Toward Predictive Failure Management for Distributed Stream Processing Systems

Xiaohui Gu[†] Spiros Papadimitriou[‡] Philip S. Yu[*] Shu-Ping Chang[‡]

[†]North Carolina State Univ.
gu@csc.ncsu.edu

[‡]IBM Research
{spapadim,spchang}@us.ibm.com

[*]U. of Illinois at Chicago
psyu@cs.uic.edu

Abstract

Distributed stream processing systems (DSPSs) have many important applications such as sensor data analysis, network security, and business intelligence. Failure management is essential for DSPSs that often require highly-available system operations. In this paper, we explore a new *predictive failure management* approach that employs online failure prediction to achieve more efficient failure management than previous reactive or proactive failure management approaches. We employ light-weight stream-based classification methods to perform online failure forecast. Based on the prediction results, the system can take differentiated failure preventions on abnormal components only. Our failure prediction model is *tunable*, which can achieve a desired tradeoff between failure penalty reduction and prevention cost based on a user-defined reward function. To achieve low-overhead online learning, we propose *adaptive data stream sampling schemes* to adaptively adjust measurement sampling rates based on the states of monitored components, and maintain a limited size of historical training data using reservoir sampling. We have implemented an initial prototype of the predictive failure management framework within the IBM System S distributed stream processing system. Experiment results show that our system can achieve more efficient failure management than conventional reactive and proactive approaches, while imposing low overhead to the DSPS.

1 Introduction

Many emerging applications call for sophisticated real-time processing over dynamic data streams such as sensor data analysis and network traffic monitoring. Distributed stream processing systems (DSPSs) have been developed to achieve scalable continuous query (CQ) processing. However, today's DSPSs are still vulnerable to various software and hardware failures. For example, in the deployed IBM System S stream processing system [16], the system log records 69 significant failure incidents during one month period. Previous failure management work (e.g., [20, 4, 15, 14]) can be classified into two categories: (1) *reactive* approach that takes recovery actions after a failure happens, and (2) *proactive* approach that takes advance preventive actions such as backup for *all* components (i.e., operators or hosts) at *all* time. The reactive approach does not have any preventive cost but can incur significant failure penalty (e.g., query result loss, query processing interruption) for stream applications. The proactive approach offers better fault tolerance but can be impractical for load-intensive DSPSs that often do not have enough resources to take preventive actions on all components. To address the problem, this paper explores a new *predictive failure management* approach that employs online failure prediction to achieve *selective*, *just-in-time*, and *informed* failure prevention. By "selective", we mean that the system performs preventive actions on failing components only. By "just-in-time", we mean that the preventive action is performed just before the failure incident. By "informed", we mean that the system can take suitable preventive actions (e.g., backup, isolation, or migration) based on the predicted failure type. As a result, our system can achieve more resource-efficient fault-tolerant stream processing than conventional reactive or proactive approaches.

We need to address a set of new challenges to achieve efficient predictive failure management. First, continuous stream processing applications demand *on-line* failure prediction [10] that can perform realtime analysis on collected measurements and raise timely alerts before a failure incident happens. Second, the failure learning schemes should be light-weight since the normal query processing can be resource-intensive by itself. Moreover, the failure management for dynamic stream environments is desired to be tunable, which should be able to adjust its behavior (i.e., detection rate, false-alarm rate) based on available resources in the system and possible failure penalties. Third, we need to provide new failure prevention schemes that can leverage failure alerts to ensure continuous system operation.

In this paper, we present the design and implementation of a novel predictive failure management system addressing the above challenges. Each component is associated with a *feature stream* that includes a set of periodically sampled

system-level and application-level metrics (e.g., available memory, free CPU time, virtual memory page-in/page-out rates, tuple processing time, buffer queue length). We employ light-weight stream classifiers to achieve online failure prediction, which continuously classifies received feature samples into three states: *normal*, *alert* and *failure*. The classifier raises failure alerts when the component state falls into the alert or failure state. The classifier is continuously updated using labelled measurement data. Each measurement is labelled with failure or normal based on user-defined failure predicates [8] (e.g., processing time $>$ 50ms, throughput $<$ 100). The alert state corresponds to a warning region where the predictor will raise a failure alert to trigger a proper failure prevention action. Specifically, this paper makes the following contributions:

- We design and implement the first online failure prediction model using stream-based decision tree classification methods. Our prediction model is tunable, which can achieve optimal trade-off between failure penalty reduction and prevention cost based on a user-defined reward function.
- We propose a new differentiated failure prevention approach that uses failure alerts as guidance to perform *just-in-time failure preventions* (e.g., isolation, backup, migration) on failing components only. The failure prevention scheme includes an iterative component inspection algorithm that can overcome prediction errors and provide feedback to the predictor for adapting to dynamic stream environments.
- We propose adaptive data stream sampling schemes to achieve low-overhead online failure learning. Each monitoring component dynamically adjusts the sampling rate based on the state of the monitored component. Each analysis component employs reservoir sampling [22] to incrementally maintain a limited size of training data selected from the infinite feature stream.

We have implemented an initial prototype of the predictive failure management framework inside the IBM *System S* stream processing system [16]. We conduct experiments using real query networks taken from the System S reference applications [9] and real data stream workloads. The experimental results show that i) our stream classification schemes can achieve good prediction accuracy for a range of software failures caused by common program bugs; ii) the predictive failure management can achieve better cost-efficiency (i.e., larger reward) than conventional reactive or proactive approaches; and iii) the on-line failure learning is feasible, which imposes low overhead to the stream processing cluster.

The rest of the paper is organized as follows. Section 2 presents the system model. Section 3 presents the predictive failure management design and algorithms. Section 4 presents the prototype implementation and experimental results. Finally, the paper concludes in Section 5.

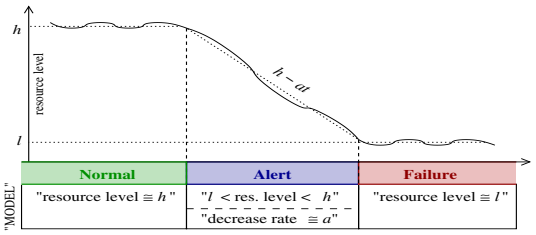

Figure 1. Triple-state operator classification.

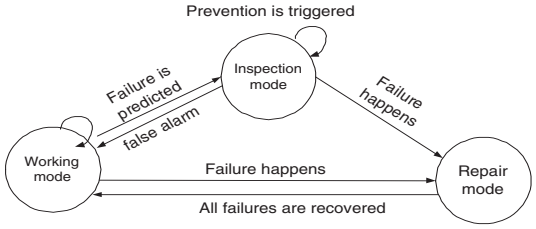

Figure 2. Triple-mode component operation.

2 System Model

We consider large-scale, shared cluster systems executing many stream processing applications concurrently. Each stream application consists of a set of components (i.e., query operators) connected into a directed acyclic graph that can be placed at different hosts. These components are typically provided by different application developers, which are often failure-prone. Drawing from our experience on building the IBM System S, we observe that most software failures are caused by common program bugs such as memory leak, buffer management errors, and CPU starvation caused by iterator update errors. Different from instant failures (e.g., machine crash) that can be easily detected, those software failures are often latent and hard to detect. The focus of our work is to manage latent software failures and minimize their negative impact on the system operation.

Application developers or system administrators can use failure predicate [8] to specify the failure type that should be watched by the system. Common failure types include service level objective violations [6] and system bottleneck (i.e., full stream buffer). For each failure type, we create an on-line failure forecast model called *predictor*, which can continuously classify the state of a monitored component into three states: *normal*, *alert*, and *failure*, illustrated by Figure 1. The *failure* state is described using a given predicate that characterizes *what* we are trying to predict. The *alert* state corresponds to a set of values in a dynamically defined time interval "preceding" the failure denoted by a *pre-failure interval* PF. The rest values correspond to the normal state. The predictor will raise a failure alert when the component state is classified as alert or failure.

Correspondingly, a component can operate under three different modes: (1) *working mode*, (2) *inspection mode*, and (3) *repair mode*, illustrated by Figure 2. When the component state is classified as *alert* by a predictor, the component is put into the inspection mode. Based on the predicted failure type, we can take different preventive actions accordingly, which will be discussed in Section 3.2. Different from conventional proactive approach, the failure prevention is performed in a just-in-time fashion.

Following the standard prediction accuracy measures, we define the detection rate A_D and false alarm rate A_F of a failure predictor as follows,

Definition 2.1. *Given the number of true positive failure predictions N_{tp}, the number of false-negative failure predictions N_{fn}, the number of false positive predictions N_{fp}, and the number of true negative predictions N_{tn}, the detection rate A_D and false alarm rate A_F are defined in a standard way as*

$$A_D = \frac{N_{\text{tp}}}{N_{\text{tp}} + N_{\text{fn}}}, A_F = \frac{N_{\text{fp}}}{N_{\text{fp}} + N_{\text{tn}}} \quad (1)$$

To quantify the efficiency of predictive failure management, we allow the user to define a *reward function* that denotes the net benefit of the predictive failure management system. In our experiments, we denote the reward function as follows [1],

Definition 2.2 (Failure Prevention Reward Function)**.** *Given a monitored component (i.e., operator or host) o_i and its associated failure predictor $\mathcal{P}_i$, let α_i denote the mean penalty reduction of each successfully predicted failure incident, let β_i denote the mean failure prevention cost caused by each false-alarm, let A_D and A_F denote the detection rate and false alarm rate achieved by the predictor, the prediction reward $\mathcal{R}_i$ for o_i is defined by*

$$\mathcal{R}_i = A_D \cdot \alpha_i - A_F \cdot \beta_i \quad (2)$$

Different from previous work, the predictive failure management provides a *tunable* solution. We can tune the predictor's detection rate and false-alarm rate by adjusting its pre-failure interval. The larger the pre-failure interval, the more likely the predictor can raise an alert since more measurements are incorporated into the alert state. At one extreme, if we set $PF = 0$, our scheme becomes conventional reactive approach where the alert state is always empty and no alerts will be generated by the predictor. At the other extreme, if we set $PF = \infty$, our scheme becomes traditional proactive approach that performs preventive actions unconditionally. However, in many cases, the optimal solution lies in-between the two extremes. The goal of our research is to provide tunable failure management system that can dynamically adapt itself to achieve optimal tradeoff between fault-tolerance benefit and cost.

[1] Our approach can be applied to other reward functions as well.

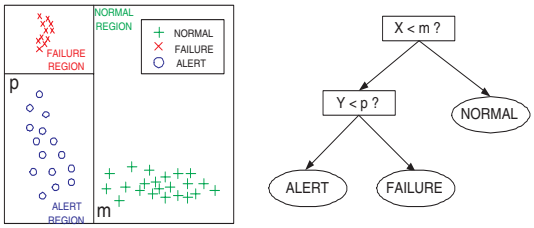

Figure 3. Failure prediction using decision trees.

3 Design and Algorithms

In this section, we present the design and algorithms of our learning-based failure management system that includes (1) online failure prediction model using stream-based decision tree classification methods; (2) differentiated failure prevention schemes using failure alerts; and (3) adaptive feature stream sampling schemes to achieve light-weight, low-overhead online failure learning.

3.1 Online Failure Prediction

Our system continuously collects feature metrics about different components and classifies received feature tuples to identify abnormal component behavior. In this work, we chose decision tree classification methods [17] since they produce rules with direct, intuitive interpretation by non-experts. Thus, the predictor can not only raise advance failure alerts but also provides cues for possible failure causes. Each decision tree classifier is trained on historical measurement data, which are appropriately labelled with "normal", "alert", or "failure", illustrated by Figure 3. The points closely follow the memory leak scenario, but are spread out for clearer visualization; the x-axis can be thought of as available memory and the y-axis as page-out rate. The system can label feature tuples as normal or failure based on the failure predicate. Then, a set of points preceding the failure incidents within the pre-failure interval are labelled as "alert." The decision tree is trained using feature tuples from all three states. Periodically, as the history grows based on feedback from the system, the decision tree is updated if its accuracy is low. For state classification, decision trees essentially apply a sequence of threshold tests on the features. For example, in Figure 3, the predicate that corresponds to the alert region is "$X < m$ and $Y < p$", and can be determined by following the path in the tree which leads to the leaf labelled "alert". Our prediction model incorporates multiple features and employs ten-fold cross-validation to select those features with best predictive power [10].

As mentioned in Section 2, we can tune the alertness of the failure predictor by adjusting the pre-failure interval length. Our experimental results have confirmed this hypothesis, which will be shown in section 4. The system

updates the prediction performance counters (i.e., N_{fn}, N_{fp}, N_{tp}, N_{tn}) of the current predictor based on the feedback from the system and calculates the reward value $\mathbf{R}$. If the reward value of the current predictor drops below a pre-defined threshold, the system triggers the adaptation algorithm to select a new decision tree that is more suitable for the current environment.

3.2 Just-In-Time Failure Prevention

We now present the just-in-time failure prevention algorithm that is triggered by the failure alert. As mentioned in Section 2, a monitored system component (i.e., operator or host) can operate under three different modes. The component is said to be in the working mode if its state is classified as *normal* by all predictors. When the component state is classified as *alert* by any predictor, the component is put into the inspection mode. If a failure does happen later as predicted, the component transfers into the repair mode. The system also sends a positive feedback to the prediction model that it has predicted a failure successfully. If the component state is classified as normal later or the inspection times out, the system considers the prediction model issues a false-alarm. The component under inspection returns to the working mode and the prediction model is notified with a feedback that it has made a false-positive error.

Prevention for failing operators If the failing component is a query operator, we first isolate the failing operator from the other operators running on the same host so that we can confine the scope of the expected operator failure. The isolation can be achieved by running the failing operator on a separate virtual machine [5] from the other normal operators. We keep the operator's input workload and computing environment unchanged and use a higher sampling rate to collect important pre-failure measurements for training the failure prediction model.

To achieve fault tolerance, we also create a temporary backup operator on a working host that processes the same input stream(s) as the failing operator in parallel. Depending on the predicted failure type, the system may take different preventive actions on the backup operator to minimize the failure impact. For example, we can increase the resource allocation to the backup operator if the failure is caused by insufficient resources. However, if the predicted failure is caused by an internal software bug (e.g., memory leak), the backup operator may experience the same failure later. However, the backup operator may delay the failure incident by starting from a clean state on a clean host [21], which provides a window of opportunity for the system to find corrective solutions.

If the failure happens on the inspected operator as predicted, the operator under the inspection mode enters the repair mode. We can then perform failure diagnosis on the failed operator using failure diagnosis tools (e.g., [7]) and collected pre-failure measurements. Based on the diagnostic result, we can take proper failure recovery actions (e.g., generating proper software patch and applying the path on the backup operator) to avoid future failure incidents. If the predicted failure does not happen, we say the predictor issues a false alarm. The inspected operator returns to the working mode. The system releases the isolation on the inspected operator and removes the backcup operator from the system. The downstream operator(s) will be notified to receive data from the original operator.

Prevention for failing hosts. If the failing component is a host running several operators, the preventive action involves more steps. First, we want to migrate normal operators from the failing host to a working host. An operator is said to be normal if the system is sure that the operator is not the cause of the host failure. By migrating the normal operators out, we can avoid the penalty of the anticipated host failure on those queries involving normal operators. Second, we let those suspicious operators that might be the cause of the host failure continue to run on the failing host for collecting pre-failure information. In parallel, we also create temporary backups for those suspicious operators on working hosts to achieve fault tolerance. Similar to the previous case, we take proper preventive actions on the backup operators based on the predicted failure type. If a predicted failure does happen, the host transfers into the repair mode and those temporary backup operators become permanent. Next, we can perform failure diagnosis to identify the cause of the failure. If the host failure is caused by the software bugs in some problematic operators, we also need to perform software patch on their backup operators to avoid future failure occurrences. If the failure prediction model issues a false alarm, the host is set back to working mode, which can provide resources for existing or new operators. We also remove those temporary backup operators from the system since they are no longer needed.

3.3 Adaptive Data Sampling

To achieve accurate failure prediction, the analysis component wishes to receive as much information as possible. However, a large-scale stream system can concurrently host thousands of operators. Each operator can be associated with tens of feature metrics. Collecting feature metrics at a fine time granularity can impose a considerable monitoring overhead. Thus, we propose an adaptive sampling scheme to reduce monitoring overhead with the help of prediction feedback. The basic idea is to use a low sampling rate when the analysis result is normal and switch to a high sampling rate when the analysis result is abnormal. For each monitored component whose state is classified as normal by the prediction model, we use a low sampling rate to reduce monitoring overhead since detailed measurement is unnecessary. When the prediction model raises a failure alarm on the component, we increase the sampling rate to collect more precise information on the abnormal component, which allows the prediction model to make more

accurate state classification. Later, if the predictor classifies the component as normal based on the detailed information, the sampling rate is reduced.

To achieve robust failure prediction in dynamic stream environments, the predictor wishes to store the execution behavior of a component under different stream workload conditions. However, given limited buffer space, the predictor can only retain a subset of life-long historical data as training data. To achieve compact prediction model, we employ reservoir sampling [22] to incrementally maintain a desired size of training data. We use non-uniform tuple retention probabilities p_s to keep the most important data. For each time instant t, we observe a set of j measurements $\langle m_{1,t}, \ldots, m_{j,t} \rangle$ with a corresponding state label ℓ_t (i.e., normal, alert, or failure) based on diagnostic feedback and the pre-failure interval. Generally, we should use higher retention probability $p_s(\ell_t, m_{1,t}, \ldots, m_{j,t})$ for more important tuples. The most significant factor determining the importance of a set of measurements is its state label. Failure and alert points are much rarer than normal points. Thus, we use the probability of each state and employ the biasing scheme of [18], where

$$p_s(\ell_t = l) := \frac{\alpha}{N_l^e}, \quad \text{with } \alpha = \frac{U}{\sum_l N_l^{1-e}}.$$

U is the number of tuples we wish to retain, and N_l is the number of tuples having label l. The exponent $e, 0 \leq e \leq 1$, determines the relative bias. For $e = 0$ the sampling is unbiased (i.e., uniform random with the same probability regardless of ℓ_t). For $e = 1$, we retain an equal number of samples from each class, regardless of the original number of points N_l in each class. We should note that, for $e > 0$ and for finite datasets, the retention probabilities may be larger than one for very rare classes. In this case, we set them to 1 and proportionally increase the sampling probabilities, so that the total number of points retained is U. In the experiments, we set $e = 0.5$ and we tried ratios of $U/N = 50\%$ and 30% , where $N = \sum_l N_l$ is the total number of measurement feature tuples.

4 Experiments

We have implemented the predictive failure management framework within the IBM System S distributed stream processing infrastructure and deployed on a commercial Linux cluster system. The cluster system consists of about 250 blade servers connected by Gigabit Ethernet. Each server host is equipped with Intel Xeon 3.2GHZ CPU and [2,4] GB memory. All of our experiments are conducted on the cluster system.

For failure prediction, the system continuously collect a set of host-level and operator-level metrics, listed by Table 1. The host-level metrics include available memory, virtual memory page in/out rate, free CPU time, free disk space, which are collected by periodically querying the operating system's `/proc` interface. We also perform simple application instrumentation on the tested query operators so that we can continuously collect dynamic operator states.

Operator metric	Description
INTUPLE	num. of tuples received/sec
OUTTUPLE	num. of tuples generated/sec
AVGPROCESSINGTIME	the mean per-tuple processing time
SAMPLEDTHROUGHPUT	num. of query operations/sec
MEANQUEUELENGTH	mean queue length
INPUTWORKLOAD	num. of output tuples generated by upstream operators per sec.
Host metric	**Description**
FREECPU	percentage of free CPU cycles
AVAILMEM	available memory (MB)
PAGEIN	virtual page in rate
PAGEIN	virtual page out rate
FREEDISK	free disk space (MB)

Table 1. Monitoring metrics.

In our experiments, the case study query network [12] comprises eleven CQ operators including three source operators, one load diffusion operator, six multi-way sliding-window stream join operators, and one fusion operator. Each source operator continuously produces a real application data stream by replaying one of the following trace files: wide-area TCP traffic streams taken from the Internet Traffic Archive [2] or news video streams taken from NIST TRECVID-2005 data set [3]. The load diffusion operator dispatches input tuples to different join operators for parallel processing of multi-way stream join. Each join operator continuously correlate tuples from different streams using a pre-defined join predicate. For video streams, the join predicate is to find similar video images for hot topic detection [11]. For network traffic streams, the join predicate is to find network packets with common source and destination IP addresses. All join results are merged at the fusion operator. Each operator runs at a different host.

The current implementation of our system focuses on predicting query processing failures caused by common program bugs. We have tested our system on the following program bugs: (1) *memory leak (memLeak) bug*: the CQ program forgets to free the allocated memory that is not needed anymore. The memory consumption accumulates as the program periodically executes the buggy memory allocation code segments; (2) *infinite loop (loopErr) bug*: the CQ program spawns a CPU-bounding thread that includes an infinite loop error (e.g., caused by iterator update mistakes) where a CPU-intensive operation is repeatedly executed; and (3) *buffer update (bufferErr) bug*: the CQ program forgets to delete a processed tuple from its buffer. These program bugs are the most common bugs we found in our real stream processing applications. The on-line decision tree classifier is implemented based on the canonical C4.5 decision tree software package [1].

To test our failure prediction algorithms, we activate the buggy code segments in different CQ operators. Each

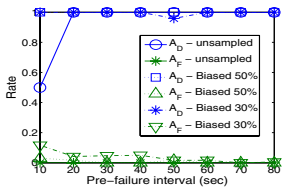

Figure 4. Prediction accuracy for memLeak bugs under network workloads.

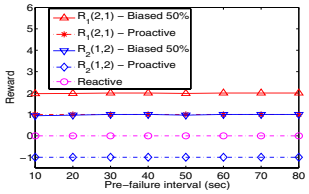

Figure 5. Prediction reward for memLeak bugs under network workloads.

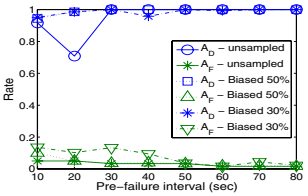

Figure 6. Prediction accuracy for loopErr bugs under network workloads.

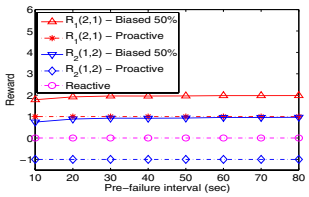

Figure 7. Prediction reward for loopErr bugs under network workloads.

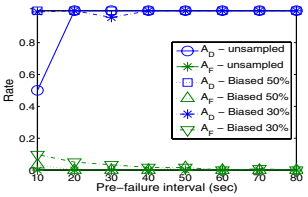

Figure 8. Prediction accuracy for bufferErr bugs under network workloads.

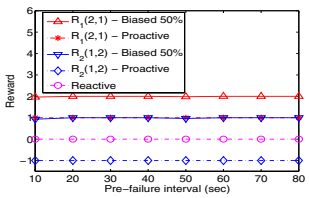

Figure 9. Prediction reward for bufferErr bugs under network workloads.

experiment run lasts 1000-1400 seconds. The buggy code is executed at different time instants to test the applicability of our failure prediction model for time-varying stream workloads. Each fault lasts 50 seconds, which means we activate the buggy code at time t and remove the buggy code at time $t + 50$. Each prediction model maintains an ensemble of eight classifiers using different pre-failure intervals $PF = 10$, 20, 30, 40, 50, 60, 70, and 80. We always use standard ten-fold cross-validation during classifier training. Each experiment was repeated, using the complete measurement traces, as well as selected samples (with measurement retention probabilities as described in Section 3.3) that retained 50% and 30% of the total measurements. In order to determine failure detection and false alarm rates, we divided each test trace into consecutive segments of 100 seconds. Each segment was labelled either as "failure event" or "normal operation," depending on whether it contained an injected fault or not. We raised failure alerts based on the criterion explained in Section 3.1. Each of the segments in the test trace was labelled as "alert raised" or "no alert" depending on whether an alert was raised or not at any time instant within that interval. Based on this information, we calculated $N_{\text{tp}}, N_{\text{tn}}, N_{\text{fp}}$, and N_{fn} for each failure prediction model. We can then calculate the detection rate A_D and false-alarm rate A_F according to Definition 2.1.

We first conduct a set of experiments using the continuous query network processing the network traffic data. Figure 4, Figure 6, and Figure 8 show the detection rate (A_D) and false-alarm rate (A_F) achieved by different decision tree classifiers for predicting the failures caused by the memory leak bug, infinite loop bug, and buffer update bug, respectively. These decision trees are trained using different pre-failure intervals ranging from 10 to 80 seconds. We also compare the performance of the classifier using full training data, denoted by "A_D - unsampled" and "A_F - unsampled", with that of the classifiers using 50% biased sampling (i.e., retaining 50% of the original complete training data), denoted by "A_D - Biased 50%" and "A_F - Biased 50%", and 30% biased sampling (i.e., retaining 30% of the original complete training data), denoted by "A_D - Biased 30%" and "A_F - Biased 30%" We observe that for the network traffic data, our failure prediction models can achieve almost perfect predictions (i.e., 100% detection rate and 0% false-alarm rate). The results indicate that the multi-way sliding-window join operators under the network traffic exhibit easily distinguishable difference between normal executions and faulty executions.

We also observe that bias-sampling can effectively maintain prediction accuracy while greatly reducing the sampling overhead. It is also interesting to note that the trees trained on the bias-sampled data can sometimes achieve a higher detection rate than the tree trained on the full data, without much change in false alarm rate. The increase in A_D is because the sampling is biased towards non-normal points, making the classifiers less conservative in raising an alert. However, A_F does not increase correspondingly in this case because the normal behavior is better separated from the faulty behavior in the measurement space. Generally speaking, bias-sampling can maintain the detection rates of different classifiers with a slight increase in false-alarm rates for classifiers trained with small pre-failure intervals. Another interesting observation is that A_F initially decreases significantly. This is because we raise the alert based on the majority voting criterion described in Section 3.1. When pre-failure interval is small, the majority voting window (e.g., $W = \lceil 0.1PF \rceil = 1$, $PF = 10$) is fairly small. Thus, a misclassification of one point may lead to a false alarm. However, the voting mechanism quickly helps reduce the false alarm rate when $PF \geq 30$. Moreover, as we increase pre-failure interval, the false-alarm rate did not show noticeable increase since the alarm raising becomes more "patient" with bigger majority voting windows. These results are encouraging since we can raise early alerts without increasing false-alarms, which can provide the system

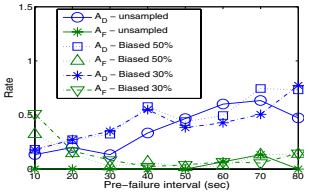

Figure 10. Prediction accuracy for memLeak bugs under video workloads.

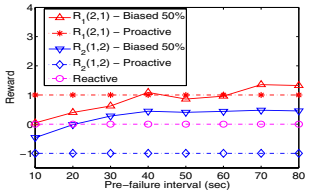

Figure 11. Prediction reward for memLeak bugs under video workloads.

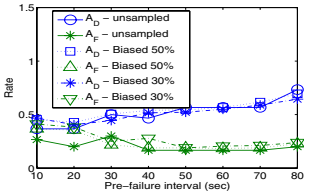

Figure 12. Prediction accuracy for loopErr bugs under video workloads.

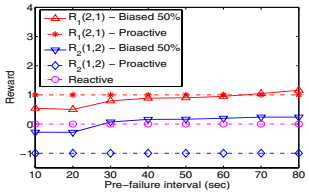

Figure 13. Prediction reward for loopErr bugs under video workloads.

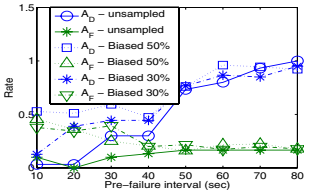

Figure 14. Prediction accuracy for bufferErr bugs under video workloads.

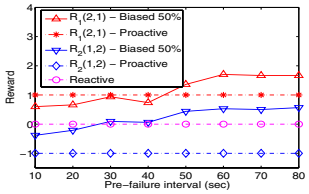

Figure 15. Prediction reward for bufferErr bugs under video workloads.

with sufficient time to take the preventive actions.

One important advantage of our approach is its tunability compared to conventional reactive and proactive approaches. The system can maximize its failure management reward by tuning its prediction models using pre-failure interval. Figures 5, 7 and 9 show the reward values achieved by different prediction models for predicting three different types of failures[2] The reward value is calculated according to Definition 2.2. We also used different reward function parameters to show their effects on prediction model selection. The first reward function ($R_1 = 2 \cdot A_D - 1 \cdot A_F$) uses a high prevention reward ($\alpha = 2$) and low prevention cost ($\beta = 1$) while the second reward function ($R_2 = 1 \cdot A_D - 2 \cdot A_F$) uses a low prevention reward ($\alpha = 1$) and high prevention cost ($\beta = 2$). The reactive approach always achieve zero reward independent of the reward function definition since it does perform any failure prevention (i.e.,$A_D = 0, A_F = 0$). The proactive approach goes to the other extreme that always takes preventive actions with $A_D = 1, A_F = 1$. In this case, the reward of the proactive approach is $R_1 = 2 * A_D - 1 * A_F = 1$ and $R_2 = 1 * A_D - 2 * A_F = -1$. In contrast, our approach employs online failure prediction to achieve more efficient failure preventions. From Figures 5, 7 and 9, we observe that our approach can consistently achieve higher failure management reward than conventional reactive and proactive approaches. Since our prediction models can achieve near-perfect prediction for the network traffic data, different prediction models have similar reward. In this case, we may want to choose the prediction model that can raise failure alerts just early enough for the system to take proper preventive actions.

We then conduct the second set of experiments using the video stream data that have much lower stream rates than the network data but demand more resource-intensive join computations. Figures 10, 12 and 14 show the detection rates and false-alarm rates achieved by eight decision tree classifiers using different pre-failure intervals for three different faults, respectively. We observe that our failure prediction models can still achieve reasonably good prediction accuracy by choosing the optimal classifiers with proper pre-failure intervals. However, the prediction models are generally less perfect for the video stream data than for the network traffic data. The reason is that the difference between normal feature streams and abnormal feature streams become more subtle under lower stream rates.

Figures 11, 13 and 15 shows the reward values achieved by different failure management algorithms using two different reward functions ($R_1 = 2 \cdot A_D - 1 \cdot A_F$ with $\alpha = 5$, $\beta = 1$ and $R_2 = 1 \cdot A_D - 2 \cdot A_F$ with $\alpha = 1$, $\beta = 5$) for three different program bugs, respectively. The reward values of different prediction models show more variance than previous set of experiments since the accuracy of different prediction model has larger difference. However, we observe that our algorithm can still achieve better failure management reward than conventional proactive and reactive approaches by choosing a proper pre-failure interval.

We now evaluate the overhead of our approach. Each box plot in Figure 16 summarizes the distribution of per-tuple overheads including feature tuple collection time, feature tuple classification time, and prediction classifier training time. The solid line inside each box is the median per-tuple overhead (i.e., 50% quantile), in milliseconds. The bottom and top of each box correspond to the 25% and 75% quantiles, respectively, and the whiskers show the full extent (minimum to maximum) of the values. The first two plots in Figure 16 show the measured time for collecting system log stream (i.e., host-level metrics) and application instrumentation data stream (i.e., operator-level metrics). We observe that each log stream sampling needs about 25ms, which involves disk operation to read from system log files. However, the system log data only need to be collected once for one host and shared among all

[2]For our algorithm, we only show the results of the prediction models using 50% biased sampling since other alternatives of our algorithm (i.e., unsampled or 30% sampled) show similar trend.

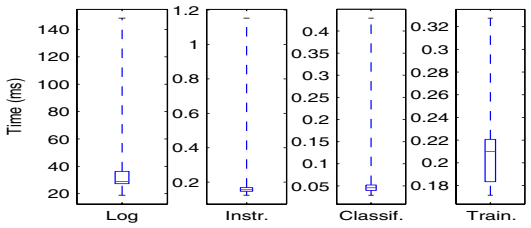

Figure 16. Failure learning overhead.

the operators on the host. Thus, the system log collection overhead is amortized over all co-located operators. In contrast, each instrumentation data collection only needs about 0.15-0.2ms. However, both log stream and instrumentation stream collection time is small compared to the stream sampling rate that is one sample per 10 to 20 seconds. The third plot in Figure 16 shows the prediction time for classifying each feature stream tuple using the decision tree, which is very quick (0.02-0.06ms). Finally, the prediction model training time amortized over the number of tuples retained in the training set after reservoir sampling. This includes ten-fold cross-validation; the overhead without cross-validation is ten times less, as expected. Still, even for 1000 training samples with cross-validation, the prediction model training takes only about 220ms (or, 0.2ms per sample). Moreover, the model training is typically triggered relatively infrequently.

5 Conclusion

In this paper, we have presented a new predictive failure management framework for distributed stream processing systems. Different from previous reactive or proactive failure management schemes, our approach provides a *tunable* failure management solution that enables the data stream management system to achieve optimal tradeoff between fault tolerance and resource cost based on user-defined reward functions. We first present the online failure prediction scheme using stream-based decision tree learning methods. We then describe the just-in-time failure prevention scheme that uses failure alerts as guidance to achieve selective failure prevention on abnormal components only. The prevention scheme includes an iterative component inspection algorithm to overcome prediction errors and provide feedback to failure predictors. To reduce online failure learning overhead, we propose adaptive data stream sampling techniques that adjust measurement metric sampling rates based on the states of monitored components, and maintain a limited size of historical training data using reservoir sampling. We have implemented an initial prototype of the predictive failure management framework within IBM System S distributed stream processing system. The experimental results show that our failure prediction schemes can achieve more efficient failure management than conventional reactive and proactive approaches while imposing low overhead to the stream processing system.

References

[1] C4.5 Release 8. *http://www.rulequest.com/Personal/*.

[2] Internet Traffic Archive. *http://ita.ee.lbl.gov/*.

[3] NIST TREC Video Archive. *http://www-nlpir.nist.gov/projects/trecvid/*.

[4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. *Proc. of SIGMOD*, 2005.

[5] P. Barham and et al. Xen and the Art of Virtualization. *Proc. of SOSP*, 2003.

[6] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. *Proc. of OSDI*, December 2004.

[7] G. W. Dunlap, S. T. King, Sukru Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *Proc. of OSDI*, December 2002.

[8] G. W. Dunlap, S. T. King, Sukru Cinar, M. A. Basrai, and P. M. Chen. Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates. *Proc. of SOSP*, October 2005.

[9] K-L. Wu et al. Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. *Proc. of VLDB*, 2007.

[10] X. Gu, S. Papadimitriou, P. S. Yu, and S.-P. Chang. Online Failure Forecast for Fault-Tolerant Data Stream Processing. *Proc. of ICDE*, April 2008.

[11] X. Gu, Z. Wen, C.-Y. Lin, and P. S. Yu. ViCo: An Adaptive Distributed Video Correlation System. *Proc. of ACM Multimedia*, October 2006.

[12] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. *Proc. of ICDE*, 2007.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[14] J. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. *Proc. of ICDE*, 2007.

[15] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High Availability Algorithms for Distributed Stream Processing. *Proc. of ICDE*, 2005.

[16] N. Jain and et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. *Proc. of SIGMOD*, 2006.

[17] R. Jin and G. Agrawal. Efficient decision tree construction on streaming data. In *Proc. of KDD*, 2003.

[18] C. R. Palmer and C. Faloutsos. Density biased sampling: An improved method for data mining and clustering. In *SIGMOD*, 2000.

[19] B. Scholkopf, C.J.C. Burges, and A.J. Smola. *Advances in Kernel Methods: Support Vector Learning*. MIT Press, 1999.

[20] M. Shah, J. Hellerstein, and E. Brewer. Highly-available, fault-tolerant, parallel dataflows. *Proc. of SIGMOD*, 2004.

[21] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and Implementation of Software Rejuvenation in Cluster Systems. *Proc. of SIGMETRICS*, 2004.

[22] J. Scott Vitter. Random sampling with a reservoir. *ACM TOMS*, 11(1), 1985.