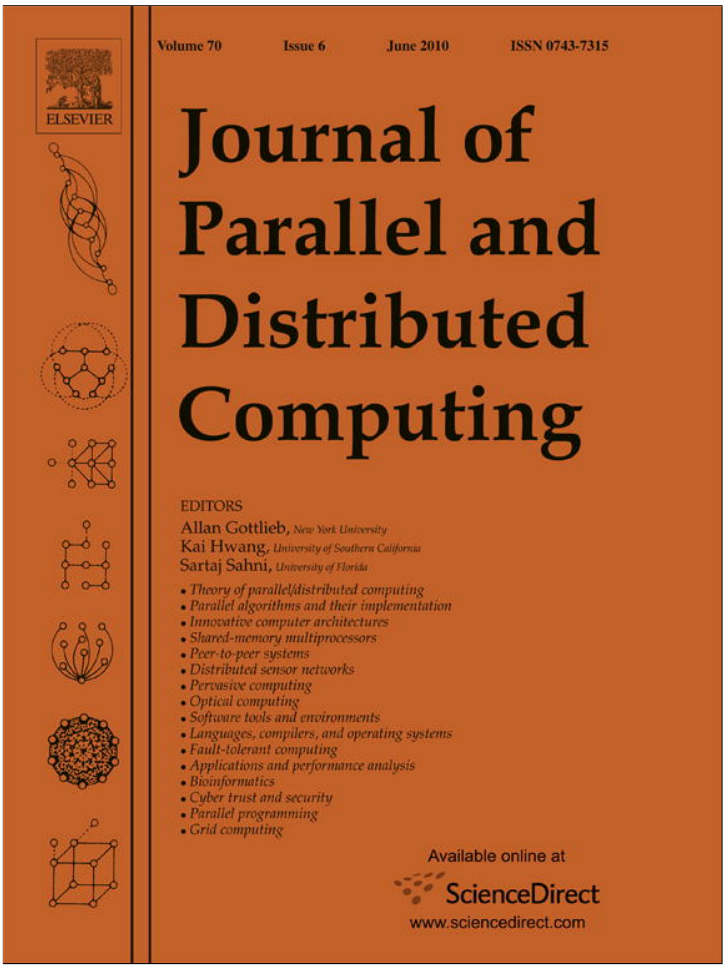


J. Parallel Distrib. Comput. 70 (2010) 630–643

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A study of dynamic meta-learning for failure prediction in large-scale systems

Zhiling Lan [a,*], Jiexing Gu [a], Ziming Zheng [a], Rajeev Thakur [b], Susan Coghlan [b]

[a] Illinois Institute of Technology, Chicago, IL 60616, United States
[b] Argonne National Laboratory, Argonne, IL, 60439, United States

ARTICLE INFO

Article history:
Received 19 December 2008
Received in revised form
10 September 2009
Accepted 4 March 2010
Available online 12 March 2010

Keywords:
Failure prediction
Meta-learning
Dynamic techniques
Large-scale systems
Blue Gene

ABSTRACT

Despite years of study on failure prediction, it remains an open problem, especially in large-scale systems composed of vast amount of components. In this paper, we present a dynamic meta-learning framework for failure prediction. It intends to not only provide reasonable prediction accuracy, but also be of practical use in realistic environments. Two key techniques are developed to address technical challenges of failure prediction. One is *meta-learning* to boost prediction accuracy by combining the benefits of multiple predictive techniques. The other is *a dynamic approach* to dynamically obtain failure patterns from a changing training set and to dynamically extract effective rules by actively monitoring prediction accuracy at runtime. We demonstrate the effectiveness and practical use of this framework by means of real system logs collected from the production Blue Gene/L systems at Argonne National Laboratory and San Diego Supercomputer Center. Our case studies indicate that the proposed mechanism can provide reasonable prediction accuracy by forecasting up to 82% of the failures, with a runtime overhead less than 1.0 min.

1. Introduction

1.1. Motivations

To meet the insatiable demand in science and engineering, supercomputers continue to grow in size. Production systems with tens to hundreds of thousands of computing nodes are being designed and deployed [36]. Such a scale, combined with the ever-growing system complexity, is introducing a key challenge—reliability—in the field of high performance computing (HPC). Despite great efforts on the design of ultra-reliable components, the increase of system size and complexity has outpaced the improvement of component reliability. Recent studies have pointed out that the mean time between failure (MTBF) of teraflop and soon-to-be-deployed petaflop machines are only on the order of 10–100 h [31,26,29].

To address the reliability problem, considerable research has been done to improve fault resilience of computer systems and their applications through various technologies. Representative works include failure-aware resource management and scheduling [42,23,25], checkpointing [9,24,32,16], proactive or adaptive runtime resilience support [20,40]. The advance of these technologies, however, greatly depends on whether we can predict the occurrence of failure, i.e., failure prediction. For example, proactive fault tolerant methods, such as preemptive process migration, require failure forecasting to enable cost-effective failure avoidance. For reactive methods such as checkpointing, an efficient failure prediction could substantially reduce their operational cost by telling when and where to perform checkpoints, rather than blindly invoking actions periodically with an unwisely chosen frequency.

Despite years of study on failure prediction, it remains an unsolved problem, especially in large-scale systems composed of substantial amount of components. We summarize its key challenges from two aspects. First is *prediction accuracy*. Existing studies mainly concentrate on exploring one specific method to capture and discover failure patterns. As a matter of fact, in a large-scale system the sources of failures are numerous and complex; thus, it is improper to expect a single method to capture all of failures alone. For example, many rule-based classifiers emphasize on discovering correlation relationships between warning messages and fatal events for failure prediction [19,30]. As we will show in our experiments, they are limited by the amount of fatal events occurring without any precursor warnings. Hence, relying on these methods alone is insufficient to provide an effective failure forecasting. Further, hardware and software upgrades are common at supercomputing centers, and system workloads tend to vary during system operation. These changes can drastically alter system behaviors [26]. As a result, static analysis that uses a fixed set of historic data to learn failure patterns cannot adapt to system changes at runtime, thereby being incapable of providing accurate forecasting.

* Corresponding author.
E-mail addresses: lan@iit.edu (Z. Lan), jgu5@iit.edu (J. Gu), zzheng11@iit.edu (Z. Zheng), thakur@mcs.anl.gov (R. Thakur), smc@mcs.anl.gov (S. Coghlan).

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.03.003

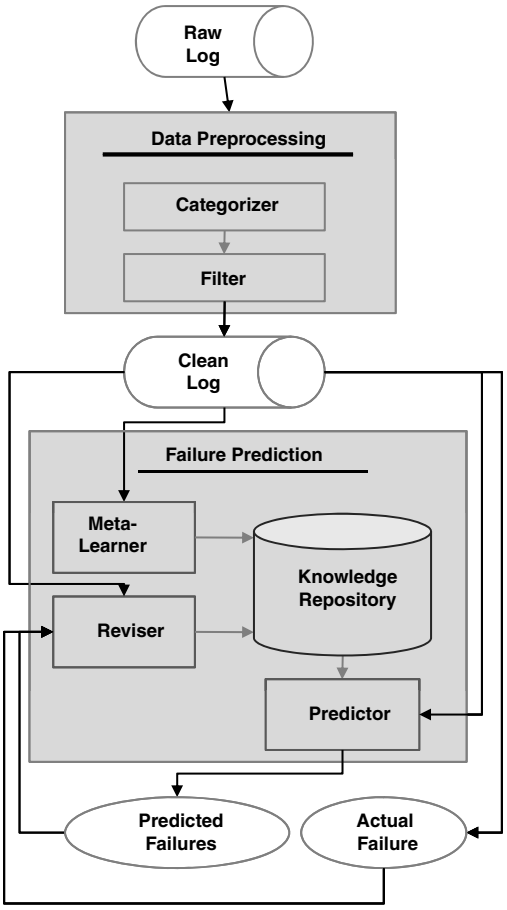


Fig. 1. Overview of our dynamic meta-learning framework for failure prediction.

The next is with respect to *practical use*. While many predictive models have been presented so far, most of them merely focus on the algorithm-level improvement and are too complicated to be of practical use for online failure prediction [22,41]. In addition, to obtain sufficient failure patterns, many predictive methods require a long training phase (e.g., one or more years), thereby being unable to provide prediction service for a long period of time [11]. Given that most systems at supercomputing centers only have a couple of years in production, this requirement must be removed.

1.2. Main contributions

In this study, we present a dynamic meta-learning framework for online failure prediction in large-scale systems. It intends to provide reasonable prediction accuracy, as well as be of practical use in realistic environments. Our framework consists of two parts to process and analyze system events: one is to preprocess system events by means of event categorization and filtering (i.e., *data preprocessing*), and the other is to examine the cleaned events for generating failure patterns and triggering failure warnings through continuous runtime event analysis (i.e., *failure prediction*). These two parts, along with their main components, are illustrated in Fig. 1. The details of the main components will be described in Sections 3 and 4.

Our method employs two key techniques to address the challenges listed above. First, *meta-learning* is explored to boost prediction accuracy by combining the benefits of multiple predictive methods. It enables us to discover a variety of failure patterns in large-scale systems, without constructing complex models of the underlying system. In this study, we integrate three widely used predictive methods, namely association rule-based learner [30,39], statistical rule-based learner [21], and probability distribution [31], in the framework by applying a simple yet efficient ensemble learning method. Next, *a dynamic mechanism* is adopted to trigger relearning periodically and to adaptively extract effective rules of failure patterns by actively tracing prediction accuracy.

To demonstrate the effectiveness and practical use of our framework, we evaluate it with the real RAS (reliability, availability and serviceability) logs collected from the production Blue Gene/L systems at Argonne National Laboratory (ANL) and San Diego Supercomputer Center (SDSC). The use of multiple RAS logs is to ensure that our framework is not bias to any specific log and thus produces representative results expected in other systems as well. To comprehensively assess our framework, our experiments are structured to answer the following questions:

Q1: How much improvement is achieved by the meta-learning?
Q2: How much improvement is achieved by the dynamic approach?
Q3: How sensitive is prediction accuracy to prediction window size?
Q4: How much runtime overhead is introduced?

Our experiments demonstrate that meta-learning can effectively improve prediction accuracy by up to three times, and the dynamic approach is capable of adapting to system changes, even after a major system reconfiguration. For both systems, our method can provide reasonable prediction accuracy by predicting up to 82% of failures, with a runtime overhead less than 1.0 min. Furthermore, prediction accuracy depends on how far away we are interested in forecasting failures. In general, the larger the window is, the higher the prediction coverage is, along with a higher false alarm rate. The rules of failure patterns change dramatically during system operation, which further proves that the dynamic approach is indispensable for better prediction. Finally, runtime overhead increases with the growing size of the training set. Overall speaking, we find that for both systems, the use of recent 6 month training set can well balance between prediction accuracy and runtime overhead.

We note that three predictive methods, namely association rule-based learning, statistical learning, and probability distribution, have been tested in our experiments. Rather than focusing on which predictive method is better, this study focuses on providing a general framework to dynamically combine multiple predictive methods for better failure prediction. We believe that other predictive methods like [22,6] can be easily integrated into our framework.

1.3. Paper organization

The rest of the paper is organized as follows. Section 2 gives the background information of Blue Gene systems and system logs. Section 3 describes the details of data preprocessing, followed by a detailed description of our dynamic meta-learning method in Section 4. The case studies with real failure logs are presented in Section 5. Section 6 discusses the related work and points out the key differences between this work and existing studies. Finally, Section 7 summarizes the paper.

2. Background

2.1. Overview of Blue Gene/L

In this paper, we use RAS (reliability, availability and serviceability) logs collected from the Blue Gene/L systems for case studies; thus, we give an overview of the systems and their RAS logging facilities below. The proposed dynamic meta-learning framework

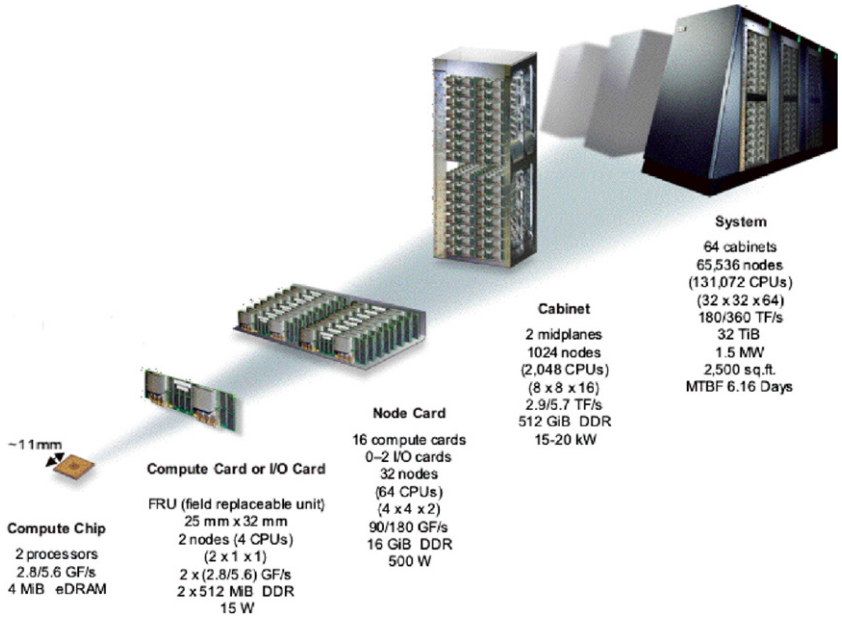


Fig. 2. Blue Gene/L system overview.

Table 1
Attributes of RAS events in Blue Gene.

Attribute	Description
Record ID	An integer denoting event sequence number
Event type	The mechanism through which the event is recorded
Event time	Timestamp associated with the reported event
Job ID	Job that detects the event
Location	Place of the event (e.g., chip/node card/service card/link card)
Entry data	A short description of the event
Facility	The service/hardware component experiencing the event
Severity	The level of severity of the reported event

can be easily extended for failure prediction of other large-scale systems.

System packaging is an integral aspect of Blue Gene/L systems (see Fig. 2). As shown in the figure, the basic building block is called computer chip. Each computer chip consists of two PPC 440 cores, with a 32 KB L1 cache and a 2 KB L2 cache. The cores share a 4 MB EDRAM L3 cache. A compute card contains two computer chips, a node card contains 16 compute cards, and a midplane holds 16 node cards with a total of 1024 processors. In addition to compute nodes, a midplane is also populated with several I/O nodes which are configured to handle file I/O and host communication. Each midplane also has one service card that performs system management services like monitoring node heartbeat and checking errors. More details of the system architecture can be found in the literature [7].

In Blue Gene, the Cluster Monitoring and Control System (CMCS) service is implemented on the service nodes for the purpose of system monitoring and error checking. The service node, which is available in each midplane, acquires specific device information, such as RAS events, directly through the control network. Runtime information is collected from computer and I/O nodes by a polling agent, reported to the CMCS service, and finally stored in a centralized DB2 repository. This system event logging mechanism works in a granularity of less than 1 ms.

The entries in the RAS log include hard errors, soft errors, machine checks, and software problems. Information about scheduled maintenance, reboot, and repair is not included. Each record has eight attributes which are described in Table 1.

The SEVERITY attribute can be one of the following levels—INFO, WARNING, SEVERE, ERROR, FATAL, or FAILURE—which also denotes the increasing order of severity. INFO events are for the purpose of general information to administrators about the reliability of various hardware/services components in the system. WARNING events report unusual events in node cards, link cards, service cards or related services. SEVERE events provide more information about the reasons causing problems in node cards or service cards, etc. ERROR events indicate problems that require further attention of administrators. An event with any of the above SEVERITY attributes is either informative in nature, or is related more to the initial configuration errors, and is thus relatively transparent to the applications/runtime environment. However, FATAL or FAILURE events (such as "uncorrectable torus error", "communication failure socket closed", "uncorrectable error detected in eDRAM bank", etc.) are more severe, and usually lead to system/application crashes. Our primary focus in this study is to predict FATAL and FAILURE events (denoted as *fatal events*,[1] while other events are denoted as *non-fatal events*). In [26], Oliner et al. have pointed out that some of the fatal events provided by the RAS log are not true fatal events. We have consulted with experienced system administrators at both ANL and SDSC, and removed these "fake" fatal events from the failure list.

2.2. Test logs

Two production Blue Gene/L systems are used in our experiments. One is at SDSC, which consists of three racks with 3072 dual-core compute nodes and 384 I/O nodes. The configuration is chosen to support data-intensive computing. Each node consists of two PowerPC processors that run at 700 MHz and share 512 MB of memory, giving an aggregate peak speed of 17.2 teraflops and a total memory of 1.5 TB [35]. The other is at ANL, which has one rack with 1024 dual-core compute nodes and 32 I/O nodes [34]. The aggregate peak performance is of 5.7 teraflops, with a total memory of 500 gigabytes. Both systems are mainly used for scientific computing. Table 2 summarizes the RAS logs used in our experiments.

[1] In the paper we use "failure" and "fatal event" interchangeably.

Table 2
Log description.

Log	Period	Weeks	Event no.	Log size
ANL BGL	Jan. 21, 2005–Jun. 19,2007	112	5 887 771	2.27 GB
SDSC BGL	Dec. 6, 2004–Jun. 11, 2007	132	517 247	463 MB

The log from ANL has much more number of records, although the system has only one rack of nodes. This is due to a large quantity of error checking messages produced at the ANL site. For example, during the 50th week of the ANL's log (between January 6 and January 13, 2006), there were over 1.15 million of machine checking information messages generated. System administrators at ANL ran diagnostics more frequently to cull out bad hardware faster, without applications seeing it.

3. Data preprocessing

Raw logs generally contain many repeated or redundant information. This is because each computer chip runs a polling agent to collect the errors reported by the chip. As each job is assigned to multiple computer chips, any failure of the job will get reported at multiple places—once from each of the assigned computer chips. Thus multiple components may report the same failure. Also, the logging mechanism records the events at a very fine granularity (e.g., in millisecond), but the recorded event time is generally in seconds or minutes, thus leading to multiple entries of an event with the same time stamp. Therefore, before a RAS log can be used for failure prediction, it needs to be processed to identify unique RAS events.

As shown in Fig. 1, data preprocessing mainly consists of two components: one is *event categorizer* and the other is *event filter*. The categorizer aims at providing a precise list of RAS types, and the filter removes redundant data by conducting temporal compression at a single location and spatial compression across multiple locations. The goal of data preprocessing is to provide a list of unique events for failure prediction.

3.1. Categorizer

Event categorization is a time-consuming process. It requires a deep understanding of system events; thus, close collaboration with system administrators is essential for obtaining a list of meaningful event categories. Fortunately, for a specific system, the process only needs to be performed once. Once a standard categorizing of system events is constructed, we can use it for a long period of time, unless a drastic change occurs in the system (e.g., system reconfiguration). In the case of minor changes during system operation, existing categorization technologies such as the one presented in [27] can be applied for dynamic tuning of event classifications.

We adopt *a hierarchical approach* for event categorization. We first divide system events into several high-level classifications, and then further group events into a number of subcategories based on their attributes. For the Blue Gene/L systems, 10 high-level event categories are identified based on the *Facility* field, which are further divided into 219 low-level event types based on the *Severity* and *Entry Data* fields. Further, it is also necessary to distinguish these event categories into fatal or non-fatal groups for the purpose of data training. Non-fatal events indicate system warnings or information messages, while fatal events refer to those critical events that lead to system or application crashes. Although RAS logs from Blue Gene/L provide a severity level for each event, it is not accurate since some fatal or failure events are not truly fatal at all [26]. By working with system administrators, we have identified and removed some of these events from the fatal list. Totally, there are 69 fatal events for the Blue Gene/L systems. Examples are shown in Table 3.

3.2. Filter

Event filtering is required to remove duplicated or unnecessary entries in the log. Common cleaning steps include removing duplicated entries, removing unnecessary entry attributes, correcting inaccurate attributes, preparing output files for corresponding learning methods, etc. In this study, we apply both temporal compression and spatial compression to remove duplicate entries by applying threshold-based techniques. With temporal compression at a single location, events from the same location with identical values in the *Job ID* and *Location* fields are coalesced into a single entry, if reported within a predefined time duration. With spatial compression across multiple locations, we remove those entries that are close to each other within a predefined time duration, with the same *Entry Date* and *Job ID*, but from different locations.

How to decide an optimal threshold for filtering is still an open question. In this study, we adopt an *iterative approach* [15,2]. We first set the threshold to a very small number, and then gradually increase the number. The search stops when there is no significant change with respect to the compression rate. Table 4 presents the

Table 3
Event categories in Blue Gene/L.

Main category	Examples	No. of fatal categories	No. of non-fatal categories
APP	Load Program failure Function call failure	10	7
BGLMASTER	Segmentation failure BGLMaster restart info	2	2
CMCS	CMCS command info CMCS exit info	0	4
DISCOVERY	Nodecard communication warning Servicecard read error	0	24
HARDWARE	Midplane service warning	1	12
KERNEL	Broadcast failure Cache failure CPU failure Node map file error	46	90
LINKCARD	Linkcard failure	1	0
MMCS	Control network MMCS error	0	5
MONITOR	Node card temperature error	9	5
SERV_NET	System operation error	0	1
TOTAL		69	150

Table 4
Number of events with different filtering thresholds (in seconds).

	Log	0 s	10 s	60 s	120 s	200 s	300 s	400 s
APP	ANL	6758	1942	1827	1684	1566	1453	1378
	SDSC	26358	754	741	675	615	579	556
BGLMASTE	ANL	123	123	120	115	115	109	107
	SDSC	119	119	114	105	99	93	90
CMCS	ANL	302	295	292	286	284	283	280
	SDSC	437	433	421	404	384	362	356
DISCOVERY	ANL	18054	1727	1429	937	676	578	497
	SDSC	60748	3621	3356	1352	750	565	556
HARDWARE	ANL	1840	668	633	601	593	539	468
	SDSC	1648	422	349	316	296	283	278
KERNEL	ANL	5819166	59784	47998	40777	33847	26754	23823
	SDSC	426816	4238	4056	3940	3747	3595	3379
LINKCARD	ANL	64	30	18	15	13	11	10
	SDSC	188	120	107	95	92	88	82
MMCS	ANL	954	561	521	484	467	444	437
	SDSC	929	654	630	590	563	523	501
MONITOR	ANL	40509	19774	16120	15969	15834	15689	15421
	SDSC	0	0	0	0	0	0	0
SERV_NET	ANL	1	1	1	1	1	1	1
	SDSC	4	4	4	4	4	4	4

numbers of events after applying different thresholds, where we separate the numbers according to the high-level event categories. The column where threshold is set to zero denotes the raw logs before any compression. For both logs, the amount of compression of events achieved is not significant when the threshold greater than 300 s is used. Additionally, as RAS events are logged at a sub-second frequency, taking a higher threshold value will increase the chances of different events being clustered together. Hence, we choose 300 s as the threshold to coalesce events, which achieves above 98% compression rate for the logs.

4. Prediction methodology

Our prediction method consists of three major components: the *meta-learner*, the *reviser*, and the *predictor* (see Fig. 1). The *meta-learner* examines system events to discover various fault patterns by applying multiple predictive methods. The generated failure patterns or rules will be stored in a knowledge repository which encompasses all of the relevant information of failure patterns. It contains all the learned rules of failure patterns and corresponding ensemble rules for meta-learning. These rules of failure patterns are subjected to modifications made by the reviser at runtime. The *reviser* monitors prediction accuracy by comparing the predicted results and the actual failures, and then modifies the knowledge base. The training set used by the meta-learner and the reviser is periodically changed during system operation. The *predictor* actively examines system events. In case that the occurrence of an event triggers a matching pattern in the knowledge base, it will trigger a warning.

Distinguishing from existing studies like [30,21,22,13,6], our framework has two novel features. One is to exploit meta-learning (i.e., ensemble learning) to boost failure prediction and the other is to dynamically learn failure patterns from a changing training set during system operation.

Before we go to the details of these components, we first present the main terms used in our framework (see Fig. 3). The training set, which may be dynamically adjusted every W_R weeks (denoted as *retraining window*), is part of the log from which the meta-learner and the reviser use to generate the rules of failure patterns. In other words, the meta-learner and the reviser will be invoked every W_R weeks. The rules are generated with a fixed time window, generally in the order of a couple of minutes to hours (denoted as *rule generation window* W_P). The rules learned will be stored in the knowledge repository, which will be used by the predictor for failure prediction before the next retraining. The predictor actively

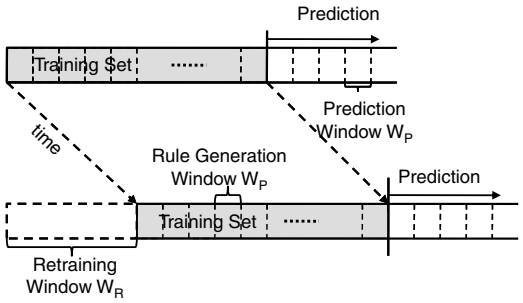


Fig. 3. Key terms in failure prediction.

monitors the events occurring during *prediction window*, whose size is the same as the rule generation window W_P, and in the case of a matching rule, it will trigger a warning.

4.1. Meta-learner

The meta-learner focuses on revealing and learning the cause-and-effect relations of system events by applying data mining techniques. Data mining, or knowledge discovery, is a computer-assisted process of searching and analyzing data sets for hidden patterns [14]. Meta-learning, also known as ensemble-learning, can be loosely defined as learning from learned knowledge [28]. It emphasizes on combining different individual models (denoted as *base learners*) to boost overall predictive effectiveness.

In this study, we choose three widely used predictive methods, namely association rule-based method [39,30], statistical rule-based method [21], and probability distribution-based method [31], as our base learners. The meta-learner intends to identify a preferable combination of these base learners. In the following, we first describe the base learners, followed by presenting our meta-learning method. Note that other base methods can be easily incorporated.

Base learners. The first base learner is based on *association rules*. It examines *causal correlations* between *non-fatal* and *fatal* events by building association rules. In general, an association rule is in the form $X \rightarrow Y$, where the rule body X and Y are subsets of an event set. It states that a transaction that contains the items in X are likely to contain the items in Y. Association rules are characterized by two measures: *support* which measures the percentage of transactions that contain both items X and Y, and *confidence* which measures the percentage of item sets containing

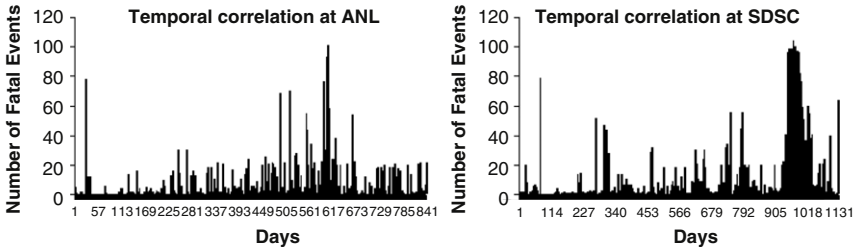


Fig. 4. Temporal correlations among fatal events.

the items X that also contain the items Y. The problem of mining association rules consists of generating all the association rules from a set of items that have both *support* and *confidence* greater than the user-defined thresholds. Given that failure is rare event, low values of *support* and *confidence* are set for the purpose of capturing infrequent events.

On the training set, for each fatal event, we identify the set of non-fatal events preceding it within the rule generation window W_P. The set, including the fatal event and their precursor non-fatal events, is called an event set. We then apply the standard association rule algorithm to build rule models for event sets that are above the minimum *support* and *confidence*. The association rules will be in the form of $\{e_1, e_2, \ldots, e_k\} \rightarrow f$, *confidence*, where e_i and e_j $(1 \leq i, j \leq k)$ are non-fatal events, and f is a fatal event. For instance, two examples from the SDSC log are listed below:

networkWarningInterrupt, networkError → *socketReadFailure:* 1.0
idoStartInfo, bglStartInfo → *fsFailure:* 0.79

Our second base learner emphasizes on discovering *statistical characteristics*, i.e., how often and with what probability will the occurrence of one failure influence subsequent failures, among fatal events and then using the obtained statistical rules for failure prediction. It is denoted as the *statistical rule-based method*. Studies have shown that temporal correlations among fatal events are common in large-scale systems [31,21,6]. Fig. 4 plots fatal events per day occurred at ANL and SDSC. We can observe that a significant number of failures happen in close proximity, and our further analysis indicates that network and I/O stream related failures form a majority of such failures.

Specifically, on the training set, we calculate the probability of k failures occurred within the rule generation window W_P. If the probability is larger than a user-defined threshold, then a statistic rule is generated, along with its probability value. As an example, we have discovered that for both logs, if four failures occur within 300 s, then the probability of another failure is 99%.

The third base learner is called the *probability distribution-based method*. It generates *probability distribution* of fatal events and stores it for failure prediction. Different from the above two methods which attempt to discover *short-term* (e.g., in the order of minutes) correlations among events for failure prediction, this method recognizes that some failure events may not have any short-term precursor events and intends to utilize *long-term* failure behavior for failure prediction. Here, the long-term means the probability distribution of failure events, which is generally in the order of hours or even days.

Specifically, the method calculates inter-arrival times between adjacent fatal events and uses maximum likelihood estimation to fit a mathematical model to these data. Distributions like Weibull, exponential, and log-normal are examined for generating the cumulative distribution function (CDF) of fatal events. Fig. 5 plots the CDFs of fatal events occurred at ANL and SDSC. By calculating the probability of possible failure based on the derived CDF function, this base method will trigger a warning if the probability is larger than a user-defined threshold, or equally saying, when the elapsed time since the last failure is longer than some threshold.

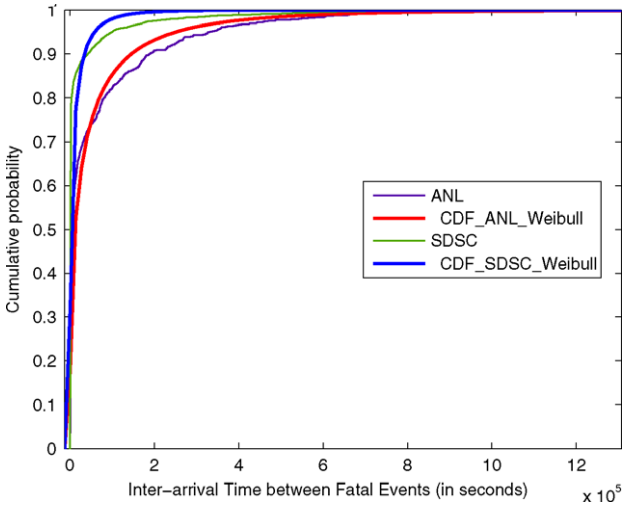


Fig. 5. Cumulative Distribution Functions (CDFs) of fatal events. The thin curves represent the actual fatal events, while the thick curves model the Weibull distributions of the events.

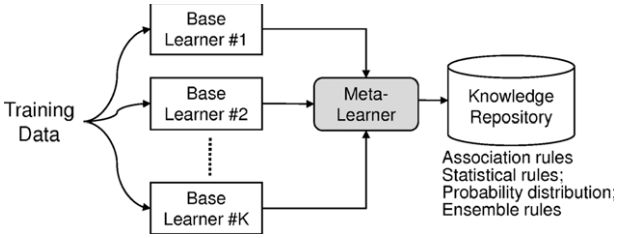


Fig. 6. Meta-learning method.

For instance, on a training set from the SDSC log, the Weibull distribution of $F(t) = 1 - e^{-(t/19984.8)^{0.507936}}$ is determined to be the best CDF to describe inter-arrival times between adjacent fatal events. Hence, if the threshold is set to 0.60, when the elapsed time since the last failure is 20 000 s, a warning will be triggered because $F(20\,000)(=0.63)$ is larger than the threshold.

Ensemble learning. There are many ways to combine base models, among which bagging, boosting, and stacking are well-known ensemble methods. In our study, we choose the *mixture-of-experts* model, which is a variation of the stacking method [11,28]. Fig. 6 illustrates our meta-learning process. The basic idea is simple: base learners are experts in some portion of the feature space, and the combination rule selects the most appropriate classifier for each instance. Based on verification on the training data, the meta-learner determines the ordering of rules used for prediction.

In our case studies, the order is the association rule, followed by the statistical rule, and finally probability distribution. Specifically, when an event occurs, if it is a non-fatal event, the meta-learner first checks whether it will trigger a matching of an association rule; if it is a fatal event, the meta-learner will check whether it will trigger a matching of a statistical rule. If no matching is found,

the meta-learner will check the elapsed time since the last failure and apply the derived probability distribution of failures for failure forecasting.

4.2. Reviser

The reviser is responsible for modifying the candidate rules generated by the meta-learner via monitoring the actual observations and the predicted results. This is to ensure the effectiveness of the learned rules in the knowledge repository. As mentioned earlier, in order to capture infrequent items, the parameters used in the base learners may be adopted without much consideration regarding their effectiveness, thereby probably resulting in some bad rules. Thus, the reviser checks each rule in the knowledge repository by applying the ROC (receiver operating characteristic) analysis [14]. It enables us to select optimal models and discard suboptimal ones independently from the class distribution. The reviser will examine each rule and only keep the rules which can provide satisfactory accuracy [12]. The detailed method is shown in Algorithm 1.

Algorithm 1 The Reviser

For each rule r generated by the meta-learner:

(1) count its true positives T_P, false positives F_P, and false negatives F_N on the training data;
(2) calculate two metrics $m_1(r)$ and $m_2(r)$:

$$m_1(r) = \frac{T_P}{T_P + F_P}, m_2(r) = \frac{T_P}{T_P + F_N};$$

(3) calculate $ROC(r)$:

$$ROC(r) = \sqrt{m_1(r)^2 + m_2(r)^2};$$

(4) keep the rule if its ROC value is larger than a predefined threshold *MinROC*; otherwise, discard the rule.

4.3. Predictor

The predictor actively monitors system events and triggers a warning when a rule is observed within the prediction window W_P. In order to be used for online forecasting, an *event-driven* method is adopted for its design [12]. That is, the predictor triggers a warning on the occurrence of events.

The detailed method is presented in Algorithm 2. The predictor maintains three event lists. One is called *F-list* which records a list of triggering events for each failure event. The second is called *E-list* which tracks a list of failure events that may be triggered by each event (fatal or non-fatal). The third is to keep the most recent events occurred within W_P. Upon an occurrence of an event e, the predictor appends the event into the third list (Step 1), and then goes through its *E-list* to find out all possible failures that may be triggered by its occurrence (Step 2). For each possible failure f^i, the predictor checks its *F-list* to see whether a cause-and-effect rule is matched in the knowledge repository (Steps 3 and 4).

5. Experiments

To evaluate the effectiveness of the proposed framework, we use the real RAS logs collected from the production systems at ANL and SDSC (see Table 2). Further, to comprehensively examine the framework, our experiments are structured to answer the key questions listed in Section 1.

5.1. Evaluation metrics

Two metrics are used to measure prediction accuracy:

Algorithm 2 The Predictor

First, it creates two lists based on the learned rules:

$F - List = \{f_i \Leftarrow \{e_{i1}, e_{i2}, \ldots, e_{ik}\} : 1 \le i \le N_f\}$

$E - List = \{e_j \Rightarrow \{f_{j1}, f_{j2}, \ldots, f_{jl}\} : 1 \le j \le N_e\}$

where f_i is a fatal event and e_i is a fatal or non-fatal event, N_f is number of fatal events, and N_e is number of any events. During operating, when an event e occurs:

(1) Append e into the monitoring event set $E = \{e_1, e_2, \ldots, e_n, e\}$ where the events are sorted in an increasing order of their occurrence times, remove e_i if its occurrence time is more than W_P before the occurrence time of e, i.e., keep the most recent events occurred within W_P.
(2) Go through the E-List of e, obtain the potential failures that may be triggered by e: $\{f^1, f^2, \ldots, f^k\}$.
(3) For each potential failure f^i, go through its F-List: $f^i \Leftarrow \{e^i_{i1}, e^i_{i2}, \ldots, e^i_{ik}\}$.
(4) If $\{e^i_{i1}, e^i_{i2}, \ldots, e^i_{ik}\} \subseteq E$, then produce a warning that the failure f^i may occur within the time of W_P.

(1) *Precision* is defined as the proportion of correct predictions to all the predictions made.

$$precision = \frac{T_p}{T_p + F_p}.$$

(2) *Recall* is defined as the proportion of correct predictions to the number of failures.

$$recall = \frac{T_p}{T_p + F_n}.$$

Here, T_p is the number of correct predictions (i.e., true positives), F_p is the number of false alarms (i.e., false positives), and F_n is the number of missed failures (i.e., false negatives). Obviously, a good prediction engine should achieve a high value (closer to 1.0) for both metrics. We note that these metrics are also used by the reviser to determine whether a rule is effective or not (see Algorithm 1).

5.2. Results

In our experiments, the training set is initially set to 6 months, which will be dynamically adjusted during operation. The default retraining window W_R is 4 weeks, and the default prediction window W_P, also the rule generation window, is 300 s.

The minimum *support* and *confidence* values for association rules are set to 0.01 and 0.1 respectively. The low values are chosen for the purpose of capturing infrequent events. The rules that are not good will be removed by the reviser. There are three other parameters used by our framework, namely *MinROC* for the reviser, and the thresholds for statistical rule-based learner and probability distribution-based learner. In our experiments, *MinROC* is set to 0.7, and the thresholds for statistical rules and probability distribution-based learner are set to 0.8 and 0.6 respectively. Choosing optimal values for these parameters is difficult, and often experimental determination might be the only viable option. We have tested different values, from a low value like 0.1 to a high value like 0.9, and found that these values can yield the best prediction accuracy for both logs. In general, low values for these parameters result in more failure rules and thus better failure coverage, at the expense of introducing more false alarms.

5.2.1. Q1: How much improvement is achieved by the meta-learning?

In this set of experiments, we compare prediction results by using static meta-learner as against individual base learners (i.e., association rule, statistical rule, and probability distribution).

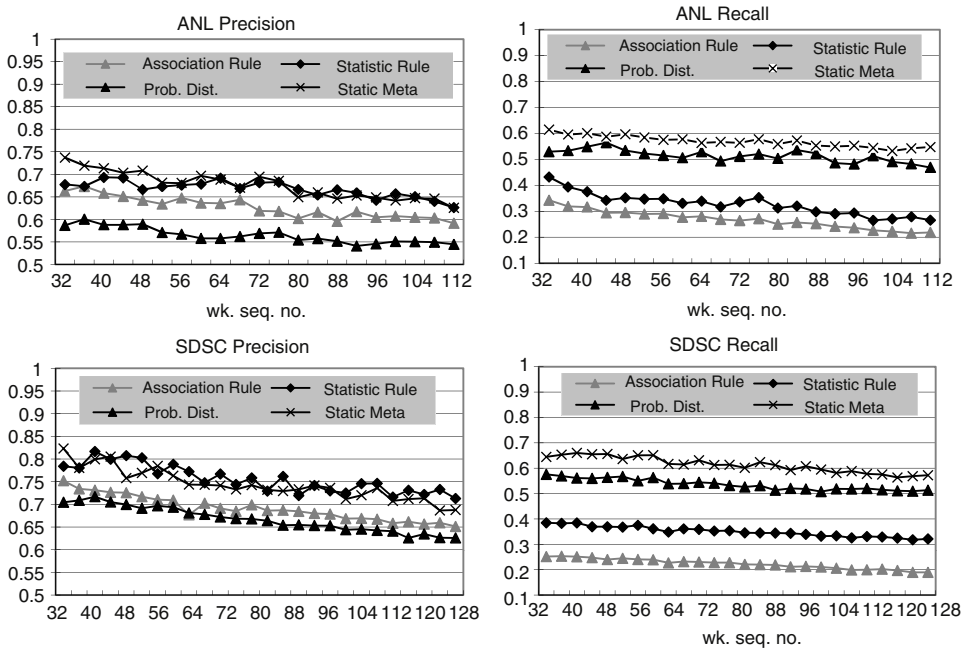


Fig. 7. Meta-learning versus base predictive methods. Each plot contains four curves, representing association rule-based learner, statistical rule-based learner, probability distribution-based learner, and static meta-learner. Here, the "static" means that the meta-learner applies mixture-of-experts ensemble of the base methods without dynamic relearning. It is clear that meta-learning can substantially boost prediction accuracy in terms of both *precision* and *recall*.

Here, the "static" means that the meta-learner simply applies mixture-of-experts ensemble of the base methods, without dynamic training, retraining and revising. Hence, for both logs, the first 6 months are used as the training set, and the remaining parts are for testing. The results are shown in Fig. 7, where the *x*-axis shows the sequence number of the week.

First, both *precision* and *recall* decrease as the time goes by, no matter which method is used. The reason is that all these methods are based on a static approach, meaning that they learn the rules on the 6 month training set and then use these rules for prediction on the rest of the logs. The rules may well capture failure patterns at the beginning. However, system behavior is dynamically changing. As a result, the established rules become outdated, thereby resulting in lower prediction accuracy as the time goes by.

We make several interesting observations regarding each base method. First, the statistical rule-based method provides a reasonably good result for *precision*; however, it results in a low value for *recall*. It suggests that this method is only good at discovering certain types of failures which exhibit temporal correlations. Second, the association rule-based method has the worse results in terms of *recall*. This is mainly due to the fact that while this method well captures causal correlation between non-fatal and fatal events, it is limited by the proportion of fatal events without any precursor warnings (e.g., low *recall* values). Our analysis shows that for both logs, there are a large portion of fatal events (up to 75%) which are not preceded by any precursor non-fatal events. Third, the *recall* results provided by the probability distribution-based method are quite good (e.g., higher than 0.5 for both logs). Nevertheless, it can introduce many false alarms. The problem of the probability distribution-based method is that it cannot pinpoint the occurrence times of the failures, thereby giving many false alarms once the elapsed time since the last failure is large enough.

A Venn diagram of these base learners is presented in Fig. 8. It shows the numbers of fatal events predicted by these base learners between the 44th and 48th week of the SDSC log. In total, there are 156 fatal events during this period, and 67 of them are

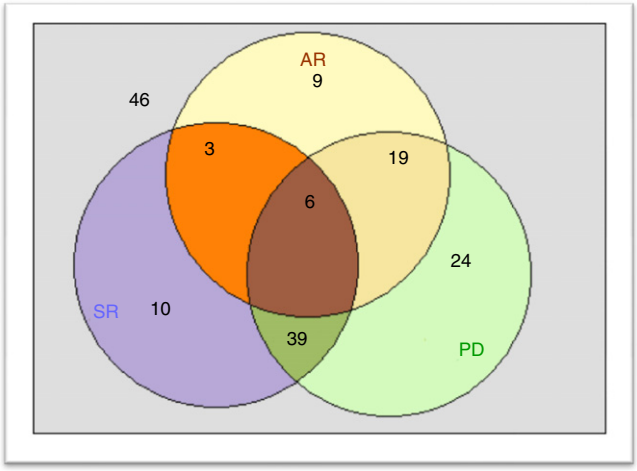


Fig. 8. A Venn diagram to show logical relations between association rule-based learner (AR), statistical rule-based learner (SR), and probability distribution-based learner (PD) between the 44th and 48th week of the SDSC log. Each number represents the number of fatal events captured by one or multiple base learners. There are totally 156 fatal events occurred during this period of time. For example, the number six in the intersection of three circles indicates that six fatal events can be predicted by all these base learners.

captured by multiple base learners. The coverage of each base learner is as follows: association rule-based learner 23.7% (37 fatal events), statistical rule-based learner 37.2% (58 fatal events), and probability distribution-based learner 56.4% (88 fatal events). The diagram clearly shows that it is improper to expect a single method to capture all of failures alone.

Observation #1: In a large-scale system, there are numerous failure patterns in general; thus, a single base learner is unlikely to capture all of them alone.

Meta-learning can substantially improve *recall*, indicating that meta-learning can improve prediction coverage by capturing various fault patterns. The impact of using meta-learning on *precision* is not as significant as on *recall*, but still non-trivial, especially as

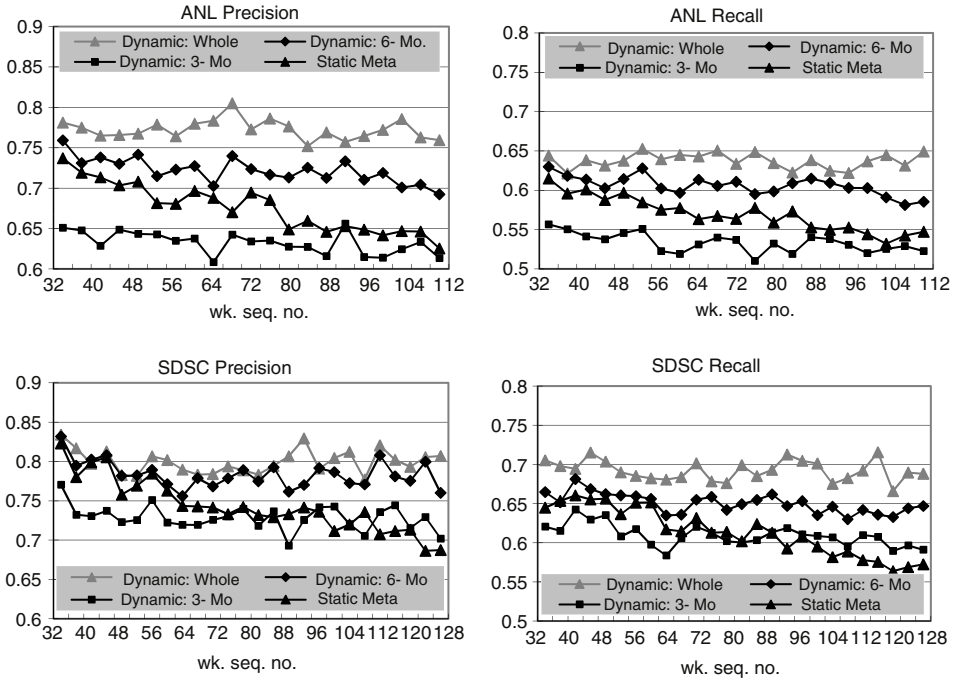


Fig. 9. What is the appropriate size for the training set? Each plot consists of four curves: (1) *dynamic-whole* means to train the rules using all the historical data; (2) *dynamic-6 mo* means to train the rules using the recent 6 months; (3) *dynamic-3 mo* means to train the rules using the recent 3 months; and (4) *static* means to use the first 6-month training set. Clearly, *dynamic-whole* and *dynamic-6 mo* are the best. Combining the results shown in Table 5, we suggest that dynamic training on the most recent 6 month provides the best balance between prediction accuracy and runtime overhead.

compared to the association rule-based method and probability distribution method. Note that the meta-learner does not modify any of these base methods; instead, it dynamically chooses one base learner for failure forecasting upon each invocation. The benefit of using meta-learner is its ability to form a good integration of these base learners so as to improve both *precision* and *recall*.

Observation #2: Meta-learning can substantially improve prediction accuracy by intelligently integrating multiple predictive methods without requiring complex system modeling.

5.2.2. Q2: How much improvement is achieved by the dynamic approach?

In this set of experiments, we assess the benefits brought by the dynamic approach. Specifically, we analyze what is the appropriate size for the training set, how often to trigger relearning, whether it is necessary to perform dynamic revising, and how many rules are changed by applying dynamic relearning.

Fig. 9 presents the answer to the first question, i.e., what is the appropriate size for the training set? In the figure, each plot consists of four curves: (1) *dynamic-whole* means to train the rules using all the historical data, e.g., in the 32nd week, the data in the previous 31 weeks are used for training; (2) *dynamic-6 mo* means to train the rules using the recent 6 months, e.g., in the 32nd week, the data in the previous 26 weeks are used for training; (3) *dynamic-3 mo* means to train the rules using the recent 3 months, e.g., in the 32nd week, the data in the previous 13 weeks are used for training; and (4) *static* means to use the initial 6-month data as the fixed training set. With the first method, as the time goes by, the training set is gradually increased every 4 weeks. With the second and third methods, the training set is sliding with the time every 4 weeks, with a fixed size of 6 months or 3 months respectively. With the fourth method, we always use the rules generated in the initial training set for failure prediction, i.e., without any retraining.

Clearly, *dynamic-whole* provides the best results in terms of both *precision* and *recall*, followed by *dynamic-6 mo*. Further, we can see that the accuracy difference between these two methods is generally less than 0.08. As we will show in Section 5.2.4, the overhead introduced by training on a large data set is not trivial. Therefore, in practice we suggest to make a tradeoff between better prediction accuracy and lower computation overhead. For these systems, we suggest to use the most recent 6-month data for dynamic training.

Next, it is shown that by using the *static method* without dynamically adjusting the training set, the prediction accuracy is monotonically decreasing. This is reasonable as the fixed rule set learned by static meta-learner is unable to adapt to new changes/reconfigurations occurring in the system.

Finally, the results produced by *dynamic-3 mo* are the worst among four different mechanisms. The reason is that this method relies on a limited amount of training data for rule generation and this could substantially limit its capability of discovering sufficient failure patterns for prediction. Nevertheless, as compared to *static*, the prediction results are more stable in the sense that they do not decrease dramatically with time. In summary, the plots indicate that dynamically adjusting the training set is needed; however it is not necessary to re-train the rules on the entire available data, generally the most recent few months are sufficient.

Observation #3: Learned rules of fault patterns may not be applicable for very long, thus dynamically adjusting the training set is indispensable for good prediction accuracy. In general, using the most recent few months like six months makes a good tradeoff between accuracy and runtime overhead.

Fig. 10 answers the second question, i.e., how often to trigger relearning. It presents the results by using different retraining windows (W_R = 2, 4, or 8 weeks). While prediction accuracy generally remains similar, more frequent retraining can provide better accuracy by up to 0.06. Further, we notice that for the SDSC log, both *precision* and *recall* decrease more than 10% during the 64th week. This is due to the fact that the system went through a major system reconfiguration around this time. As a

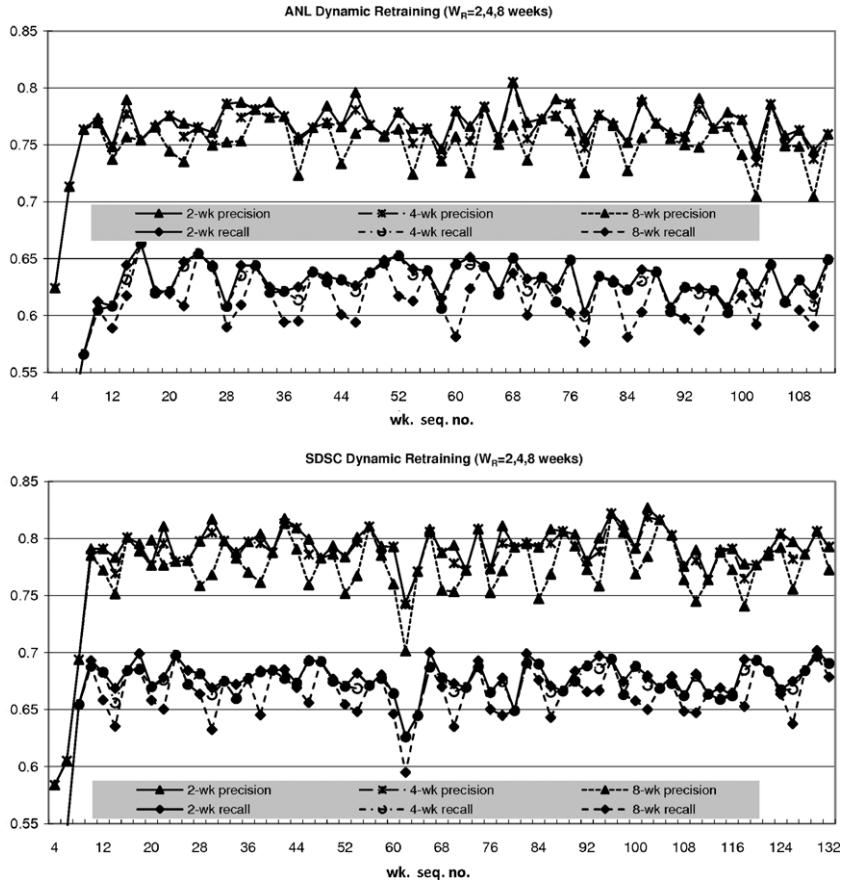


Fig. 10. How often to trigger relearning? The plots present the cases where the rules are re-trained every 2, 4, or 8 weeks, i.e., W_R is set to 2,4, or 8. Obviously, more frequent retraining can boost prediction accuracy; however, the improvement is not drastic with the difference less than 0.06 in terms of both *precision* and *recall*.

consequence, failure patterns were changed, thereby resulting in lower prediction accuracy during this period of time. Dynamic training is able to construct a new set of pattern rules. As we can see, both *precision* and *recall* are changed back to the normal range after a few retraining processes. Generally speaking, if the system is constantly changing or its workload is highly dynamic, frequent retraining is necessary, which can help to rapidly build up the effective rules for online prediction.

Observation #4: The frequency to trigger relearning depends on system characteristics. If the system is highly dynamic, frequent retraining is necessary to maintain satisfactory prediction accuracy.

The plots also indicate that our method can start to provide a good failure prediction service only after 8 weeks of training. For the ANL log, *precision* is between 0.72 and 0.81 and *recall* is ranging between 0.56 and 0.66; for the SDSC log, *precision* is between 0.70 and 0.83 and *recall* is ranging between 0.59 and 0.70. In other words, our method does not need a long training phase to provide an acceptable prediction service. We shall also point out that even when the training set is 2 weeks (not shown in the figure), the predictor is still capable of capturing more than 43% of failures. In our previous study [20], we have found that runtime adaptive fault management is capable of providing positive performance gain as long as the underlying prediction mechanism can capture 30% of failures. Therefore, our dynamic meta-learning framework is able to serve a runtime fault tolerant tool after as few as 2 weeks of training.

Fig. 11 compares the prediction results with and without using the reviser. The plots show that dynamic revising can boost both *precision* and *recall*, and the improvement is up to 6%. As stated in Section 4, failures are rare events. In order to ensure these infrequent events to be analyzed, the parameters like *confidence* and *support* adopted in the association rules are typically chosen without much consideration to the effectiveness of the generated rules, thereby resulting in some rules that may mislead the prediction. The reviser acts like an additional learning process. It works on the candidate rules generated by the meta-learner, and filters out those rules that are not effective on the training set so as to improve prediction accuracy. The results shown in this figure demonstrate the necessity of using dynamic revising.

Observation #5: Dynamic revising can help improve failure prediction by filtering out bad rules of fault patterns.

Next, we examine the number of rules changed by using dynamic meta-learning, and the results are presented in Fig. 12. Each plot contains four curves representing the number of rules unchanged, the number of rules added by the meta-learner, the number of rules removed by the meta-learner, and the number of rules removed by the reviser respectively.

It is clear that the numbers are dynamically changing (i.e., the rules are added or removed) during the operation. Initially, when the training starts, there are only dozens of rules, which will be gradually popularized in the following retraining steps. For a period of 1 year, the knowledge repository will accumulate more than 100 rules for both systems. The number of unchanged rules starts to stabilize for the ANL log around the 70th week—about 140–160 rules. However, for the SDSC log, the number keeps increasing (up to 260 rules at the 120th week). In general, the number of rules used for runtime prediction with the ANL log is between 60 and 115, and it is between 100 and 190 for the SDSC log. The difference between these logs is due to many factors,

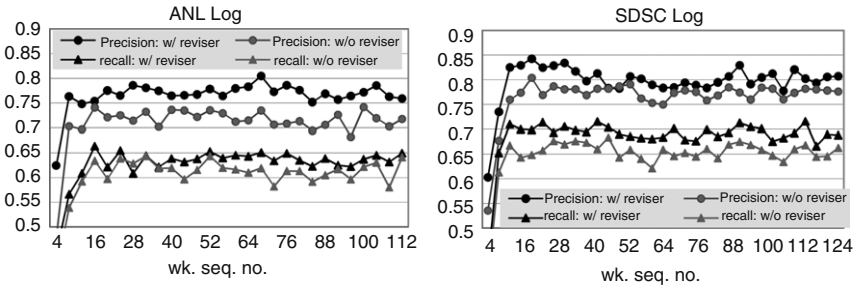


Fig. 11. Is it necessary to conduct dynamic revising? It is clear that dynamic revising can boost prediction accuracy by up to 6%.

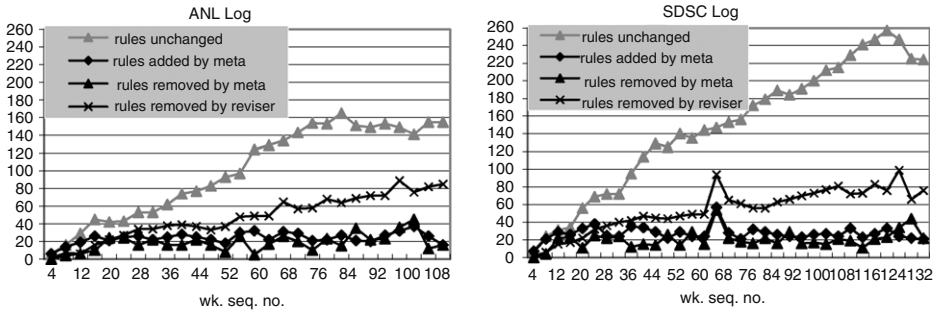


Fig. 12. Number of rules changed. We measure the numbers of rules that are unchanged, added by the meta-learner, removed by the meta-learner, and removed by the reviser. These numbers are constantly changing, indicating that the dynamic approach is essential for capturing varying pattern rules.

including system management, workload characteristics, etc. The change rate of rules, i.e., the ratio between changed and unchanged rules, ranges between 44% and 212%.

Further, we notice that a substantial change occurs during the 64th week with the SDSC log, where 57 rules are added and 148 rules are removed. The change is significant since normally about 20–30 rules are added and 50–80 are removed per retraining. Between the 60th and 64th week, a system reconfiguration occurs. Our method retrains the rules every 4 weeks, meaning that it extracts a set of rules at the 60th week and then retrains the system at the 64th week. Due to the system change, these two sets of rules are quite different, thereby resulting in significant rule changes. This is consistent with the results shown in Fig. 10.

The plots also show that the number of rules removed by the reviser is not trivial, by up to 80. This implies that the reviser can significantly remove non-trivial amount of rules. This result, combined with the information shown in Fig. 11, proves that dynamic revising is indispensable for better prediction by removing bad rules.

Observation #6: The rules of fault patterns are constantly changing during system operation. It further implies that dynamic relearning is essential for maintaining prediction accuracy.

5.2.3. Q3: How sensitive is prediction accuracy to prediction window size?

Fig. 13 presents prediction results by using different prediction windows (5 min, 15 min, 30 min, 45 min, 1 h, 1.5 h, and 2 h). The reason for choosing these durations is based on the results reported in [5,22] and our own experiments with different applications [20]. A time window smaller than 5 min may become too small for taking preventive action based on the prediction, whereas a time window larger than 2 h will induce an increased overhead on the system as it will require maintaining the history of all the events for the duration of 2 h. Also, the processing/analysis cost of these events for online failure prediction is not trivial.

The trend is obvious: the larger the prediction window is, the higher the *recall* is and the lower the *precision* is. When the prediction window is set to a larger number, it is more likely for the predictor to capture more events, thereby resulting in more chances to find a matching rule of the failure pattern. This leads to a higher value for *recall*, meaning the predictor can capture more failures. As an example, when the prediction window is set to 2 h, *recall* can be as high as 0.82. On the other hand, if the prediction window is large, it is also likely for the predictor to trigger false alarms due to the growing possibility of catching a misleading rule. For different prediction windows, the difference on *precision* is less than 0.25, and it is about 0.15 in terms of *recall*. Further, for all the cases, both *precision* and *recall* is generally above 0.55.

Observation #7: The larger the prediction window is, the higher the recall is and the lower the precision is.

5.2.4. Q4: How much runtime overhead is introduced?

Operation overhead depends on the size of the training set. Table 5 summarizes the overheads as a function of training data, in which the overhead is classified into two parts, namely rule generation overhead and rule matching overhead. These times are measured on a local PC configured with a 1.6 GHz Intel Pentium processor and 768 MB memory. Obviously, the overhead could be less when a more powerful PC is used.

The overhead mainly comes from rule generation, while the rule matching process (i.e., the event-driven predictor) is trivial, usually in dozens of seconds. As shown in the table, when the training set is set to 6 months (half of a year), the rule generation may take 6.0 min; and it can increase to 13 min when the training set is set to 30 months (two and a half year). Note that the rule generation process can be conducted in parallel when the production system is in operation; therefore, this cost should not be counted into the actual runtime overhead for failure prediction. The actual runtime overhead introduced by the event-driven predictor is normally less than 1.0 min. Thus, we believe that the framework is feasible as a

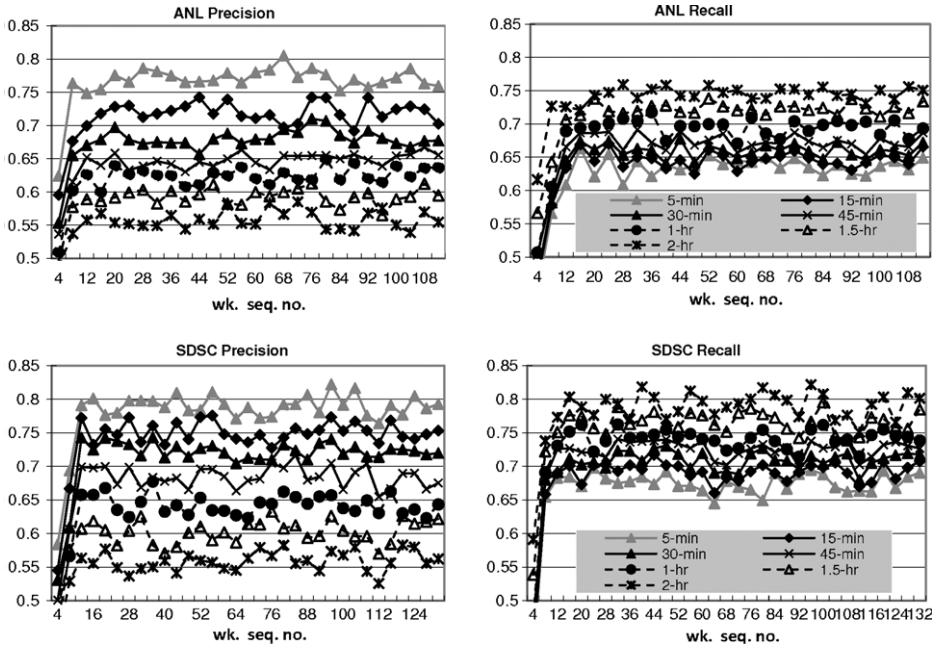


Fig. 13. Impact of Prediction Window. In general, the larger the window is, the higher the *recall* is and the lower the *precision* is.

Table 5
Operation overhead (in minutes) as a function of training size.

Training size (mo)	Rule generation				Rule matching
	Stat_Rule	Asso_Rule	Prob_Dist	Ensemble & revise	
3	<1	1	<2	1	<1
6	<1	2	<2	1	<1
12	<1	3	<2	2	<1
18	<1	4	<2	2	<1
24	<1	5	<2	3	<1
30	<1	6	<2	4	<1

runtime prediction mechanism. Combining the results shown in this table and in Fig. 9, we suggest that dynamic meta-learning on the recent 6 months is practical and time efficient.

Observation #8: The runtime overhead is trivial (e.g., in dozens of seconds), while the major overhead introduced by rule generation can be conducted in parallel when the target machine is in operation.

6. Related work

Recognizing the importance of fault tolerance, the community has paid much attention to failure prediction. Exiting predictive approaches can be broadly classified as *model-based* or *data-driven* methods. The model-based approach derives a probabilistic or analytical model of the system and triggers a warning when a deviation from the model is detected [10,17,37,18]. Examples include an adaptive statistical data fitting method called MSET presented in [38], Semi-Markov reward models described in [8], and a naive Bayesian based model to predict disk drive failures [13]. In large-scale systems, errors may propagate from one component to other component, which is commonly addressed by developing fault propagation models (FPM) [33]. *While model-based methods are effective for forecasting some failures, they seem too complicated to be practical for failure prediction in large-scale systems composed of tens of thousands of components.*

A data-driven method, such as using data mining techniques, attempts to learn failure patterns from historical data for failure prediction, without constructing an accurate model ahead of time. These methods extract fault patterns from system normal behaviors and detect abnormal observations based on the learned knowledge without assuming a priori model ahead of time. For example, the group at the RAD laboratory has applied statistical learning techniques for failure diagnosis in Internet services [3,1]. The SLIC (Statistical Learning, Inference and Control) project at HP has explored similar techniques for automating fault management of IT systems [4]. Sahoo et al. have applied association rules to predict failure events in a 350-node IBM cluster [30]. In [21, 22], Liang et al. have examined several data mining and machine learning techniques for failure forecasting in a Blue Gene/L system. Other representative works include system log analysis [31,26] and a prediction framework for networked systems [6].

While this paper is built upon existing studies, it distinguishes from the above studies at several aspects. First, unlike existing studies focusing on one specific predictive method, this paper presents a dynamic meta-learning framework to dynamically integrate existing predictive methods for better prediction. In this study, we have examined three widely used predictive methods, namely association rule-based learner [30,39], statistical rule-based learner [21], and probability distribution-based learner [31] in the framework. We believe that other predictive methods can be easily incorporated into our framework. Second, this study emphasizes dynamic training and learning, which is rarely examined in the literature. By means of real system logs from production systems, we have demonstrated that dynamic relearning is essential to capture behavior changes during system operation. By examining our framework in various ways, we have shown that using the most recent few months like 6 months makes a good tradeoff between accuracy and runtime overhead. Next, unlike offline log analysis studies, our prediction is event-driven, meaning that our framework triggers a

warning on the occurrence of events during system operation. An event-driven approach is well suited for online failure prediction. Last but not the least, in addition to presenting the key techniques for boosting prediction accuracy, we have also systematically analyzed our framework and answered several key questions commonly raised in failure prediction. It provides a deep insight into failure prediction in large-scale systems. To the best of our knowledge, we are among the first to comprehensively evaluate the impact of different factors in failure forecasting.

7. Summary

In this paper, we have presented a dynamic meta-learning prediction engine for large-scale systems. Recognizing problems in failure prediction, our prediction mechanism relies on two key techniques to improve prediction accuracy in real systems. *Meta-learning* is applied to boost prediction accuracy by integrating multiple predictive methods, while a dynamic approach is employed to train the rules of failure patterns at runtime. Our prediction mechanism does not require a long training phase by dynamically adjusting the training set during system operation. Further, it can adapt to system changes, even after a major system reconfiguration. Our case studies with real system logs have demonstrated its effectiveness with a good accuracy, e.g., capturing up to 82% of failures. The studies have also shown that the proposed mechanism is practical and well suited for forecasting failures in real systems.

Our study has some limitations that remain as our future work. First, in the current design, the prediction window size is fixed. Our on-going work includes adaptively changing this window size such that the system can automatically tune its size to reduce the training cost, without sacrificing the prediction accuracy. Second, we plan to examine other data mining methods, such as decision tree and neural network, to popularize our base learners. We will also investigate other ensemble learning techniques to improve the meta-learner. Finally, more case studies with a variety of HPC systems will be conducted. Although our case studies focus on the Blue Gene/L systems, we believe the proposed mechanism is applicable to other systems. For the systems that do not have an error checking and logging facility, the first step is to develop a monitoring tool which is capable of gathering fault-related information from various system components and archive the information in a centralized repository. The proposed framework can be easily extended to these systems by linking to their event repositories.

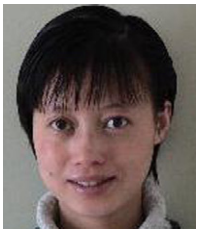
Acknowledgments

Zhiling Lan is supported in part by US National Science Foundation grants CNS-0834514, CNS-0720549, CCF-0702737, and a TeraGrid Compute Allocation. Susan Coghlan and Rajeev Thakur are supported by the Office of Advanced Scientific Computing Research, Office of Science, US Department of Energy, under Contract DE-AC02-06CH11357. We would like to thank John White at Revision[3] Company and Eva Hocks at San Diego Supercomputer Center for the discussion of the SDSC system log. Some preliminary results of this work were presented in [11,12].

References

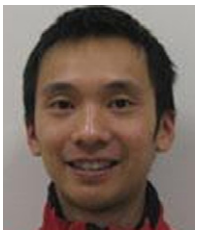
[1] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. Jordan, D. Patterson, Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization, in: Proc. of The 2nd IEEE International Conference on Autonomic Computing, ICAC '05, 2005.
[2] M. Buckley, D. Siewiorek, Comparative analysis of event tupling schemes, in: Proc. of Fault-Tolerant Computing, FTCS, 1996.
[3] M. Chen, A. Zheng, J. Lloyd, M. Jordan, E. Brewer, Failure diagnosis using decision trees, in: Proc. of ICAC'04, 2004.
[4] I. Cohen, J. Chase, Correlating instrumentation data to system states: a building block for automated diagnosis and control, in: Proc. of OSDI'04, 2004.
[5] E. Elnozahy, J. Plank, Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery, IEEE Transactions on Dependable and Secure Computing 1 (2) (2004) 97–108.
[6] S. Fu, C. Xu, Exploring event correlation for failure prediction in coalitions of clusters, in: Proc. of SC'07, 2007.
[7] A. Gara, M. Blumrich, D. Chen, G. Chiu, P. Coteus, M. Giampapa, R. Haring, P. Heidelberger, D. Hoenicke, G. Kopcsay, T. Liebsch, M. Ohmacht, B. Steinmacher-Burow, T. Takken, P. Vranas, Overview of the Blue Gene/L system architecture, IBM Journal of Research and Development 49 (2/3) (2005).
[8] S. Garg, A. Puliafito, K. Trivedi, Analysis of software rejuvenation using Markov regenerative stochastic petri net, in: Proc. of 6th International Symposium on Software Reliability Engineering, 1995.
[9] R. Gioiosa, J. Sancho, S. Jiang, F. Petrini, K. Davis, Transparent incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers, in: Proc. of SC'05, 2005.
[10] A. Goyal, S. Lavenberg, K. Trivedi, Probabilistic modeling of computer system availability, Annals of Operations Research (1987).
[11] P. Gujrati, Y. Li, Z. Lan, R. Thakur, J. White, A meta-learning failure predictor for Blue Gene/L systems, in: Proc. of ICPP'07, 2007.
[12] J. Gu, Z. Zheng, Z. Lan, J. White, E. Hocks, B. Park, Dynamic meta-learning for failure prediction in large-scale systems: a case study, in: Proc. of ICPP'08, 2008.
[13] G. Hamerly, C. Elkan, Bayesian approaches to failure prediction for disk drives, in: Proc. of ICML'01, 2001.
[14] J. Han, M. Kamber, Data Mining: Concepts and Techniques, 2nd edition, Morgan Kaufmann, 2006.
[15] J. Hansen, D. Siewiorek, Models for time coalescence in event logs, in: Proc. of Fault-Tolerant Computing, FTCS, 1992.
[16] P. Hargrove, J. Duell, Berkeley Lab Checkpoint/Restart (BLCR) for linux clusters, in: Proc. of SciDAC 2006 (publication LBNL-60520), 2006.
[17] J. Hellerstein, F. Zhang, P. Shahabuddin, A statistical approach to predictive detection, Computer Networks: The International Journal of Computer and Telecommunications Networking (2001).
[18] G. Hoffmann, F. Salfner, M. Malek, Advanced failure prediction in complex software systems, in: Proc. of SRDS'04, 2004.
[19] M. Joshi, R. Agarwal, V. Kumar, Mining needle in a haystack: classifying rare classes via two-phase rule induction, in: Proc. of SIGMOD'01, 2001.
[20] Z. Lan, Y. Li, Adaptive fault management of parallel applications for high performance computing, IEEE Transactions on Computers 57 (12) (2008) 1647–1660.
[21] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, R. Sahoo, Blue Gene/L failure analysis and models, in: Proc. of DSN'06, 2006.
[22] Y. Liang, Y. Zhang, H. Xiong, R. Shaoo, Failure prediction in IBM BlueGene/L event logs, in: Proc. of ICDM'07, 2007.
[23] Y. Li, Z. Lan, P. Gujrati, X. Sun, Fault-aware runtime strategies for high performance computing, IEEE Transactions on Parallel and Distributed Systems 20 (4) (2009) 460–473.
[24] A. Oliner, L. Rudolph, R. Sahoo, Cooperative checkpointing theory, in: Proc. of the International Parallel and Distributed Processing Symposium, IPDPS, 2006.
[25] A. Oliner, R. Sahoo, J. Moreira, M. Gupta, A. Sivasubramaniam, Fault-aware job scheduling for Blue Gene/L systems, in: Proc. of IPDPS'04, 2004.
[26] A. Oliner, J. Stearley, What supercomputers say: a study of five system logs, in: Proc. of the International Conference on Dependable Systems and Networks, DSN, 2007.
[27] W. Peng, T. Li, S. Ma, Mining logs files for computing system management, in: Proc. of ICAC'05,2005.
[28] R. Polikar, Ensemble based systems in decision making, IEEE Circuits and Systems Magazine 6 (3) (2006).
[29] D. Reed, C. Lu, C. Mendes, Big systems and big reliability challenges, in: Proc. of Parallel Computing, 2003.
[30] R. Sahoo, A. Oliner, I. Rish, M. Gupta, J. Moreira, S. Ma, Critical event prediction for proactive management in large-scale computer clusters, in: Proc. of SIGKDD'03, 2003.
[31] B. Schroeder, G. Gibson, A Large-scale study of failures in high performance computing systems, in: Proc. of DSN'06, 2006.
[32] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, P. Stodghill, Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs, in: Proc. of SC'04, 2004.
[33] M. Steinder, A. Sethi, A survey of fault localization techniques in computer networks, Science of Computer Programming 53 (2004).
[34] The Blue Gene system at ANL. Available at http://www.bgl.mcs.anl.gov/.
[35] The Blue Gene system at SDSC. Available at http://www.sdsc.edu/us/resources/bluegene/.
[36] The TOP500 supercomputer site. Available at http://www.top500.org/.
[37] K. Trivedi, K. Vaidyanathan, A measurement-based model for estimation of resource exhaustion in operational software systems, in: Proc. of ISSRE'99, 1999.

[38] K. Vaidyanathan, K. Gross, MSET performance optimization for detection of software aging, in: Proc. of ISSRE, 2003.
[39] R. Vilalta, S. Ma, Predicting rare events in temporal domains, in: Proc. of ICDM'02, 2002.
[40] C. Wang, F. Mueller, C. Engelmann, S. Scott, A job pause service under LAM/MPI+BLCR for transparent fault tolerance, in: Proc. of IPDPS'07, 2007.
[41] G. Weiss, Mining with rarity: a unifying framework, ACM SIGKDD Explorations 6 (1) (2004) 719.
[42] Y. Zhang, M. Squillante, A. Sivasubramaniam, R. Sahoo, Performance implications of failures in large-scale cluster scheduling, in: Proc. of Workshop on Job Scheduling Strategies for Parallel Processing, 2004.



Zhiling Lan received the B.S. degree in Mathematics from Beijing Normal University in 1992, the MS degree in Applied Mathematics from Chinese Academy of Sciences in 1995, and the Ph.D. degree in Computer Engineering from Northwestern University in 2002. She is currently an associate professor of computer science at the Illinois Institute of Technology. Her research interests are in the area of parallel and distributed systems, in particular, fault tolerance, dynamic load balancing, and performance analysis and modeling.

Jiexing Gu is currently a Master student at the Computer Science Department of IIT. She received her B.E. degree from Nanjing University, China. Her research interest is parallel and distributed computing in general and is currently working on fault tolerance.



Ziming Zheng received his B.S. and M.S. degrees in the University of Electronic Science & Technology of China in 2003 and 2006. He is now a Ph.D. candidate of Computer Science at Illinois Institute of Technology since 2007. His research focuses on fault tolerance in large-scale computer systems. He is also an IEEE student member.

Rajeev Thakur is a Computer Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory. He is also a Fellow in the Computation Institute at the University of Chicago and an Adjunct Associate Professor in the Department of Electrical Engineering and Computer Science at Northwestern University. He received a B.E. from University of Bombay, India, in 1990, an M.S. from Syracuse University in 1992, and a Ph.D. from Syracuse University in 1995, all in Computer Engineering. His research interests are in the area of high-performance computing in general and particularly in parallel programming models and message-passing and I/O libraries.

Susan Coghlan has worked on parallel and distributed computers for 20 years. Throughout her career, she has addressed a diverse range of challenges, including developing scientific applications, such as a model of the human brain and managing ultra-scale supercomputers like ASCI Blue Mountain. In 2000, she co-founded a research laboratory in Santa Fe. In recent years, Susan was involved in the creation of the Argonne Leadership Computing Facility (ALCF), where she was responsible for the installation and operation of the 557-TF Blue Gene/ P system. In her current role as Associate Division Director for the ALCF, she is project manager for the facility's next big system.