R, Leangsuksun CB, Paun M (2013) Reliability model of a system of k nodes with simultaneous failures for high-performance computing applications. Int J High Perform Comput Appl 27(4):474–482
76. Tiwari D, Gupta S, Vazhkudai S (2014) Lazy checkpointing: exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In: 2014 44th annual IEEE/IFIP international conference on dependable systems and networks (DSN), pp 25–36
77. Tsai T, Theera-Ampornpunt N, Bagchi S (2012) A study of soft error consequences in hard disk drives In: IEEE international conference on dependable systems and networks, pp 1–8
78. US Department of Energy (2012) Fault Management Workshop. http://shadow.dyndns.info/publications/geist12department.pdf. Accessed July 2013
79. Vilalta R, Apte C, Hellerstein J, Ma S, Weiss S (2002) Predictive algorithms in the management of computer systems. IBM Syst J 41:461–474
80. Wang C, Talwar V, Schwan K, Ranganathan P (2010) Online detection of utility cloud anomalies using metric distributions. NOMS. IEEE, pp 96–103
81. Workshop, I-A (2012) HPC resilience at extreme scale. http://institute.lanl.gov/resilience/docs/Inter-AgencyResilienceReport.pdf. Accessed July 2013
82. Xu W, Huang L, Fox A, Patterson D, Jordan M (2009) Online system problem detection by mining patterns of console logs. In: Proceedings of the 2009 ninth IEEE international conference on data mining, ICDM'09. IEEE Computer Society, Washington, pp 588–597
83. Yamanishi K (2005) Dynamic syslog mining for network failure monitoring. In: Proceedings of the 11th ACM SIGKDD, international conference on knowledge discovery and data mining. ACM Press, pp 499–508
84. Yigitbasi N, Gallet M, Kondo D, Iosup A, Epema D (2010) Analysis and modeling of time-correlated failures in large-scale distributed systems. In: 2010 11th IEEE/ACM international conference on grid computing (GRID), pp 65–72
85. Yu L, Zheng Z, Lan Z, Coghlan S (2011) Practical online failure prediction for Blue Gene/P: period-based versus event-driven. In: IEEE conference on dependable systems and networks workshops, pp 259–264
86. Zheng G, Shi L, Kale L (2004) FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In: 2004 IEEE international conference on cluster computing, pp 93–103
87. Zheng Z, Yu L (2011) Co-analysis of RAS log and job log on Blue Gene/p. In: Proceedings of the 2011 IEEE international parallel and distributed processing symposium, pp 840–851
88. Zheng Z, Li Y, Lan Z (2007) Anomaly localization in large-scale clusters. In: IEEE international conference on cluster computing, pp 322–330
89. Zheng Z, Lan Z, Gupta R, Coghlan S, Beckman P (2010) A practical failure prediction with location and lead time for Blue Gene/P. In: IEEE conference on dependable systems and networks workshops, pp 15–22