Automatic Fault Characterization via Abnormality-Enhanced Classification

Abstract

Enterprise and high-performance computing systems are growing extremely large and complex, employing many processors and diverse software/hardware stacks [1]. As these machines grow in scale, faults become more frequent and system complexity makes it difficult to detect and diagnose them. The difficulty is particularly large for faults that degrade system performance or cause erratic behavior but do not cause outright crashes. The cost of these errors is high since they significantly reduce system productivity, both initially and by time required to resolve them. Current system management techniques [2], [3] do not work well since they require manual examination of system behavior and do not identify root causes.

When a fault is manifested, system administrators need timely notification about the type of fault, the time period in which it occurred and the processor on which it originated. Statistical modeling approaches can accurately characterize normal and abnormal system behavior [4]. However, the complex effects of system faults are less amenable to these techniques. This paper demonstrates that the complexity of system faults makes traditional classification and clustering algorithms inadequate for characterizing them. We design novel techniques that combine classification algorithms with information on the abnormality of application behavior to improve detection and characterization accuracy significantly. Our experiments demonstrate that our techniques can detect and characterize faults with 85% accuracy, compared to just 12% accuracy for direct applications of traditional techniques.

***Keywords*-fault detection, root cause analysis, statistical modeling, autonomic management**

I. Introduction

Global computing demands lead to complex systems with up to hundreds of thousands of cores, terabytes of RAM and diverse software and hardware components. This vast scale increases the probability of component failure. The complex interactions between the components increases the likelihood that single component failure leads to cascaded failures. Overall, the estimates of economic loss in the US due to faulty software are just under 1% of the national GDP [5]. Further, their frequency and complexity will increase with required increases in system capabilities. The costs of these failures will also increase unless we develop tools that can quickly detect problems and localize their root causes.

Increasing system complexity is making today's system administration tools inadequate. 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 resources [2], [3]. However, they require humans to interpret this data to detect errors and they provide little insight into their root causes. These limitations arise from the complex impact of faults, which propagate across components until they cause subtle problems such as service degradation. To overcome this challenge, we need models of system behavior that accurately characterize how this behavior deviates when faults occur.

Prior research has developed techniques to generate models of specific systems based on manual system specifications. Although such models can predict the root causes of faults [6], [7], they are labor intensive and may miss complex interactions in which one component influences distant components without affecting intervening ones [8]. Alternatively, fully automated techniques infer the impact of faults on key system behaviors based on statistical models and empirical observations [4]. These models can, in the best case, classify the system's behavior as normal or abnormal and identify the source of the abnormality.

In this paper, we make two fundamental contributions. First, we study the limitations of baseline machine learning models to detect and to characterize system faults. We show that intuitively applying supervised statistical models to complex system faults fails to characterize fault type, time, and location accurately. *Second*, we design a method that improves model accuracy by combining traditional supervised classifiers with an unsupervised model based on event probabilities that computes the abnormality of individual events. We focus on the most critical capabilities for fault analysis: identification of the type of fault, the time period when the fault is manifested and the system component(s) in which it originates. *We target detection and characterization of faults that cause performance degradations, which are more complex than easy-to-detect fail-stop failures.* These faults include errant OS daemons or worker threads, poor network performance due to unexpected congestion or cable degradation, and resource exhaustion (e.g. excessive paging and cache churn).

This rest of the paper is organized as follows. Section II contrasts our work with traditional statistical machine learning-based failure detection. Section III describes the applications and statistical methods that our behavior monitoring infrastructure uses. Section IV presents our general approach to model application behavior and shows that the intuitive approach achieves poor accuracy. Section V shows how combining event abnormality information with classification algorithms can significantly improve detection accuracy while Section VI explores the details of how abnormality information should be incorporated into

Regular Paper: Cleared through author affiliations, ∼9000 words

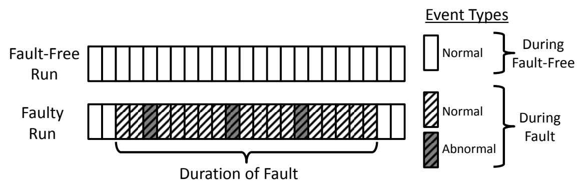

Figure 1. During a system fault events are usually normal and occasionally abnormal

a model to achieve accurate fault characterization. Section VII then shows how to aggregate individual fault predictions to identify the fault's location, type, and starting and ending times.

II. State of the Art and Our Approach

Current techniques create a statistical model for an application through a training phase during which the application is executed and generates a set of observations, or *events*, such as, which component communicates with which other component, or the amount of time a function takes. To build its model, a supervised algorithm uses faulty/nonfaulty labeling of the training observations as well as context information such as the code region that was being executed and the type of fault. In contrast, unsupervised machine learning does not require labeled observations. Instead, these techniques use a priori assumptions about the structure of the data (e.g., non-faulty observations are common and similar to each other, while faulty observations are abnormal and rare) to build the model. Either kind of training leads to a classification algorithm (e.g. Naïve Bayes or Decision Trees) that uses the generated model to identify individual events and production application runs as faulty or non-faulty and even to localize the fault's source and type.

Traditional techniques fail because system faults affect different code regions within the same application differently. While some regions may be significantly affected, others may behave normally. For example, suppose that a chip overheats due to an internal defect or an unusually hot machine room. Modern processors react to such events by reducing the chip's operating frequency. While this change affects CPU-intensive code regions, it has less impact on memory-intensive regions. Similarly, suppose the problem arises due to a problematic interaction between software components. For example, an errant daemon or a utility thread may degrade the application's performance by polluting the cache whenever it is scheduled. Because operating and runtime systems schedule software in coarse time slices (in Linux this is typically 100 ms), the cache pollution only impacts the code regions executed soon after the errant software component runs.

The above effects, illustrated in Figure 1, complicate differentiation of normal and abnormal behavior and identification of the type of fault. To train a classification algorithm, we must label as faulty all observations from when the fault is occurring. However, if most code regions during this time period behave normally then the statistical algorithm has insufficient information to differentiate (mostly normal) faulty behavior from (completely normal) non-faulty behavior. While observation data includes information on the type of fault and executed code regions, it does not characterize the vulnerability of a given execution of a code region to that fault. The lack of key context information makes the classification problem challenging. We use this insight to design a new classification algorithm that recovers this missing information to classify correctly the type of fault based on its effect on application behavior, *without requiring any additional information about the application, the system or the hardware.*

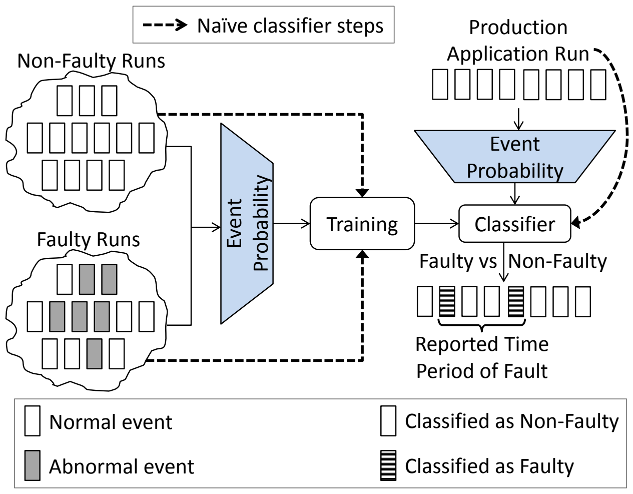

Figure 2. Naïve versus New classification approach.

Our novel solution builds a secondary unsupervised model that captures the probability distribution of each code region's normal behavior, making it possible to identify the events when it behaves abnormally. We use this abnormality information to filter the labeling presented to the classification algorithm to focus it on the abnormal observations. Thus, the classifier can correctly identify many more faulty events with fewer false negatives at the cost of a few false positive predictions. Figure 2 shows a diagram that compares the naïve classification approach to our refined approach.

The naïve approach provides a classification algorithm with execution data that labels each event with the type and location of any fault occurring at the time, if any. The approach then applies the classifier to events from a new execution to label events. Labels that indicate faults, including the fault type and location, are presented to the administrator for further review. This approach has low accuracy and frequently reports many events that correspond to the same fault, which can overwhelm system administrators.

In contrast, our approach first applies an unsupervised model to identify the events that are abnormal and removes the fault labels from all other events. It then presents this filtered training set to the classification algorithm, which results in improved accuracy. Finally, our approach aggregates fault detections to provide just one notification for each system fault, thus reducing the reporting to system administrators. By focusing the classifier's attention on the low probability events, our approach improves the fault detection and classification accuracy from 12% to 85% on faulty runs, while maintaining a 5% false positive rate.

III. Experimental Setup

We focus on detecting faults in High-Performance Computing (HPC) systems during the execution of scientific applications. HPC systems are large: Top 500 systems can sustain 96 to 10,000 TeraFlops of computational power [9]. Even though these systems use high-quality components, they fail regularly due to their large scale and the complex interactions between components. For example, 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 averages 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 primarily run large-scale scientific applications that use MPI (Message Passing Interface). We use P^{n}MPI [12] to capture the calling stack, time and performance counters at the start and end of each MPI operation. Individual MPI operations and code between adjacent operations are thus denoted "events". Further, we denote each observed call stack and operation arguments as an *event context* and model all events with the same context. Intuitively, these events exhibit similar runtime behavior. Our experiments explore various techniques to model these behaviors to detect and to localize faults.

We use the NASA Advanced Supercomputing (NAS) Parallel Benchmarks [13] to represent typical scientific applications. Of the 8 benchmarks in the suite, we focus on BT, CG, LU, MG and SP; we omit EP, FT and IS because their use of MPI is too simplistic or infrequent to capture their behavior accurately at the granularity of MPI calls. These applications have setup, main computation and shutdown phases. Since only the main computation phases represent long running application behavior, we focus on faults that manifest during those phases. We conduct our experiments on 4-socket quad-core Opteron 8356 nodes (10h microarchitecture) that have 32GBs of RAM. We execute each application 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). Each machine has 16 cores so we run all all processes on a single node (one process per core).

Since real system faults are rare, we rely on several synthetic faults that model resource exhaustions and slowdowns. Our test harness starts a thread that interferes with the concurrent execution of the main computation. Table I shows the types of faults that we inject. These faults represent slowdowns or interference problems that affect different system resources on HPC nodes, focusing on CPU, memory and socket problems. Our fault injector thread repeatedly executes the code that Figure 3 shows. We do not consider disk faults since nodes in most large HPC systems do not have disks due to power and reliability concerns. Also, we do not consider soft faults (e.g., erroneous computations and data) because they affect application values and, thus, require different detection strategies.

The following sections describe how to train a model to detect faults and to characterize their fault class: `CPU`, `MEM` or `SOCK`. We have two use cases. `KnownFault` represents situations in which administrators must detect recurrences of previously observed faults for which they have code examples. The `UnknownFault` use case represents situations in which we must detect new faults that are similar but not identical to the example faults. We train the model on three emulators for each class. `CPU_incr`, `CPU_powlog` and `CPU_mmm` represent `CPU` faults; `MEM_1MB_All`, `MEM_1GB_All` and `MEM_1GB_Walk` represent `MEM` faults; while `SOCK_1KB_1Mesg`, `SOCK_1KB_10Msg` and `SOCK_32KB_10Msg` represent `SOCK` faults. We use the model to detect and to characterize faults that affect `CPU_incr`, `MEM_1GB_All` and `SOCK_1KB_1Mesg` (marked in light gray in Table I) to evaluate the its effectiveness for the `KnownFault` use case. We evaluate its effectiveness on `UnknownFault` by analyzing faults affecting `CPU_rank1`, `MEM_1MB_Walk` and `SOCK_1KB_10Mesg` (marked in dark gray in Table I). The analyses in Sections IV, V and VI focus on `KnownFault`; our observations for `UnknownFault` are similar. We evaluate both use cases in Section VII.

We use hardware performance counters to measure software and hardware behavior. Modern microprocessors provide hundreds of counters; the Opteron 10h microarchitecture has 272 major counters, many with multiple options. However, this architecture imposes a constraint that only four counters can be monitored simultaneously (most architectures impose similar constraints), so we must choose them carefully to ensure that different fault types have a different effect on each counter. Since we had little intuition about which counters would be the best to monitor, we used observations from fault-free runs and runs with each of the three fault types to select them. We repeated this experiment 20 times to compute the range

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

Table I. Types of injected faults

CPU_incr
```
for (int j =0; j<100000; j++) n++;
```
CPU_powlog
```
for (int j=0; j<100000; j++) val+=pow(x+j, y+j) + log(x);
```
CPU_mmm
```
cblas_dgemm(100,100,100,mtx0,mtx1,mtx2);
```
CPU_rank1
```
cblas_dger(100,100,vec0,vec1,mtx);
```
MEM_*MB_All
```
for(int i=0; i <1000; i++)
   buf[randInt64()%size] += buf[randInt64()%size];
```
MEM_*MB_Walk
```
for (int i=0; i<1000; i++) {
   // Accesses offset n1Off/n2Off from n1/n2
   int n1Off=randInt16(), n2Off=randInt16();
   buf[(n1*32768 + n1Off ) % size] +=
      buf[(n2*32768 + n2Off) % size];
   // Advance current location of walk n1/n2
   n1 = (n1 + sizeof(int)) % 32768;
   n2 = (n2 + sizeof(int)) % 32768;
}
```
SOCK_*KB_*Mesg
```
// numMesgs = 1 or 10
// mesgSize = 1KB or 32KB
// mesg[] is a buffer of size mesgSize
for (int j=0; j < 10; j++) {
   // Establish socket from thread to itself
   outSock = socket(...); connect(outSock, ...);
   select(...); inSock = accept(...);
   // Communicate on the socket
   for(int i=0; i<numMesgs; i++) {
      write(outSock, mesg, mesgSize);
      read(inSock, mesg, mesgSize);
   }
}
```

Figure 3. Pseudocode of injected fault threads

of each counter and fault combination: [average value +/- 100 standard deviations]. We select counters with non-overlapping ranges in each of these four execution scenarios since this helps to differentiate these fault types. In this study we used the following counters [14], which our experiments showed are the most useful for differentiating the injected faults:

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

IV. Modeling Approach

This section describes our approach for creating statistical models of application behavior. Models are trained on example non-faulty application runs as well as runs with various types of faults injected. When applied to new runs they classify individual events as non-faulty or faulty and if faulty, indicate the fault's type and location. This initial approach represents a naïve application of classification algorithms to the problem of fault detection and classification.

Figure 2 illustrates our modeling procedure, which begins by collecting a set of training and evaluation runs for each application. Our training set consists of 16 non-faulty runs and 16 faulty runs of each type, giving us a total of 160 runs of each application. During the ith faulty run, we inject the fault into the ith process of the MPI application (recall that each run uses 16 processes) to capture the effect of faults on each process. In each faulty execution, the fault thread's execution is overlapped with most of the application's main computation phase. Our evaluation set is similar, except that we draw the faults from either the KnownFault or UnknownFault use cases (depending on the experiment) and execute the fault thread for 1 second at a random point in the main computation phase. Further, we evaluate the techniques on 40 additional non-faulty runs.

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

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

We then train a supervised classifier on the event's feature vector: (i) unique ID of its starting and ending MPI call stacks, (ii) execution time, and (iii) elapsed values of the four performance counters. We used the Weka 3.6.2 [15] implementations of the following supervised classifiers: Random Forest, C4.5 Decision Tree, Logit Boost and Random Committee. Since the large event counts in the training runs (from 7e+5 for MG to 2e+7 for LU) make

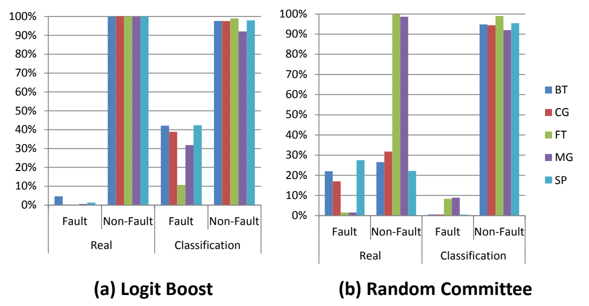

Figure 4. Prediction accuracy in the `KnownFault` use case using naïve training. Note that in this use case too, not all windows are faulty and likewise, the classifier does not flag every window as faulty. This explains the labels "Fault" and "Non-fault" in the figure.

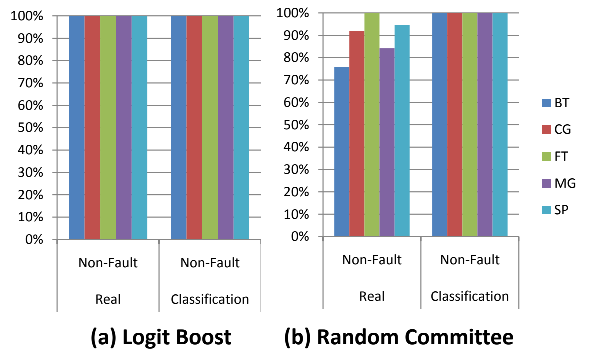

Figure 5. Prediction accuracy on non-faulty runs using naïve training.

training on all events too expensive, we trained classifiers on 500 randomly chosen events in each of the 160 runs, for a total of 80,000 events.

We use each classifier to label each event in the evaluation runs. To make a statement about the health of the overall application at a given point in time we then aggregate events across all processes into 50ms non-overlapping time windows and use the labels of the events in each window to label the window itself. A window is labeled `NO_FAULT` if no faults were detected or the only fault labels are for `OTHER_PROCESS`. If the window does have events with `THIS_PROCESS` fault labels, it is given a fault label. The type of the fault (`CPU`/`MEM`/`SOCK`) is chosen to be the the most common fault type among the individual event labels. The fault's location is computed by looking at which processes have events with `THIS_PROCESS` fault labels. The process that has the largest fraction of such labels is identified as the fault's location.

Figure 4 shows the accuracy of this approach with the Logit Boost and Random Committee classifiers when `KnownFault` injections are used in the evaluation phase. The Real Fault bars indicate, of the windows during which there really is a fault, the percentage of these did the technique flag with the correct fault type and location/ The Real Non-Fault bars represent, of the windows during which there is no fault, the percentage of these did the technique correctly flag as non-faulty. These therefore correspond to the traditional machine learning notion of *Recall*. The Classification Fault bars indicate, of the windows the classifier flagged as faulty, the percentage of these really had a fault of the correct type and location. The Classification Non-Fault bars indicate, of the windows the classifier flagged as a non-faulty, the percentage of these really did not have a fault. These therefore correspond to the traditional machine learning notion of *Precision*. The results are similar for `UnknownFault` and the Random Forest and Decision Tree algorithms perform similarly to Random Committee.

The most important insight from these experimental results is that all classifiers perform poorly on windows with faults. LogitBoost is slightly better when it classifies a window as faulty, but in all the faulty cases, the performance of the classifiers is such that they will be unusable in practice. On the encouraging side, the data shows that the supervised classifiers are usually correct when they label windows as non-faulty. However, the Random Committee classifier can mislabel windows in which no fault occurred. This message is reaffirmed from Figure 5, which shows the performance of the classifiers when no fault is injected in the evaluation runs. Because it is more selective in its classifications, Logit Boost performs best on most of our experiments, especially as part of the fault clustering algorithm that we describe in Section VII. Therefore, our remaining results focus on the Logit Boost classifier.

To understand the source of the classifiers' poor performance, we considered the possibility that the classifiers were simply not provided with sufficient information about the application's behavior at the time of the fault. As such, we expanded the feature vectors that are given to the classifiers to consist of the counter values of sequences of five preceding events, as opposed to the original experiment where each feature vector corresponds to one event. Each feature vector was five times larger and was labeled using the label of the last event in the sequence. Figure 6(a) and (b) show the accuracy of this model on labeling 50ms time windows in the `KnownFault` faulty runs and non-faulty runs with Logit Boost. The data shows that the resulting classifier misses all faults, indicating that additional history information does not help and indeed reduces accuracy. This is likely due to model overfitting which affects models with too many parameters.

V. Abnormality-Enhanced Classification

The naïve approach performs poorly because system faults do not affect applications consistently. For example, faults that affect CPU performance have little effect on memory-intensive code. Software problems are even more

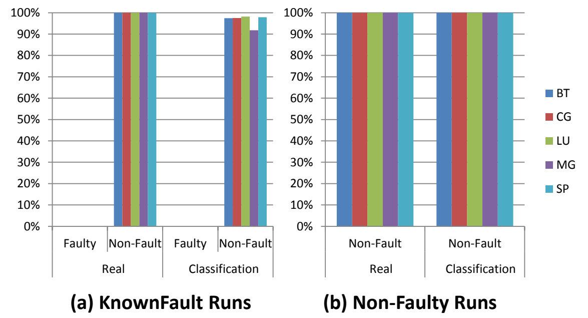

Figure 6. Prediction accuracy of Logit Boost classifier on `KnownFault` faulty runs and non-faulty runs, using naïve training with 5-event history on faulty runs

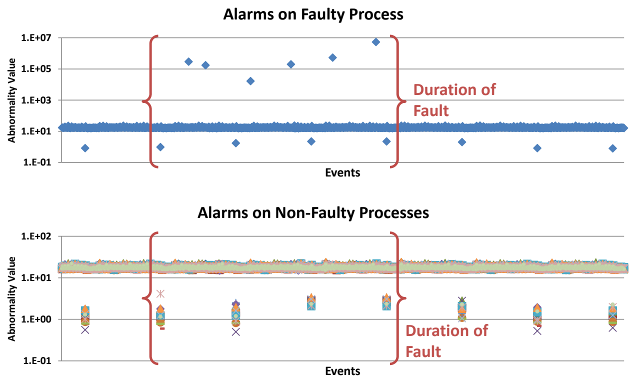

Figure 7. Abnormality values of events in a run of `BT` injected with `CPU_incr` (top sub-figure) and with no injected fault (bottom sub-figure)

irregular because they are subject to OS scheduling. Figure 7 illustrates this effect in a run of the `BT` application with an injected CPU-intensive thread. The horizontal axis shows individual events ordered in time and the vertical axis shows the abnormality value of the event, measured as $-log$(probability of event's counter values). These abnormality values, defined precisely below, are larger for more unlikely events; the logarithm focuses the view on the most unlikely events. The figure shows that most events during the non-faulty time period behave normally (abnormalities $<$ 1E+2) and even when the fault is injected most events still behave normally; *only a few events are significantly affected* (abnormalities $\sim$ 1E+6). The reason for this is that the CPU intensive fault only affects some code regions in BT and only events corresponding to those code regions have high abnormality values.

Drawback of fault classification with traditional classifiers: Traditional classifiers perform poorly on this problem because the probability distribution from which event behaviors are sampled depends on information that is not available as part of training. This is illustrated in Figure 8. A given application execution may be fault-free or may be affected by one of several types of faults at some point(s) in the execution. An execution is a sequence

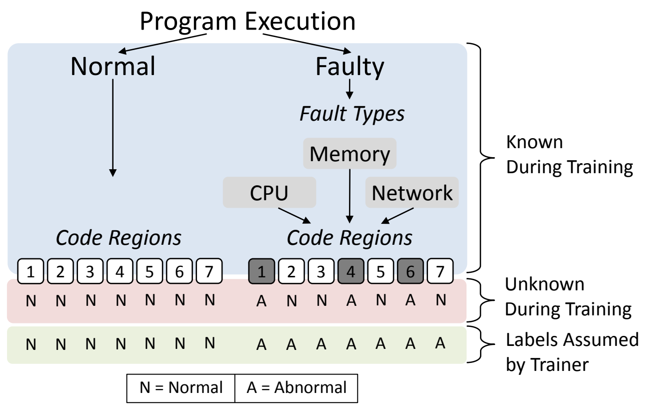

Figure 8. Hierarchical Probability Distribution of Application Vulnerability to Faults

of events, which correspond to various code regions. The values of performance counters observed during each event depend on its code region, the type of fault (if any) affecting the application and whether the current execution of the code region is vulnerable to this fault. The model training procedure runs the application, chooses the faults to inject (if any) and observes which code regions are executed and their associated performance counter values. However, the training procedure cannot determine how vulnerable a given execution of a code region is to the given fault because this requires a very detailed understanding of both the program and the infrastructure on which the program is executing, (e.g., how a code region uses the hardware and the OS scheduler's state during the region's execution). In fact, if this analysis could be performed for arbitrary code regions, there would be no need for a statistical model. Since this key piece of information cannot be provided by the training procedure to the classifier being trained, it is necessary to approximate this information by labeling *all* the events that occur during a fault as faulty.

Unsupervised learning as a preprocess step before training: This choice of labeling produces a very noisy training set that consists of the "Normal" events - almost all behave normally with a few outliers, and "Faulty" events - most behave normally and some behave abnormally. The significant overlap in behaviors of the two sets makes it very difficult for most classifiers to differentiate them. Because this problem occurs due to a lack of information about the vulnerability of code region executions to faults, a possible approach to improve model accuracy is to design an additional analysis that infers this property from the available information. A code region's execution is vulnerable to a fault if its behavior is significantly affected by it. As such, we need an analysis that determines whether a given event represents normal or abnormal behavior of its code region within its associated execution context. The resulting Normal/Abnormal labels can be used before the the traditional classifiers are trained. The hypothesis is that this will allow the classifiers to achieve significantly higher fault detection and classification accuracy.

We evaluate this hypothesis through a simple unsupervised algorithm. This algorithm builds for each code region and event context (call stack) a separate probability distribution for each of the five observed quantities (execution time and the four performance counters) over all non-faulty events in training executions that share this context (a total of five 1-dimensional distributions per context). A given event is Normal if there is a high probability of observing its time and counter values given the distribution of its code region and context. It is deemed Abnormal otherwise. Since the true probability distributions of the time and performance counters are not known in advance, we approximate them using normalized histograms, which is a non-parametric density estimator.

We implement normalized histograms through an algorithm used in the AutomaDeD tool [16]. This algorithm, which does not require an a priori range of values, consists of several steps. First, we remove the top and bottom 10% of the observed data values to eliminate outliers. We then assign the remaining values to a fixed number of clusters and create a histogram bucket for each cluster. We derive a continuous probability distribution from these buckets as follows. We linearly interpolate between adjacent buckets and attach the upper and lower halves of a Gaussian distribution to the largest and smallest bucket to model probabilities outside the observed region. We then normalize the resulting function so the area under the curve is 1, as required for a probability distribution. Our experiments use histograms with 30 clusters, which as our experiments show are appropriate for most of the data sets. They result in density models that faithfully capture the underlying distributions, balancing between models that are too smooth and too noisy.

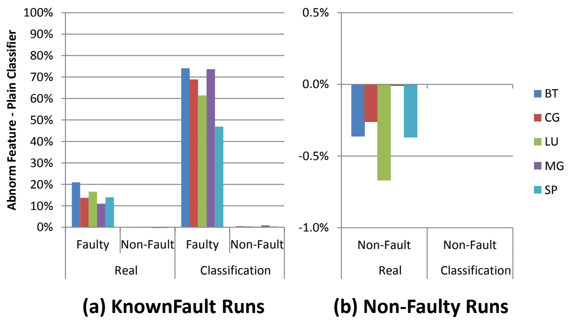

Figure 9. Improvement in prediction accuracy of abnormality-enhanced Logit Boost classifier over the vanilla Logit Boost classifier on `KnownFault` faulty runs

The simplest way to use these probability distributions is to refine the existing faulty/non-faulty label within the training inputs provided to the classification algorithms in Section IV. For each event in the training runs we compute the probabilities of its observed quantities (time and performance counters) to compute its *abnormality value*, which is the negative logarithm of its probability. We use this measure because the logarithm function provides greater resolution for the small probabilities of abnormal events and the negation ensures that higher values imply greater abnormality, which is more intuitive. The abnormality value of an entire event is the Euclidean average of the abnormality values of its observed quantities: $\sqrt{\frac{\sum_{i=1}^{n}(-log(prob_i))^2}{n}}$, where $prob_i$ is the probability of quantity i within the event given the observed distribution of that quantity, and $n = 5$ for our work.

In training, we label as "faulty" all events that occur while the fault thread executes and have abnormality values above a given threshold. Any event is tagged with the fault type that was injected and the fault's location. We label as "non-faulty" all events that occur while there is no fault or that have abnormality values below the threshold. The threshold is the maximum abnormality value observed during the non-faulty training runs value plus three standard deviations. This clearly differentiates abnormal events from the normal ones. Further, since the raw abnormality value $-log(prob_i)$ of each event with respect to the distribution of each observable i is available and can help point to the type of fault that is occurring, these five probabilities (time and 4 counters) are included as additional features in the training set. Thus, for each event, we have 11 features: the event context, 5 observables (time and 4 counters), and 5 abnormality values. Finally, while we trained our original model on 80,000 randomly chosen events, we train the abnormality-enhanced model on 40,000 normal events and 15,000 abnormal events (divided evenly among training runs) to ensure that it is trained on both event types. These event counts ensure that the comparison between the naïve and abnormality-enhanced models is not biased against the naïve by constraining the latter to train on fewer events than the naïve model.

Results from using the unsupervised learning pre-step: Figure 9(a) evaluates this algorithm, showing the difference between the success rates of the new predictor with abnormality information (denoted ``Abnorm Feature'') and that of the original predictor (denoted ``Plain Classifier'') on `KnownFault` faulty runs (similar results are seen for `UnknownFault`). As before, we show accuracy for labeling 50ms time windows, which describe the health of the overall application rather than just a single event. The data shows that the abnormality information increases the fraction of correct classifications by approximately 60% and the fraction of real faults windows by approximately 15%. Note that most real faulty time windows are still classified incorrectly.

Model	Abnormality Filtering	Absolute Observations	Relative Observations	Abnormality Values	Abnormality Probabilities
		5 features	5 features	5 features	15 features
`Plain`		✓			
`Abnorm Feature`	✓	✓		✓	
`Abnorm Feature Relative`	✓		✓	✓	
`Abnorm Prob Feature`	✓	✓		✓	✓
`Only Abnorm Prob Feature`	✓				✓
`Abnorm Filtered`	✓	✓			
`Abnorm Filtered Relative`	✓		✓		

Table II. Models and their features

However, when a window is actually classified as faulty these classifications in general consistently predict the correct fault type and location (the application process that was faulty). When windows are erroneously classified as faulty, the assigned labels are erratic, with nearby windows being labeled with different fault locations and types. Section VII presents an algorithm that uses these insights to detect faults by looking at clusters of frequent identical fault classifications.

Figure 9(b) shows the difference between `Abnorm Feature` and `Plain Classifier` for non-faulty runs. Although abnormality information improves fault detection rates (Figure 9(a)), it also slightly increases the number of non-faulty windows classified as faulty by approximately .25%. As discussed in Section VII, these errors cause some false positives in tools based on this model but do not significantly degrade the technique's overall utility.

VI. Managing Training Features

We have established that filtering the training set based on event probabilities can improve model accuracy. We now investigate how the way the new information is used by the `Abnorm Feature` algorithm influences its accuracy. Table II describes the models analyzed in this section, detailing the number and type of feature they use.

First we explore the possibility of providing additional features that further clarify the type of fault that is affecting the application. This algorithm, denoted `Abnorm Prob Feature`, computes probability distributions for the abnormal events for each fault type. The scheme is illustrated in Figure 10. Probability distributions are created with normalized histograms as before. In our experiments this produces 15 distributions: 5 observables times 3 fault types. Each event's feature vector from the `Abnorm Feature` training set is then extended with the probability of its observables with respect to each of these distributions (these are denoted "abnormality probabilities"). For a given event the probability of some observable O with respect to a distribution associated with fault type F measures how similar this event's observable O is to observable O of abnormal events induced by fault F. This training type produces a training set with 26 features: the calling context, 5 observables (time and 4 performance counters), 5 abnormality values and 15 abnormality probabilities.

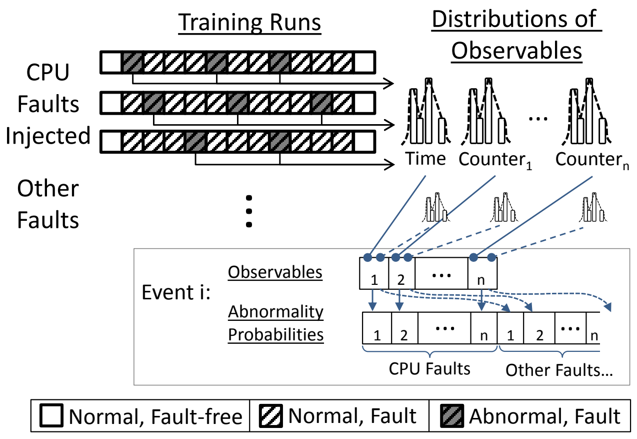

Figure 10. Generation of probability distributions from abnormal events for `Abnorm Prob Feature`

Since the above training set has a large number of features, model accuracy may be poor due to overfitting. Further, the combination of abnormality values and abnormality probabilities obscures their relative utility. As such, we also evaluate a variant of this algorithm that prunes the features used by `Abnorm Prob Feature` by removing the observables and the abnormality values, leaving only each event's calling context and its abnormality probabilities (16 features). This approach, which we denote `Only Abnorm Prob Feature`, is similar to a Naïve Bayes classifier. Note that just like `Abnorm Feature`, fault labels in both these training sets are filtered such that only abnormal events are given a fault label. Since we build abnormality distributions using only abnormal events, they correspond to far fewer events than the distributions of non-faulty runs. Thus, if we do not observe any events for a given event context, we set the probability of observables of all events with this context to a dummy constant.

Figures 11(a,b) and 12(a,b) show the accuracy difference between these training sets and `Abnorm Feature` on labeling 50ms time windows using the Logit Boost classifier on `KnownFault` faulty runs and non-faulty runs, respectively (e.g. `Abnorm Prob Feature - Abnorm Feature`). The introduction of new features (`Abnorm Prob Feature`) reduces the accuracy of the classifier both for windows during faulty runs that are classified as faulty (23% reduction on average) and windows that

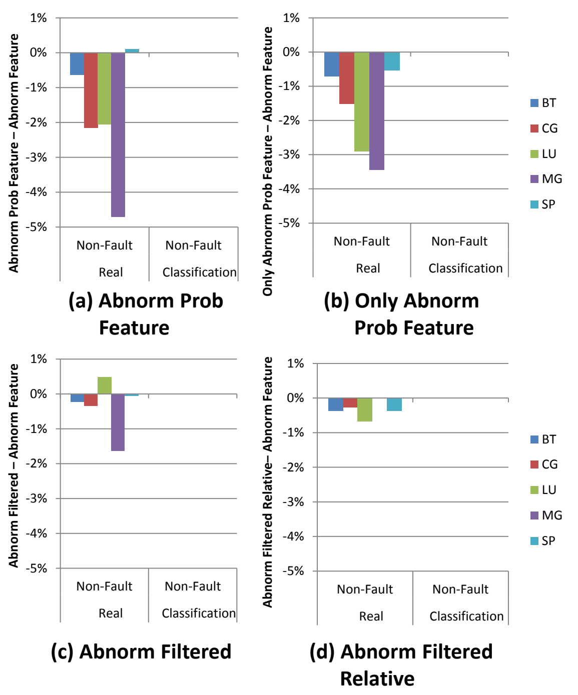

Figure 12. Difference in accuracy on non-faulty runs using abnormality as a feature, relative to `Abnorm Filter`

are actually non-faulty during non-faulty runs (2% reduction). However, it is still better than the `Plain` classifier. Training using `Only Abnorm Prob Feature`, which removes observables and abnormality values, also reduces accuracy relative to `Abnorm Feature` but to a smaller extent (average 11% less accurate on windows classified as faulty in faulty runs, 1.5% less on actually non-faulty windows during non-faulty runs). This indicates that abnormality probabilities are less useful than abnormality values but the reason for this is not clear. Since `Only Abnorm Prob Feature` is more accurate than `Abnorm Prob Feature`, the larger number of features in the latter (26 compared to 16) must be confusing the classifier. This may also explain the poorer accuracy of `Only Abnorm Prob Feature` relative to `Abnorm Feature` since the former has almost three times as many features as the latter. If that is the case then a more sophisticated classifier may be able to take advantage of the additional features. Another explanation may be the fact that abnormality distributions are constructed from just the abnormal events, which are relatively few in number. In contrast, the probabilities used by `Abnorm Feature` come from distributions of normal events, which are much more plentiful. If this is the reason for `Only Abnorm Prob Feature`'s lower accuracy relative to `Abnorm Feature`, then the use of more training runs should improve it. We leave a more detailed evaluation of these models to future work.

Since addition of features results in worse accuracy, we also evaluate a variant of the algorithm that takes the training set from `Abnorm Feature` and excludes each event's 5 abnormality values with respect to each observable (time and 4 counters). This training approach is denoted `''Abnorm Filter''`. Note that here too, the abnormality probability has first been used to separate the perceived abnormal events and the marked events are used for training. Figures 11(c) and 12(c) show the accuracy of the resulting classifier using Logit Boost on labeling 50ms time windows in `KnownFault` runs and non-faulty runs, respectively. These figures show the difference between the accuracy of the two techniques (`Abnorm Filter - Abnorm Feature`). The removal of the abnormality features has little effect on overall model accuracy in general and actually improves it for LU in faulty runs. This means that these features do not provide any additional power with respect to the original observation values, at least not with the Logit Boost classifier. As such, the main value of the probability distributions is in filtering fault/non-fault labels rather than as serving as a feature in their own right.

Another issue with applying statistical methods to application behavior is that different code regions behave very differently, with some significantly longer than others. Thus, if a given number of cache misses (counters provide the total count of misses, not the rate) is highly abnormal for one code region may be perfectly ordinary in another. Because this makes it more difficult for classifiers to find general patterns that apply to different code regions, we evaluated the use of relative rather than absolute observations to train the models. Specifically, we modified the training set from `Abnorm Filter` such that instead of providing raw event counter values in the training set we divided the value of each counter by the event's execution time. This training approach is denoted `''Abnorm Filter Relative''` and Figures 11(d) and 12(d) show the difference between it and `Abnorm Feature` in `KnownFault` runs and non-faulty runs. The use of relative observations improves accuracy on faulty windows during faulty runs by 2% and reduces it for actually non-faulty windows during non-faulty runs by .4%. Section VII shows that the improved accuracy during faulty runs has a greater effect on our tools ultimate effectiveness than the difference for non-faulty runs.

VII. Event Clustering Fault Detection

Fault detection and characterization at the level of individual events or 50ms time windows is too fine-grained and verbose for human consumption. To provide usable information for system administrators, we need a tool that concisely summarizes the time period when the fault occurred, the portion of the system affected and the fault's type without excessive duplication. We now present

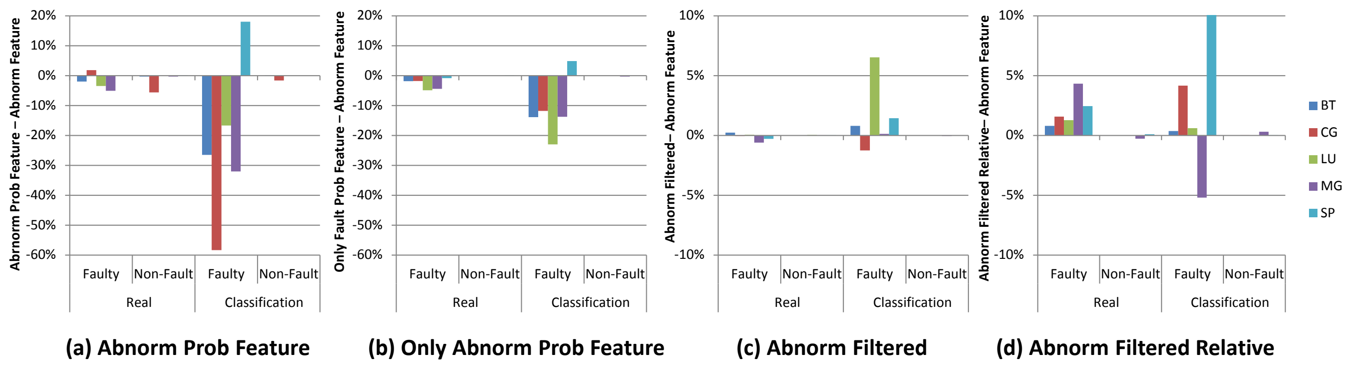

Figure 11. Difference in accuracy on `KnownFault` faulty runs using abnormality as a feature, relative to `Abnorm Filter`

techniques that provide this level of information.

Our classification algorithms label individual events and time windows with the type of fault that occurred, if any. To aggregate these labels into a single report of a fault's time, location and type it is necessary to look for a dense set of time windows that have the same fault classification label. We do this via a simple one-pass algorithm that identifies a time period as faulty if:

- Its starting and ending windows have fault labels; and
- In the intervening windows, those with fault labels have the same fault type and location process and remaining windows have the `NO_FAULT` label; and
- All adjacent time windows with a fault label are less than τ seconds away from each other.

The parameter τ controls the fault density of the desired clusters; our experiments use $\tau = .5s$. While simple, this algorithm provides low-latency online fault detection. Our experiments with more complex, less intuitive algorithms yielded similar performance so this algorithm is preferred.

Figure 13 shows fault prediction accuracy on the `KnownFault` faulty runs and non-faulty runs with our algorithm and the Logit Boost classifier that was trained using the `Plain Classifier`, `Abnorm Filter` and `Abnorm Filter Relative` algorithms. We define successful fault detection as identification of the correct fault type and location for a single contiguous time period that overlaps with the time when the fault was injected and does not exceed the duration of the actual fault injection by more than a factor of 2. This definition ensures that we declare success in the case that for a real fault, the tool identifies as faulty a single time region that closely resembles the fault's actual properties. We see that the clustering algorithm has poor accuracy when it uses the `Plain Classifier`, with only 9% of real faults detected and only 32% of alerts being correct. However, it has no false positives on the non-faulty runs. This is because its predictions are very erratic. Since it is unlikely that two adjacent predictions are identical, the result is that it predicts no faults at all.

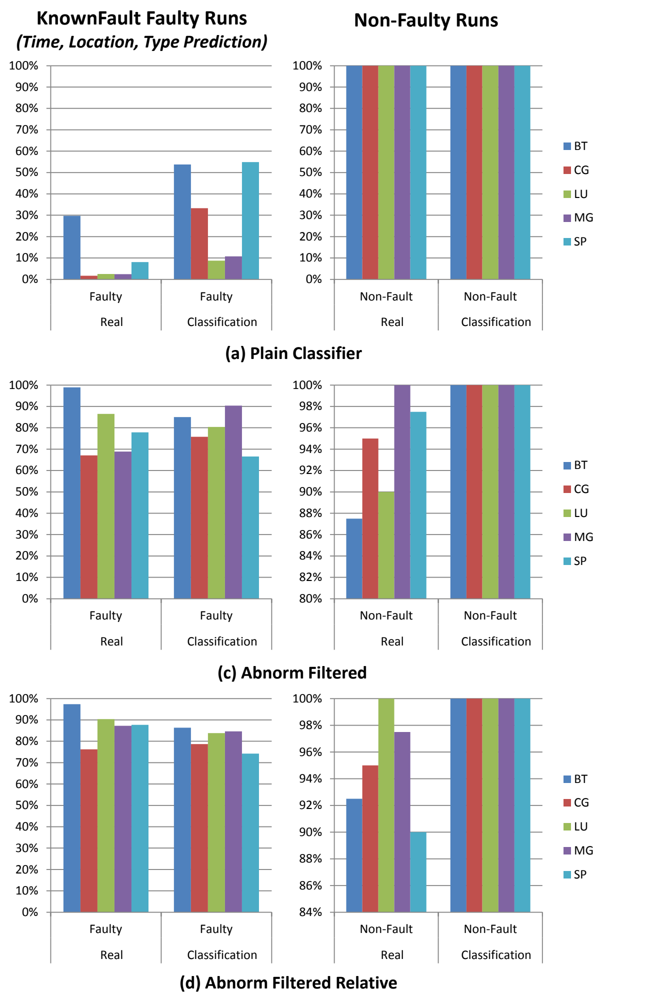

Figure 13. Fault detection and classification accuracy of clustering on `KnownFault` faulty and non-faulty runs

The accuracy of `Abnorm Filter` for real faults is significantly higher, between 67% for CG and 99% for BT. Fault classifications are accurate from 67% of the time for SP to 90% for MG. On non-faulty runs it mis-detects faults between 0% of the time for MG to 12.5% for

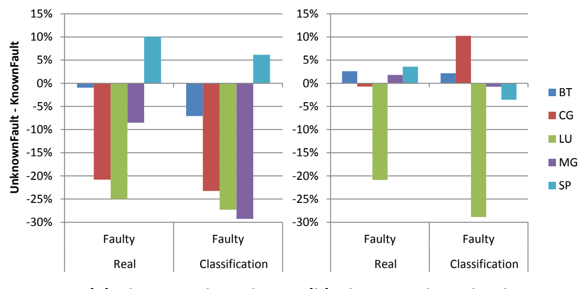

Figure 14. Difference between fault detection and classification accuracy for `UnknownFault` and `KnownFault`

BT. `Abnorm Filter Relative` improves accuracy further, detecting real faults between 76% of the time for CG and 97% for BT. On non-faulty runs, its false positive rate ranges from 0% for LU and 7.5% for BT.

Figure 14 shows the difference in accuracy for detecting faults that were similar to the training set but not included in it (`UnknownFault`) and those that were in the training set (`KnownFault`). It focuses on models `Abnorm Filter` and `Abnorm Filter Relative`. Not surprisingly, detection of faults that were anticipated by administrators (`KnownFault`) is generally easier. Accuracy with `Abnorm Filter` is 5-30% lower for BT, CG, LU and MG but 5-10% better for SP. The performance of `Abnorm Filter Relative` is more consistent the two datasets, only significantly degrading in accuracy for LU (21% worse for real faults and 28% worse for fault classifications). Thus, our approach works best when administrators understand fault types that may occur and design synthetic faults that accurately represent them.

Figure 15 summarizes our conclusions. It shows accuracy for three fault detection scenarios: (i) anticipated faults (`KnownFault` runs); (ii) unanticipated faults (`UnknownFault` runs); and (iii) fault-free runs. All use the Logit Boost classifier. Without abnormality information real faults are detected on 1-39% of the runs (8% for `KnownFault` and 12% for `UnknownFault` on average) with the `Plain` Classifier. In contrast, `Abnorm Filter` is successful on 46-99% of runs (80/70% on average) and `Abnorm Filter Relative` succeeds on 70-100% of runs (87/85% on average). Similarly, classification of runs as faulty improves from 10-67% (32/66% on average) with `Plain` to 53-90% (80/63% on average) with `Abnorm Filter` and 55-86% (76/77% on average) with `Abnorm Filter Relative`.

On non-faulty runs, `Plain` never mistakenly detects faults. `Abnorm Filter` incorrectly detect faults in 0-12.5% (6% on average) of non-faulty runs and `Abnorm Filter Relative` makes such mistakes on 0-8.5% (5% on average) of the runs.

Clustering based on `Abnorm Feature` results in slightly worse accuracy than with `Abnorm Filter`.

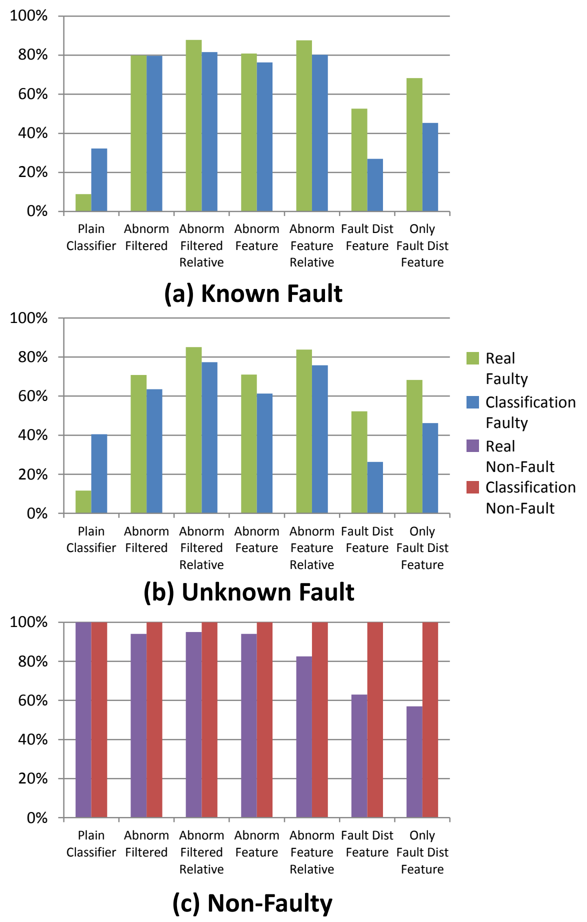

Figure 15. Summary of fault detection accuracy results for all model types and sets of evaluation runs

While, accuracy improves if it is augmented with relative observations, much like `Abnorm Filter` the rate of mistaken fault detections during non-faulty runs grows worse. With `Abnorm Prob Feature` and `Only Abnorm Prob Feature`, which use abnormality probabilities, accuracy is significantly worse both in faulty and fault-free runs, usually by tens of percentage points.

VIII. Related Work

We group automatic fault detection methods into two categories. The first category, the *probability-based-classifier* (or generative) approach, classifies events based on their probability as derived from a probability model (usually a distribution). Cohen et al. [17] build a Tree-Augmented Naive (TAN) Bayesian network model to predict Service Level Objectives (SLOs) violations by capturing correlations of system metrics more efficiently than in a Bayesian network. Guo et al. [18] use a mixture of Gaussian distributions to capture the probability of performance metrics. Other approaches [8] model probability distributions with histograms to characterize deviations from normal behavior. Hamerly and Elkan [19] predict disk failures with Bayesian classifiers and mixture models.

The second method, the *traditional-classifier* (or discriminative) approach, uses a classifier such as a decision tree, neural network or a clustering algorithm to determine whether events are normal or abnormal. Chen et al [20] train a probabilistic context-free grammar on faulty and non-faulty runs to identify abnormal web requests in large e-commerce systems. Ozonat et al [21] detect performance anomalies by clustering application traces and looking for small clusters. Gao et al [22] use a Markov model to identify abnormal changes of system metrics correlations.

Our work combines these approaches. We use abnormality information (a generative approach) to filter input for the traditional-classifier approach. We demonstrate that the combination improves detection accuracy for common faults. To the best of our knowledge, no previous work has studied a similar hybrid technique.

IX. Summary

We examined the problem of detecting, localizing and characterizing system faults using statistical modeling of application and system behavior. We showed experimentally that intuitive use of supervised statistical models with this problem performs poorly. We identified the reason for this disappointing performance to be the fact that system faults affect application behavior inconsistently, strongly affecting some application regions, and leaving most to execute normally. We used this information to improve the quality of fault detection by event abnormality information. Our approach builds a secondary unsupervised model to evaluate the probability that a given event will appear in a non-faulty execution. We then labeled only the truly abnormal events as faulty and used these probabilities as model features. Our experiments showed that filtering significantly improves the accuracy of detecting a fault's type, location and time period while probability features are less useful. Specifically we demonstrate that the `Abnorm Filter Relative` classifier works best, characterizing faults that were anticipated by administrators with 87% accuracy, un-anticipated faults with 85% accuracy and with a 5% false positive rate on non-faulty runs

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, “High Speed and Robust Event Correlation,” *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, “PnMPI 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] G. Bronevetsky, I. Laguna, S. Bagchi, B. de Supinski, M. Schulz, and D. Anh, “AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks,” in *International Conference on Dependable Systems and Networks (DSN)*, Jun. 2010.

[17] 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 *Operating Systems Design and Implementation (OSDI)*. Berkeley, CA, USA: USENIX Association, 2004, pp. 16–16.

[18] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira, “Tracking Probabilistic Correlation of Monitoring Data for Fault Detection in Complex Systems,” in *International Conference on Dependable Systems and Networks (DSN)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 259–268.

[19] 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.

[20] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer, “Path-Based Failure and Evolution Management,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004, pp. 309–322.

[21] K. Ozonat, “An Information-Theoretic Approach to Detecting Performance Anomalies and Changes for Large-Scale Distributed Web Services,” in *International Conference on Dependable Systems and Networks (DSN)*. Anchorage, Alaska, USA: IEEE, 2008, pp. 522–531.

[22] 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.