LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-464273

# Automatic Fault Characterization via Abnormality-Enhanced Classification

G. Bronevetsky, I. Laguna, B. R. de Supinski

December 20, 2010

Conference on Dependable Systems and Networks
Hong Kong, China
June 27, 2011 through June 27, 2011

## Disclaimer

# Automatic Fault Characterization via Abnormality-Enhanced Classification

Greg Bronevetsky[§], Ignacio Laguna[‡], Bronis R. de Supinski[§]

[§]Lawrence Livermore National Laboratory [‡]Purdue University
{bronevetsky, bronis}@llnl.gov {ilaguna}@purdue.edu

## Abstract

*Enterprise and high-performance computing systems are growing extremely large and complex, employing hundreds to hundreds of thousands of processors and software/hardware stacks built by many people across many organizations [1]. As the growing scale of these machines increases the frequency of faults, system complexity makes these faults difficult to detect and to diagnose. Current system management techniques [2], [3], which focus primarily on efficient data access and query mechanisms, require system administrators to examine the behavior of various system services manually. Growing system complexity is making this manual process unmanageable: administrators require more effective management tools that can detect faults and help to identify their root causes.*

*System administrators need timely notification when a fault is manifested that includes the type of fault, the time period in which it occurred and the processor on which it originated. Statistical modeling approaches can accurately characterize system behavior [4]. However, the complex effects of system faults make these tools difficult to apply effectively. This paper investigates the application of classification and clustering algorithms to fault detection and characterization. We show experimentally that naively applying these methods achieves poor accuracy. Further, we design novel techniques that combine classification algorithms with information on the abnormality of application behavior to improve detection and characterization accuracy. Our experiments demonstrate that these techniques can detect and characterize faults with 65% accuracy, compared to just 5% accuracy for naive approaches.*

## I. Introduction

Global research and commerce require level of computing capabilities that lead to complex systems that have hundreds to hundreds of thousands of cores, gigabytes to terabytes of RAM and software and hardware components from many sources. This vast scale increases the probability that some component will fail and the complexity of the effects of those failures. More importantly, the dependence of businesses on the continued availability of these systems leads to an annual cost of such failures between $22.2 and $59.5 billion [5]. Further, the frequency and complexity of these failures will increase with the demand for systems with even greater capabilities. The costs of these failures will also increase unless we provide system administrators with tools that can quickly detect and help solve failures.

Today's system administration techniques are increasingly inadequate for modern systems. These tools provide vast amounts of data about the systems and mechanisms to search and to filter system logs and health reports from system nodes and resource [2], [3]. However, they require humans to interpret this data to detect faults and provide little insight into their root causes. These limitations arise from the complex ways in which faults affect large systems. They often propagate from one component to another, eventually causing either full failures or more subtle problems like degradations in quality of service or functionality. Overcoming these limitations will require models of system behavior and how faults affect it.

Prior research has developed techniques to generate models of specific systems, either manually or based on an explicit system specification. Although manually generated models can predict the root causes of faults [6], [7], they require significant effort to create for the wide variety of system components and their interactions. Further, components can exhibit complex interactions in which one component influences distant components without affecting intervening components [8]. Alternatively, fully automated techniques infer key system behaviors and how faults affect them based on statistical models and empirical observations [4]. These models can then determine whether the system's behavior is normal or abnormal and identify the source or the nature of the abnormality.

In this paper, we study limitations of naively applying machine learning models to detect and characterize system faults. We design methods that improve model error-detection accuracy by leveraging information about event probabilities. We focus on the most critical capabilities for a fault analysis system: the fault type; the time period in which the fault is manifested; and the system component(s) in which it originated.

Current techniques train a statistical model on sample application runs that exhibit normal behavior as well as runs with one or more types of abnormal behavior. They decompose application runs into individual events of approximately a few seconds or milliseconds so that the

latency of fault diagnosis and resolution can be roughly that long. When presented with a specific execution, the model can then determine whether the execution is more consistent with normal or abnormal behavior and, if abnormal, the class of faulty runs to which it is most similar.

This paper makes two fundamental contributions. First, we show that this intuitive statistical modeling fails for common types of system faults. Second, we overcome this limitation by using event probability information that requires no additional monitoring. Intuitively, the naive application of machine learning classification algorithms cannot detect accurately complex system faults. Consider, for example, a fault that causes a reduction in CPU performance such as a CPU-hang or a change in core frequency. Traditional classification algorithms cannot detect or characterize this fault because it affects software inconsistently. This fault will affect CPU-bound code regions significantly but will have little impact on memory-bound regions. Further, if a misbehaving piece of software is the root cause of the fault, the operating system will schedule this software into discrete time periods, leading to sporadic effects. This fact that many events during faulty execution behave normally can cause traditional techniques, which label all events during the faulty time period as faulty, to train inaccurate models that cannot differentiate between normal and faulty behavior.

Our novel solution can employ traditional statistical classifiers for complex fault detection and analysis. It enhances the quality of the information being classified by building a secondary statistical model that captures the probability that a given event came from a normal or faulty run. We use these probabilities to filter the original labeling presented to the classification algorithms, which focuses their power on the abnormal events. Specifically, it enables the classifier to correctly identify many more faulty events at the cost of a small number of false positive predictions, while reducing the number of false negatives. Figure 1 shows a diagram that compares the naive classification approach with our refined approach.

The naive model, which only classifies individual events, can overwhelm system administrators with many individual reports that correspond to the same fault. Our approach eliminates this problem by clustering fault detections to provide administrators with just one notification for each system fault. In another major result, we show that even a small rate of false fault detections when traditional classifiers correctly differentiate most normal and faulty events can cause clustering to produce incorrect fault predictions. Our technique addresses this problem by focusing attention to fault detections that correspond to very low probability events, which improves accuracy of fault detection from 5% to 65% on faulty runs, while maintaining a 5% false positive rate.

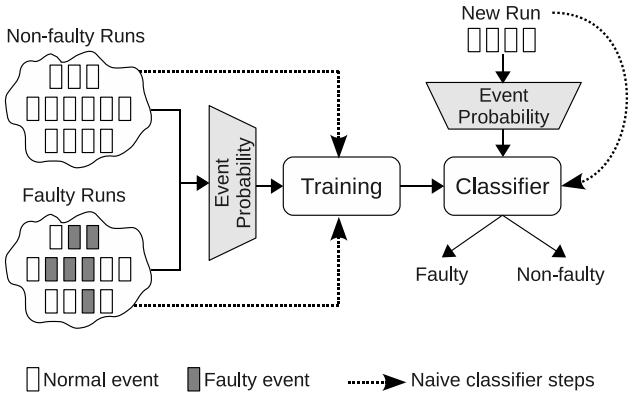



**Figure 1.** Naive- versus new-classification approach.

This paper is organized as follows. Section II presents our experimental set up, describing the applications and statistical methods that our analysis uses our behavior monitoring infrastructure. Section III describes our general approach to model application behavior and shows that the intuitive approach results in very poor accuracy. Section IV then explains the causes of this inaccuracy and shows how to combine event abnormality information with classification algorithms to improve detection accuracy significantly. We examine this approach in detail in Section V to show that information about the distance of an event from its normal behavior is useful for fault characterization. Section VI then shows how to aggregate individual fault predictions into a single statement of the fault's starting and ending times, location and type.

## II. Experimental Setup

For our experimental analyses we have chosen to focus on detecting faults that occur on High-Performance Computing (HPC) systems during the execution of the scientific applications that typically run on these systems. HPC systems are among the largest and most powerful in the world, with the Top 500 most powerful systems capable of sustained 31 to 2,500 TeraFlops of computational power [9]. The largest systems have over 200,000 processors, 200TB of RAM and Petabytes of disk storage. Even though HPC systems are built from high-quality components and use light-weight software stacks, the very large scale of these machines means that they fail very frequently. Major systems like the ASCI Q machine experienced 26.1 CPU failures per week [10], and the 100,000 node BlueGene/L machine at Lawrence Livermore National Laboratory suffers from one L1 cache bit flip every 4 hours. From the perspective of applications, HPC systems fail 10-20 times each day due to failures in system hardware and software [11].

HPC systems are primarily used to run large-scale scientific applications written using the Message Passing

Interface (MPI) programming model, which allows application processes to communicate via operations such as send, receive and broadcast. We modeled the behavior of such applications by breaking them up into individual events and then creating statistical models that predict whether any given event corresponds to a normal or faulty execution. We used the PnMPI [12] tool to monitor application MPI calls. At each MPI operation, we monitor the application's calling stack and the values of the `count` and `datatype` arguments of the MPI routine. The time period between two such points is denoted as an "event" and corresponds to the execution of an MPI function or application-code between two MPI calls. A given combination of call stack and MPI arguments is denoted "event context" and intuitively, events that share the same context will exhibit the same runtime behavior.

We use the NASA Advanced Supercomputing Parallel Benchmarks (NAS)[13] to represent a typical scientific application. These benchmarks consist of 8 parallel application kernels written using MPI Specifically, we focus on the applications BT, CG, LU, MG and SP—we omitted EP, FT and IS because their use of MPI is too simplistic or infrequent to accurately capture their behavior at the granularity of MPI calls. These applications consist of a setup phase, a main compute phase and a shutdown phase. Since only the main compute phase is designed to represent realistic application behavior, our experiments focus on faults that manifests only during this phase. Our experiments were conducted on the Hera cluster at LLNL, which consists of 800 4-socket nodes that are equipped with quad-core 2.3Ghz Opteron 8356 CPUs (10h microarchitecture) and 32GBs of RAM. Each application was executed with 16 processes on an input that results in a 10-60 second execution time (class "A" for BT, LU and SP, class "B" for CG, and MG).

Since real system faults are sufficiently rare that they cannot be used for a large-scale experimental evaluation, we rely on several synthetic faults that model various types of resource exhaustions and slowdowns. Specifically, while the main application is executing, our test harness starts up a single thread on one of the cores being used by the application. This thread repeatedly executes an operation that interferes with the execution of the main computation. Table I shows the types of faults injected in our experiments. These faults represent slowdowns or interference problems that affect different system resources, focusing on CPU, Memory and Socket problems, and capture the major issues of detecting and characterizing system faults.

In the sections below we describe how to train a model that detects faults and characterizes the fault class they belong to: `CPU`, `MEM` or `SOCK`. To this end we will focus on two use-cases. The `KnownFault` use-case represents the situation where administrators can describe previously observed faults in terms of code examples and need help to identify them if they appear again in the future. The `UnknownFault` use-case represents the scenario where administrators describe fault classes using one more representative codes and need the system to detect new faults that are similar to the representatives. The model will be trained on three codes from each class. `CPU` faults will be represented by `CPU_incr`, `CPU_pow` and `CPU_mmm`; `MEM` faults by `MEM_1MB_All`, `MEM_1GB_All` and `MEM_1GB_Walk`; `SOCK` by `SOCK_1KB_1Mesg`, `SOCK_1KB_10Msg` and `SOCK_32KB_10Msg`. To evaluate the model's effectiveness on the `KnownFault` use-case we will use it to detect and characterize `CPU_incr`, `MEM_1GB_All` and `SOCK_1KB_1Mesg` (marked in dark gray in Table I). We will evaluate its effectiveness on `UnknownFault` by using it to analyze `CPU_rank1`, `MEM_1MB_Walk` and `SOCK_1KB_10Mesg` (marked in light gray in Table I). The analysis in Sections III, IV and V will focus on `KnownFault` because the major observations are the same for both use-cases. Section VI, which discusses the design of a real system administration tool will look at the tool's effectiveness on both evaluation sets.

Hardware performance counters are a valuable resource for measuring the behavior of software and hardware. Modern microprocessors provide hundreds of possible performance counters, with the Opteron 10h microarchitecture providing 272 major counters, many of which have multiple options. Unfortunately, the Opteron 10h allows only four of these counters to be monitored simultaneously, making it necessary to carefully choose the counters to be monitored to ensure that the observed counter values properly differentiate the phenomena being studied. Since we had little intuition about which counters would be the best to monitor, we selected the four counters that are most significantly affected by our faults. We do this by observing the values of all counters when a sample application was injected with each individual fault and selecting the counters the values of which differ the most when exposed to the different fault types.

The counters chosen by this method were:

- `INSTRUCTION_FETCH_STALL` - "The number of cycles the instruction fetcher is stalled." [14]
- `X87_FLOPS_RETIRED_MULT` - "The number of multiply operations (uops) dispatched to the FPU execution pipelines." [14]
- `BRANCH_TAKEN_RETIRED` - "The number of taken branches retired." [14]
- `DATA_CACHE_ACCESSES` - "The number of accesses to the data cache for load and store references." [14]

| Fault Type | Fault Variants | | | |
|---|---|---|---|---|
| | Fault variants used for training | | | |
| CPU | incr | pow | mmm | rank1 |
| CPU-intensive workloads | Increment of variable | clib pow function | Dense matrix-matrix multiplication of 100x100 matrixes | Rank-1 update on 100x100 matrix |
| MEM | 1MB_All | 1GB_All | 1MB_Walk | 1GB_Walk |
| Memory-intensive workloads | Random access of 1MB or 1GB memory region | | Random access of 256KB window that is iteratively shifted over a 1MB or 1GB memory region | |
| SOCK | 1KB_1Mesg | 1KB_10Mesg | 32KB_1Mesg | 32KB_10Mesg |
| Socket-intensive workloads | Establishes a socket and connects to it, sends 1 or 10 messages 1KB or 32KB in size, then closes socket | | | |
| | KnownFault use-case | | UnknownFault use-case | |

**Table I.** Types of injected faults

## III. Modeling Approach

Our modeling procedure, illustrated in Figure 1, begins by collecting a set of training and evaluation runs for each application. We used two types of application runs: (1) *non-faulty* runs, where the application is executed free of faults, and (2) *faulty* runs, where we inject training faults listed in Table I during the application execution. For the training set, we used 16 non-faulty runs and 16 faulty runs. During the $i^{th}$ faulty run, we inject the fault into the $i^{th}$ application MPI process to include faults originating in all processes in our training set. In each faulty execution, a fault thread was executed for most of application's main computation loop, ensuring that the model is provided with many faulty events on which to train. Our evaluation set is similar to the training set, except that the faults are drawn from the KnownFault or UnknownFault use-cases and includes additional 40 non-faulty runs.

Given a set of training runs, we analyze the observed events and annotate them as follows:

- NO_FAULT: No injected fault thread was executing during the event
- THIS_PROCESS - CPU/MEM/SOCK: A fault thread of the given type was executing at the same time and on the same process as the event.
- OTHER_PROCESS - CPU/MEM/SOCK: A fault thread of the given type was executing at the same time as the event but on a different process.

We then take a traditional classifier and train it on these events, where each event has the above class label and the following feature set: (i) unique ID of its starting end ending MPI calls, including call stack and arguments, (ii) event's execution time, (iii) values of all 4 performance counters measured from the start to the end of the event. The classifiers used in this study were Random Forest, C4.5 Decision Tree, Logit Boost and Random Committee, using the implementations available in Weka 3.6.2 [15]. Since the number of events generated by 160 training runs was very large (from 7e+5 for MG to 2e+7 for LU), it was infeasible to build classifiers on all the events in all the runs in the training set. As such, we trained classifiers on a randomly-chosen subset of upto 80,000 events, choosing an equal number of events from each training run using a uniform distribution.

Each classifier was executed on the events in the evaluation runs to predict each event's label. These predictions were used to determine when the application's execution was affected by a fault, the fault loctaion as well as the type of fault (CPU, MEM or SOCK. We did this by dividing the application's execution into time windows of 50ms, and using the labels of the events in the window to label the window itself. The label of a given process within a window is NO_FAULT if no faults were predicted. Otherwise, it was the most common fault label among the individual event predictions. The label of an entire window across all processes is NO_FAULT if either no faults were detected on any process or if all predicted fault labels were for OTHER_PROCESS, since in this case the fault was not sufficiently evident for any one process to detect the fault within itself. If one or more processes did detect a fault on themselves (the THIS_PROCESS - FAULT_TYPE label), the time window is assigned the label of the process that is most confident about its labeling, i.e., the largest fraction of its events in the time window have this label.

Figure 2 shows the accuracy of this approach with all the classifiers for the KnownFault use-case (results are very similar for UnknownFault) and Figure 3 shows the accuracy for non-faulty runs. The figures show the fraction of the application's running time where the classifier predicted that a fault was manifested and this prediction was either true or false. A prediction is true if the fault's location type is correctly identified, and it is false otherwise. These correspond in the figure to true positive and false positive predictions. The figure also shows the fraction of time when the classifier predicted that no fault was occurring and this prediction was correct or wrong. These are the true negative and false negative predictions. The first observation is that the Logit Boost classifier is significantly more accurate than the others because it is significantly more careful about labeling an event as a fault. As a result, on non-faulty runs its false positive rate is almost 0%, whereas for other classifiers it ranges between 2% and 40%. Also, while it has a much lower true positive rate for the faulty runs, the false positive rates of the other

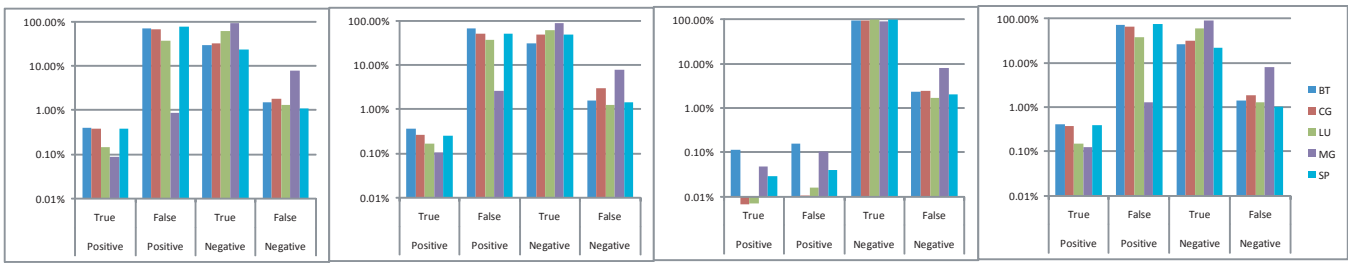



**Figure 2.** Prediction accuracy of classifiers on the faulty runs of the `KnownFault` use-case using naive training

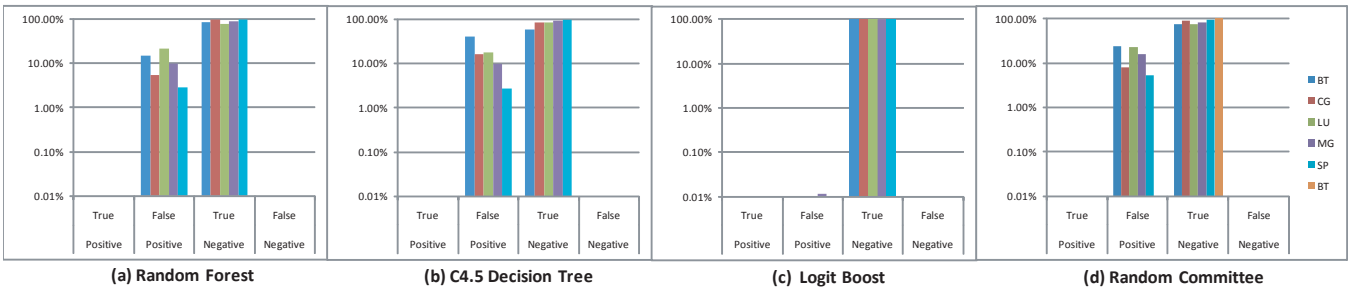



**Figure 3.** Prediction accuracy of classifiers on non-faulty runs using naive training

classifiers are significantly higher than their true positive rates. In contrast, the two rates are similar for Logit Boost, meaning that any tool built using its classifications will not need to filter out a few good predictions from many bad ones. Since Logit Boost performed consistently better for most of our experiments, for the remainder of the paper we focus on models based on this classifier.

Looking at the accuracy of the resulting model, we see that its has a low true positive rate of .11% for BT and less than .01% for CG and LU. To understand the source of this problem we examined the possibility that the model was not being provided with sufficient information about how the application was behaving before each event. To evaluate this possibility, we repeated the above modeling procedure but for each event we also provided the calling context, time and performance counters of the preceding four events. Figure 4(a) shows the accuracy of the resulting model on the `KnownFault` faulty runs and Figure 4(b) on non-faulty runs with the Logit Boost classifier. The data shows that the introduction of additional historical information actually degrades the model's accuracy, dropping the true positive rate to 0%. This means that a lack of history is not the source of the poor accuracy and indeed, that the introduction of additional features that have little impact on application's behavior have a negative impact on accuracy.


**Figure 4.** Prediction accuracy of Logit Boost classifier on `KnownFault` faulty runs and non-faulty runs, using naive training with 5-event history on faulty runs

## IV. Abnormality-Enhanced Classification

The reason for the poor accuracy of the naive approach is that system faults do not affect applications in a consistent. Instead, they only affect a small fraction of them. For example, faults that affect CPU performance will have little effect on memory-intensive code. Further, if the problem

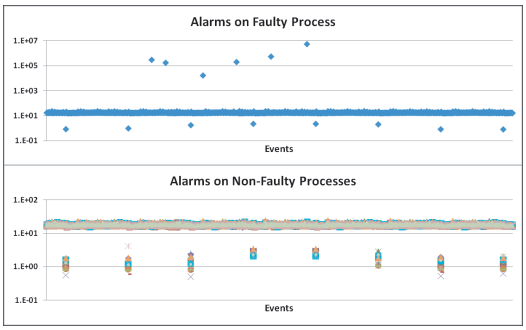



**Figure 5.** Abnormality values of events in a run of `BT` injected with `CPU_incr`

comes from errant software activity, its effects will be even more irregular because the execution of this software will be scheduled into discrete units by the operating system. Figure 5 illustrates this effect as exhibited in a run of the `BT` application when injected with the CPU-intensive thread. The horizontal axis shows individual events and the vertical axis shows a measure of how significantly the event's behavior is affected by the fault (i.e., *abnormality values*, which are computed as shown below). The figure shows that most events during the non-faulty time period behaved normally (with abnormalities on the order of 1E+1), with a few outliers. Further, it shows that when the fault was activated, most events still behave normally, with a few events that were affected very significantly (with abnormalities on the order of 1E+6).

A problem with the basic modeling approach is that while the effect of faults is irregular, this model assumes the exact opposite, labeling as faulty all events in the faulty-run. This presents the classification algorithm with a very noisy training set, forcing it to differentiate two sets of events:

- "Normal" events - almost all (e.g. 99.9%) behave normally with a few outliers, and
- "Faulty" events - most (e.g. 90%) behave normally and some behave abnormally.

Since the behavior of the events in the two sets overlaps significantly, most classifiers cannot easily differentiate them. This suggests that the key to creating a more accurate model is to pre-process (or filter) the training data provided to classifiers to focus them on just truly abnormal events.

To evaluate this hypothesis, we create a probability distribution that describes the behavior of the events in each context. The distributions are estimated using the events in the non-faulty runs in our training set. For each event context, we compute five probability distributions, one for each quantity observed by our monitor in each context: elapsed time and four performance counters. Distributions capture the normal behavior of application events and make it possible to measure the deviation of an event from its normal behavior as the event's probability given the distributions of any of its observed quantities.

Since we don't know in advance what probability distribution would be the best fit for the data, we use non-parametric density estimation to build the distributions. In particular we used normalized histograms to represent the distributions. Since in our case the range of possible values was unknown, we used an adaptive histogram algorithm that chooses the locations and sizes of the histogram value ranges to fit the observed data. The algorithm works as illustrated in Figure 6. When a new data value is added to the histogram, is it assigned to its own value range. When the number of ranges in the histogram rises above a predefined bound, the two nearest ranges are merged. Finally, we turned these histograms into continuous probability distributions by (i) connecting adjacent buckets with lines, (ii) attaching the upper and lower halves of a Gaussian distribution to the largest and smallest bucket, respectively, to model the probability of data points above and below the observed data region and (iii) normalizing the entire region to ensure that the sum under the curve is equal to 1. In our experiments we used histograms with 30 buckets since in practice this provides sufficient resolution to ensure good model accuracy.

While the resulting distributions accurately capture the distributions of observed time and counter values, they also include some noise in the form of outlier values that are poor representatives of the application's normal behavior. We minimize this effect by removing the 10% largest and smallest values seen of each observation at each event context.

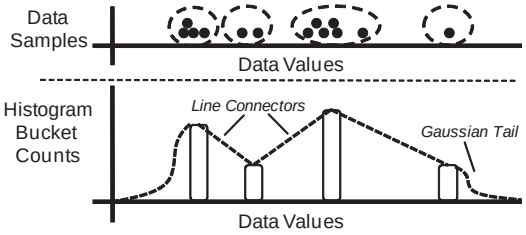



**Figure 6.** Example histogram

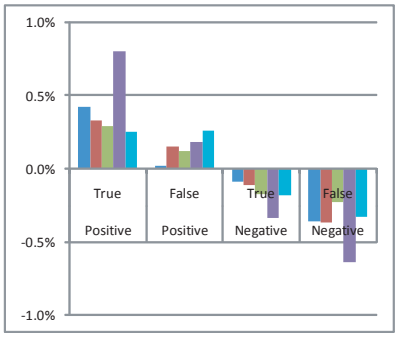
We use the probability distributions as a filter on the training sets provided to the classification algorithms in the procedure described in Section III. For each event from a faulty run we compute the probability of each of its observed quantity (time and performance counters). We then compute the event's *abnormality value* as the negative logarithm of its probability. We use this measure because the logarithm function helps classifiers differentiate between the small probability values of abnormal events. This is useful if abnormality values are used as features. Further, it is negated so that unlikely events have larger values, which is more intuitive for our fault detection use-case. We compute the abnormality value of

the entire event as the Euclidean average of the abnormality values of each observed quantity ($\sqrt{\frac{\sum_{i=1}^{n} abnorm_i^2}{n}}$ for $n$ observed quantities, where $abnorm_i$ is the abnormality value of quantity $i$). Events that occur while the fault thread is executing and have abnormality values above a given threshold are labeled with the fault type and location, as described in Section III. Events that occur while there is no fault or those that have abnormality values below the threshold are labeled as non-faulty. the threshold is equal to the maximum abnormality values ever observed, plus three standard deviations to make sure that normal events are never confused with abnormal ones. Finally, while we trained our original model on a randomly chosen subset of upto 80,000 events, in this adjusted model we must ensure that the classifier is provided is many examples of both normal an abnormal events. We use the same procedure to choose upto 40,000 normal events and upto 15,000 abnormal events to train the classifier. The number of events used in this experiment was smaller than the number used for the original model to show that the number of events is not important to the model's ultimate accuracy.

Figure 7(a) shows the result of using event probabilities to filter the training set for the `KnownFault` faulty runs, using Logit Boost classifier (results for `UnknownFault` are very similar). Specifically, it shows the absolute difference between the true/false positive/negative rates of the original predictor (labeled ``Plain Classifier'') and the new predictor with abnormality filtering (labeled ``Abnorm Filtered''). The difference is `Abnorm Filtered - Plain Classifier`. The data shows that the use of abnormality filtering increases the number of positive detections, with many more true positives than false positives. The same is observed with negative detections, where the the number of false negatives is reduced more than the true negatives. Note, however that the number of true positive predictions is still not significantly higher than the number of false positive predictions and is much lower than the number of false negative predictions:

| | BT | CG | LU | MG | SP |
|---|---|---|---|---|---|
| $\frac{\text{true positives}}{\text{false positives}}$ | 2.99 | 2.09 | 2.17 | 2.97 | 0.92 |
| $\frac{\text{true positives}}{\text{false negatives}}$ | 0.27 | 0.16 | 0.2 | 0.12 | 0.16 |

The large number of false negatives relative to true positives occurs because the model is trained to differentiate normal events from different types of abnormal events. As such, even when a fault thread is executing it will only label the abnormal events as faulty, classifying the remaining events as non-faulty even though they are surrounded by many faulty events. This means that the key to detecting faulty regions of application execution is to check whether fault predictions are common. Further, if we look at the actual positive predictions we find that the true positives consistently predict the same fault type and location whereas false positives are more erratic, making a variety of different predictions. Both these properties is used in Section VI to detect faulty periods of application execution by looking at sequences of frequent identical fault predictions.




(a) KnownFault Runs


(b) Non-Faulty Runs

**Figure 7.** Prediction accuracy of Logit Boost classifier using abnormality-enhanced training on `KnownFault` faulty runs

Figure 7(b) shows the filtered model's accuracy for non-faulty runs relative to the plain classifier. It hows that although filtering significantly improves fault detection rates, it also has a side-effect of increasing the false positive rate by 0% - .35%. As discussed in Section VI this causes tools based on this model to have some false positives but overall does not significantly degrade this technique's utility for creating a useful system administration tool.

## V. Using Abnormality as a Feature

Having established that event probabilities can be used to improve model accuracy by filtering the training set, another key question to answer is whether these probabilities can be used as features to further improve the accuracy of the model. We explored this by looking at two sets of additional features.

First, we incorporated the abnormality value of each observed quantity (elapsed time and performance counters) of an event as additional classification features. The abnormality values were computed as above, with respect to the probability distributions from the non-faulty runs. The new

training set thus contained almost twice as many features as the training set evaluated in Section IV. For each event we used the event context as well as two numbers for each observable quantity: its observed value and its abnormality value. In subsequent figures this type of training is denoted ``Abnorm Feature''. Figures 8 (a) and 9 (a) show the accuracy of these classifiers using the Logit Boost classifier on the KnownFault faulty runs (very similar results for UnknownFault) and non-faulty runs, respectively. These figures show the difference between the accuracy of the two techniques (Abnorm Feature - Abnorm Filter). This data shows that the use of abnormality as a feature adds little to the model's accuracy. However, as shown in Section VI, these small differences can be important when trying to detect the faulty time period from a relatively small number of fault detections. For instance, in the KnownFault experiments observe that Abnorm Feature's number of true negatives for CG rises and true and false negative counts fall. Meanwhile, for MG it sees a rise in the number of true positives, a smaller rise for false positives, and a drop in false negatives. The increased number of true positive and negative detections will make easier to identify time periods as being faulty. Meanwhile, note that the dominant effect of Abnorm Feature on SP is a rise in the number of false positives, which is much larger than the rise in true positives. This will serve to confuse the signal, making fault detection less precise.

To better understand the value of probabilities we examined the effect of using all available probability information as classifier features. To this end we took all the abnormal events that occurred during each faulty training run and computed a probability distribution from them, following the algorithm described in Section IV. We used just the abnormal events rather than all events because these events represent the particular ways in which each type of fault affects the application's execution. We then used them in two ways to build a training set for our classifiers. First, we extended the above training set by providing for each event the probability of each observed quantity with respect to the each fault type's probability distribution. Each event was thus described by its (i) type, (ii) observed quantities, (iii) their abnormality values and (iv) their probabilities with respect to fault distributions. This was termed ``Fault Prob Feature'' training. Further, to understand the relative value of probabilities vs observable quantities, we also created a training set that omits observable quantities, providing just event types, the abnormality values and probabilities with respect to fault distributions, denoting this ``Only Fault Prob Feature''. Note that since we built distributions of faulty runs from only abnormal events, these distributions correspond to far fewer events than the distributions of non-faulty runs. If we did not observe any events for a given event context, the probability of observables of all events within this context was set to a dummy constant.

Figures 8(b,c) and 9(b,c) shows the results of training the Logit Boost classifier using the modified training sets. Figure 8 shows their accuracy on KnownFault faulty runs (results for UnknownFault are similar) and Figure 9 shows results for non-faulty runs. The introduction of new features (Fault Prob Feature) results in a somewhat lower accuracy for faulty runs as well as additional false positives on non-faulty runs. However, it is not clear whether the reduced accuracy is caused by probabilities being a poor data source or by the noise introduced by constructing probability distributions from a small number of events. Further, the data also shows that removing the observable quantities from the training set (Just Fault Prob Feature) further degrades model quality. This suggests that abnormality values do not carry as much information as the original observables about an event's behavior.

Note that while the absolute differences in true/false positive/negative counts differ only percent, this may be a large amount compared to the total number of faulty events during the application's execution. As shown in Section VI, these differences in event detections have a large effect on the accuracy of tools built from these models.

## VI. Concrete Fault Predictions via Event Clustering

Fault detection and characterization at the level of individual events or 50ms time windows is interesting but on its own it is not useful as a system administration tool because it produces output that is too fine-grained and verbose for human consumption. A real administration tool must synthesize the individual fault classifications into a report provided to an administrator that concisely summarizes the time period when the fault occurred, the portion of the system affected by the fault and the fault's type. The final section of this paper examines ways to build such a tool and how the use of probability information affects the tools accuracy.

The classification algorithm described above labels individual events and time periods with the type of fault that was experienced at that time, if any. As discussed above, the effects of faults are erratic, with some events behaving abnormally and most events behaving normally. As such, to succinctly summarize the fault's time, location and type we must identify the region of time that has a high density of identical fault predictions. To this end we employed a simple one-pass algorithm that identifies a time period as faulty if

- All the time windows inside it are labeled with the same fault type and faulty process or have the NO_FAULT label, and

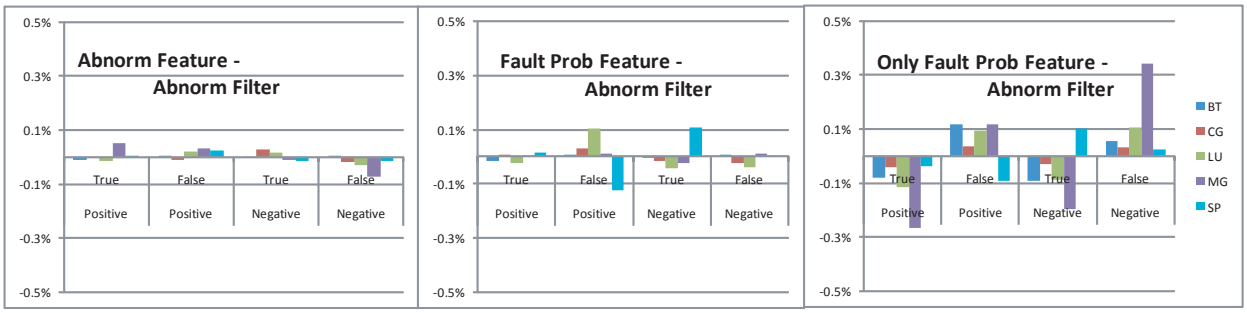



**Figure 8.** Difference in prediction accuracy on `KnownFault` faulty runs using abnormality as a feature, relative to `Abnorm Filter`

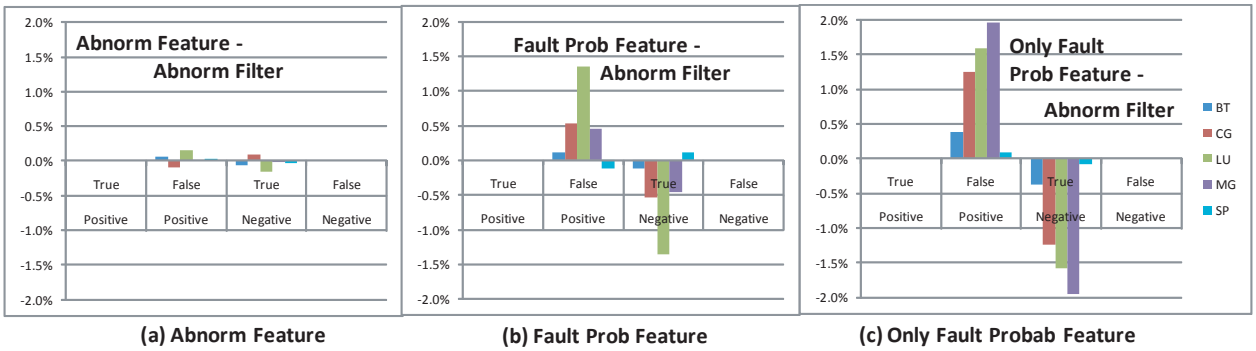



**Figure 9.** Difference in accuracy on non-faulty runs using abnormality as a feature, relative to `Abnorm Filter`

- There are at least two windows inside the time period that have fault labels, and
- The first and last time window have fault labels, and
- All adjacent time windows with a fault label are no more than a given amount of time $\tau$ away from each other.

The parameter $\tau$ controls the fault density of the desired clusters and was set to 500ms in our experiments. While we also experimented with more complex clustering algorithms that use elaborate distance metrics and free parameters, they do not perform better than the above algorithm and their notions of density are less intuitive. Further, since this algorithm can be implemented to use one pass, it is well-suited for low-latency online fault detection.

Figure 10 shows the accuracy of fault prediction on the `KnownFault` faulty runs and non-faulty runs using the above algorithm and the Logit Boost classifier that was trained using the `Plain Classifier`, `Hist Classifier` and `Abnorm Filtered` algorithms. Here successful fault detection is defined as producing (i) a single time period labeled with the correct fault type and fault process that (ii) overlaps with the time when the injected fault was actually executing, and (iii) the predicted duration of the fault is no more than twice as long as the actual fault. This definition ensures that the predicted fault both points to the type and location of fault as well as a short time period where the fault occurred. The data shows that when the classifier is trained on the `Plain Classifier` training set the clustering algorithm can correctly predict some faults, reaching 20% accuracy on BT but much lower on the other applications. Meanwhile, `Hist Classifier`'s 0% rate of positive event detections results in 0% overall fault detection accuracy. `Abnorm Filtered` features significantly better accuracy that ranges from 56% for CG to 84% for BT.

Meanwhile, looking at the non-faulty runs, `Plain Classifier` and `Hist Classifier` have no false positives. `Plain Classifier` has no false positives because its predictions are so erratic, it is very unlikely to get two identical predictions that the clustering algorithm can use to make a fault prediction. In contrast, `Hist Classifier` has no false positives simply because it makes no positive fault predictions. `Abnorm Filtered`'s fault detection is more aggressive and accurate and this results in it detecting faults during a normal execution. Although the clustering algorithm guards against erroneous fault detections by requiring that the same fault is detected in at least two nearby time windows, real behavioral abnormalities that were not part of the injected fault can generate such a consistent detection if they are similar to the injected fault types. This suggests the choice of fault examples on which a model should be trained is very complex and in the future we will work on

ways to more clearly separate normal application variation from truly faulty behavior.

Figure 12 presents the difference between the accuracy of clustering based on `Abnorm Feature`, `Fault Dist Feature` and `Only Fault Dist Feature` and `Abnorm Filtered` on both `KnownFaults` faulty runs and non-faulty runs. On faulty runs `Abnorm Feature` is competitive with `Abnorm Filtered`, producing similar fault detection rates. Further, it out-performs `Abnorm Feature` on non-faulty runs, reducing the false positive rate in BT from 12.5% to 10%, in CG from 5% to 0%, in LU from 10% to 2.5% and in MG from 10% to 2.5%. However, on SP the false positive rate of `Abnorm Feature` rises from 2.5% to 12.5%. These results are consistent with the per-event accuracy analyses presented in Section V and are explained in differences in number of true and false predictions of individual events.

Similarly, the poorer per-event accuracy of `Fault Dist Feature` and `Only Fault Dist Feature` results in poorer accuracy of full fault detection based on these models, with false positive rates 2.5% to 50% higher than for `Abnorm Filtered`.

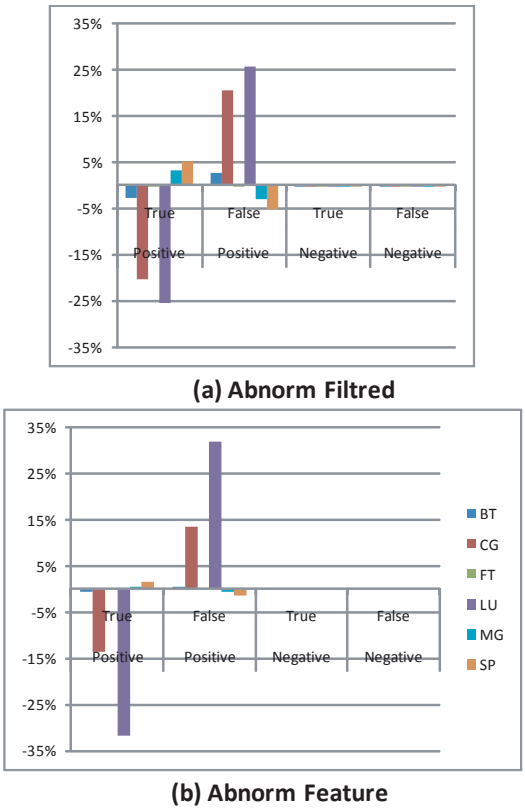



**Figure 11.** Difference between fault detection and classification accuracy for `KnownFault` and `UnknownFault`

Looking more broadly at the applicability of this model and clustering algorithm for system administration tasks, Figure 11 shows the absolute difference between the accuracy for the `KnownFault` and `UnknownFault` use-cases (`UnknownFault` - `KnownFault`). The data shows that when detecting a fault that the model was not explicitly trained to detect, the tool has lower accuracy. Detection is 5%-30% lower for BT, CG, LU and MG but it is actually 10% better for SP. This suggests that proper use of such a tool requires a good understanding of the types of faults that are important to administrators so that appropriate fault representatives can be created.

Our conclusions are summarized in Figure 13, which presents the accuracy of fault detection in three scenarios: (i) detecting faults that were anticipated by system administrators (`KnownFault` runs), (ii) not anticipated by administrators (`UnknownFault` runs) and (iii) for not detecting faults in non-faulty runs. The Logit Boost classification algorithm was used for these results. Abnormality filtering has a clear effect on the accuracy of fault detection, improving it from 5%-10% with the `Plain Classifier` to 55%-65% with the `Abnorm Filtered Classifier`, averaged across all our applications. Further, the use of abnormalities as features for the classifier helps reduce the rate of false positive detections during non-faulty runs from 6% to 5%. Finally, our experiments show that the use of event probabilities with respect to faulty does not contribute the accurate fault detection, resulting in reduced detection accuracy and a higher rate of false detections.

## VII. Related Work

Automatic fault detection can be broadly grouped under two categories according to the way system events are classified as normal or abnormal: (1) the *probability-based-classifier* (or generative) approach, in which a probability model is used to classify events on application executions by evaluating their probability value. The model used is typically a probability distribution that can be estimated using parametric and non-parametric methods. In [16] authors build a TAN model to capture correlations of system metrics with the goal of predicting whether new metrics belong to SLA violations or not. The TAN model allows them to get probability values of multiple metrics more efficiently than in a Bayesian network. In [17] authors use a mixture of Gaussian distributions to capture the probability of performance metrics. Approaches like [8] use histograms as probability distributions, along with the KL norm, to characterize deviations from normal behavior. The use of Bayesian classifiers along with mixture models to predict disk drives failures is explored in [18]; (2) the *traditional-classifier* (or discriminative) approach, in which a classifier is used to determine whether events are normal or abnormal. This approach typically does not rely on a probability distribution to make decisions but in discriminative machine-learning classifiers like decision

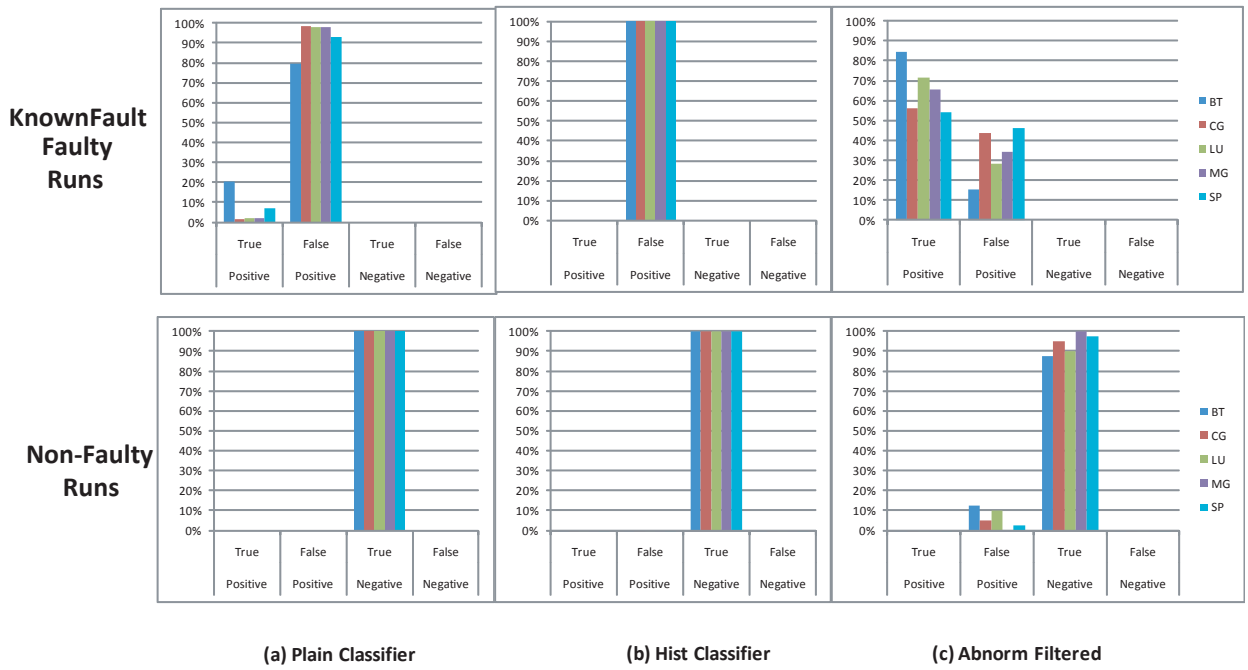



**Figure 10.** Fault detection and classification accuracy of clustering on `KnownFault` faulty runs

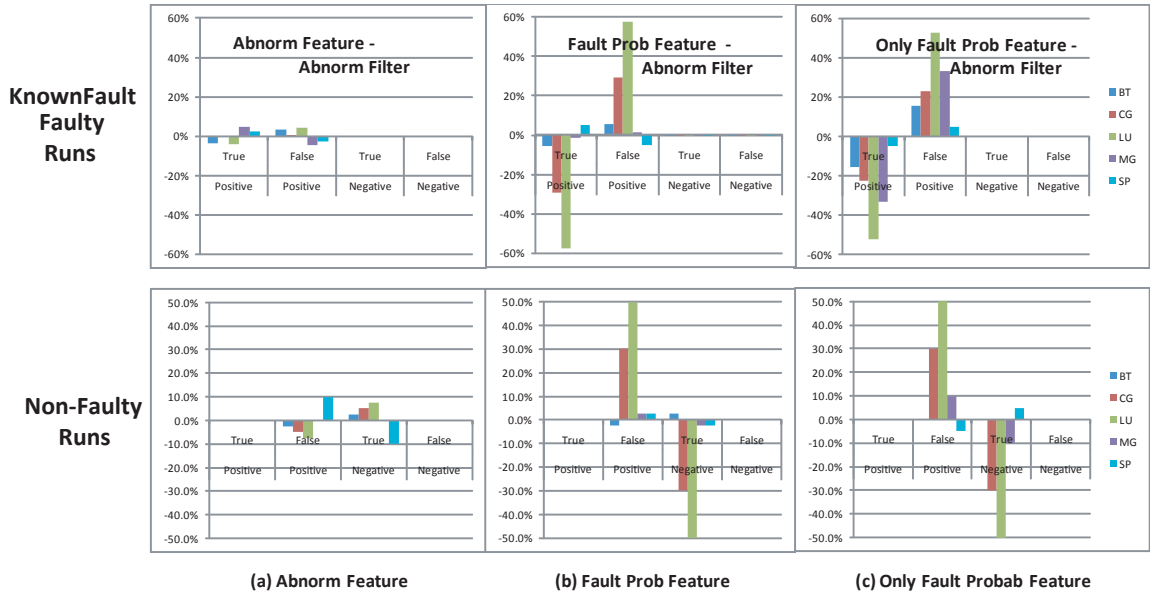



**Figure 12.** Difference in fault detection accuracy on non-faulty runs relative to `Abnorm Filtered`

trees, neural networks, support vector machines, and unsupervised methods like clustering. In [19] a probabilistic context-free grammar (PCFG) is used to identify abnormal web requests in large e-commerce systems. The grammar is trained using faulty and non-faulty runs (which are generated by fault injection). In [20] clustering of application traces is used to detect performance anomalies by looking at clusters with small number of elements (abnormal clusters). In [21] authors use a Markov model to identify abnormal changes of system metrics correlations.

A contribution of our paper is the use of abnormality information (a generative approach) as a filter for the traditional-classifier approach in order to make it more accurate in detecting common faults. To the best of our knowledge, no previous work has been devoted to study this hybrid technique.

## VIII. Summary

This paper examines the the problem of detecting, localizing and characterizing system faults using statistical modeling of application and system behavior. It (i) consid-

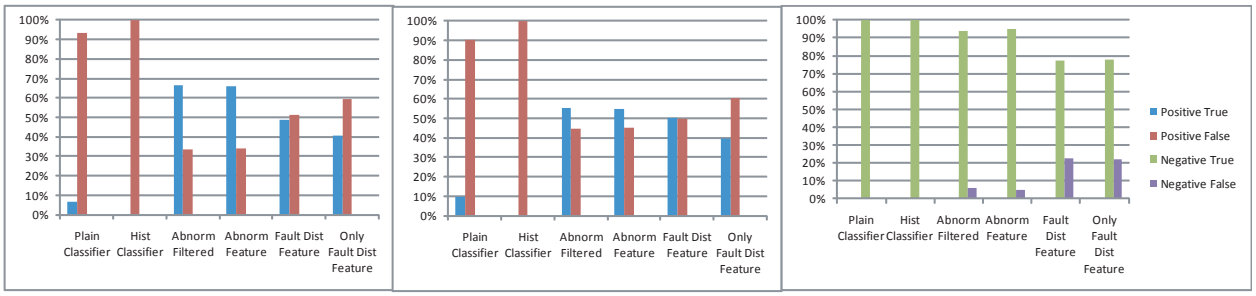



**Figure 13.** Summary of fault detection accuracy results for all model types and sets of evaluation runs

ers an intuitive application of machine learning techniques to this problem, (ii) experimentally shows the limitations of this approach and (iii) identifies how it can be improved significantly via the use of event abnormality information. Our approach builds a secondary probabilistic model to both filter the training set provided to the primary model and to provide the model with additional features. It is built on the observation that system faults have a very inconsistent effect on application behavior, strongly affecting some application regions, while leaving most to execute normally. We have experimentally shown that it is possible to significantly improve the quality of the training set by computing the probability that a given event will appear in a non-faulty execution and only labeling truly abnormal events as faulty. We have further demonstrated that the false positive rate of fault detection can be improved if these measures of abnormality are also used as additional features for the classifier. Finally, our experiments indicate that while the probability of events with respect to normal executions is useful for fault detection, probabilities with respect to faulty execution are not useful for fault detection.

## References


[1] L. A. Barroso and U. Hlzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[2] J. Stearley, S. Corwell, and K. Lord, "Bridging the Gaps: Joining Information Sources with Splunk," in *Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML)*, 2010.

[3] "The syslog protocol," IETF Network Working Group, Tech. Rep. RFC 5424, Mar. 2009.

[4] F. Salfner, M. Lenk, and M. Malek, "A Survey of Online Failure Prediction Methods," *ACM Computing Surveys*, vol. 42, no. 3, 2010.

[5] "The Economic Impacts of Inadequate Infrastructure for Software Testing," National Institute of Standards and Technology, Tech. Rep. 02-03, May 2002.

[6] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie, "A Survey of Online Failure Prediction Methods," *IEEE Communications*, vol. 34, no. 5, pp. 82–90, May 1996.

[7] "Smarts Family - IT Operations Intelligence," http://www.emc.com/products/family/smarts-family.htm.

[8] A. J. Oliner, A. V. Kulkarni, and A. Aiken, "Using Correlated Surprise to Infer Shared Influence," in *International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 191 – 200.

[9] "Top 500 Supercomputer Sites," http://www.top500.org.

[10] S. E. Michalak, K. W. H. abd N. W. Hengartner, and B. E. T. abd S. A. Wender, "Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratory's ASC Q Supercomputer," *IEEE Transactions on In Device and Materials Reliability*, vol. 5, no. 3, pp. 329–335, 2005.

[11] J. Daly, "Big Science Meets the Bathtub Curve." Roadrunner Open Science Presentation, 2008.

[12] M. Schulz and B. R. de Supinski, "PñMPI Tools: A Whole Lot Greater Than the Sum of Their Parts," in *ACM/IEEE Supercomputing Conference*, 2007.

[13] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," NASA Ames Research Center, Tech. Rep. RNR-94-007, Mar. 1994.

[14] "BIOS and Kernel Developers Guide (BKDG) For AMD Family 10h Processors," Advanced Micro Devices Corporation, Tech. Rep. 31116 Rev 3.48, Apr. 2010.

[15] S. R. Garner, "WEKA: The Waikato Environment for Knowledge Analysis," in *In Proc. of the New Zealand Computer Science Research Students Conference*, 1995, pp. 57–64.

[16] 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," in *OSDI '04*. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16.

[17] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira, "Tracking probabilistic correlation of monitoring data for fault detection in complex systems," in *DSN '06*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 259–268.

[18] G. Hamerly and C. Elkan, "Bayesian approaches to failure prediction for disk drives," in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML '01, San Francisco, CA, USA, 2001, pp. 202–209.

[19] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, "Path-based failure and evolution management." in *NSDI'04*, 2004, pp. 309–322.

[20] K. Ozonat, "An information-theoretic approach to detecting performance anomalies and changes for large-scale distributed web services," in *DSN '08*. Anchorage, Alaska, USA: IEEE, 2008, pp. 522–531.

[21] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 623–630.



This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.