Problem Diagnosis in Large-Scale Computing Environments

Alexander V. Mirgorodskiy*
VMware, Inc.

Naoya Maruyama†
Department of Mathematical and Computing Sciences
Tokyo Institute of Technology

Barton P. Miller‡
Computer Sciences Department
University of Wisconsin

Abstract

We describe a new approach for locating the causes of anomalies in distributed systems. Our target environment is a distributed application that contains multiple identical processes performing similar activities. We use a new, lightweight form of dynamic instrumentation to collect function-level traces from each process. If the application fails, the traces are automatically compared to each other. We find anomalies by identifying processes that stopped earlier than the rest (sign of a fail-stop problem) or processes that behaved different from the rest (sign of a non-fail-stop problem). Our algorithm does not require reference data to distinguish anomalies from normal behaviors. However, it can make use of such data when available to reduce the number of false positives. Ultimately, we identify a function that is likely to explain the anomalous behavior. We demonstrated the efficacy of our approach by finding two problems in a large distributed cluster environment called SCore.

1 Introduction

Finding the root causes of bugs and performance problems in high-performance computing and e-commerce environments is a difficult task. One of the main reasons for this difficulty is the distributed and concurrent nature of such systems. Analyzing execution of multiple interacting threads or processes can be significantly harder than sequential execution. In a concurrent system, events can happen in different order in different runs. Furthermore, concurrent systems suffer from bugs not present in sequential software. Bugs such as deadlocks and race conditions are notorious for being hard to track down. Finally, the non-interactive nature of many system components complicates error detection. A silent failure of a process on one of the hosts may remain unnoticed until a later time.

To simplify the tasks of problem detection and root cause analysis in large distributed environments, we need to automate them to the extent possible. We envision that the programmer or the analyst would use an autonomous agent tool to help locate the cause of a problem as follows. First, the analyst injects the agent code into a running system. The agent starts monitoring processes of interest. It collects run-time execution data, identifies anomalies in the data that potentially correspond to abnormal behavior, and makes the results available to the analyst for further investigation.

In our previous work [20], we presented *self-propelled instrumentation*, which enables creation of such agents. It allows us to obtain execution traces from a running program with low overhead and without modifying the program's source code. This paper builds on that foundation to provide solutions for locating problems in distributed high-performance computing environments. To achieve this goal, we extended our techniques to collect traces from multiple processes on multiple hosts. The key challenge in this approach, however, proved to be in analyzing large sets of data collected from different hosts. The primary contribution of this paper lies in a collection of techniques for finding anomalies and their causes via automated analysis of collected data.

Our analyses currently apply to identification of failures in a distributed collection of identical processes that perform similar and comparable activities. Environments that often have this property are cluster management tools [17, 27], Web server farms, and parallel numeric codes. The as-

*This work was performed while the author was a student at the University of Wisconsin. email: mirg@cs.wisc.edu
†email: naoya.maruyama@is.titech.ac.jp
‡email: bart@cs.wisc.edu

SC2006 November 2006, Tampa, Florida, USA
0-7695-2700-0/06 $20.00

sumed similarity of behaviors allows us to find one or more processes that are substantially different from the others; we regard these processes as potential anomalies that require investigation. This approach can be further generalized to support collections of different processes by grouping identical processes together and performing analyses within each group.

To locate the root causes of bugs in a distributed application, we use a three-step process. First, we collect control-flow traces from a chosen application until it fails. Section 2 describes faults that are manifested by unusual control flows. To collect such traces, our current prototype injects a tracing agent into every process of the application. The agents record function calls and returns made by the application processes on all hosts, as described in Section 3.

Second, we perform *data categorization*, that is, identify a process or a small group of processes that failed, as described in Section 4. Finding such processes is a non-trivial task in presence of silent and non-fail-stop failures. In this step, we use two different techniques. The simplest case is when we identify anomalous processes with fail-stop behavior by looking at the process that stopped generating trace records first. The more complex case is when we identify anomalous processes with non-fail-stop behavior by using a *distance-based outlier detection* approach [4, 24]. Namely, we compare per-process traces automatically and identify ones that appear substantially different from the others. Our algorithm can work without prior knowledge of correct and incorrect behaviors, simply looking for unusual activities. However, if examples of previously investigated behaviors are available, it can incorporate this knowledge into analysis for improved accuracy.

Finally, we perform *root cause identification*, as described in Section 5. That is, we determine why the anomalous processes behaved differently from the others. In this step, we automatically locate a symptom of the problem. The symptom corresponds to the key difference in behavior between an anomalous process and the most similar normal process. Then, we describe techniques to help the analyst transition from knowing a symptom to finding the root cause of the problem.

There are several projects that propose techniques for automated anomaly detection. Magpie [2], Pinpoint [6], the work of Dickinson et al. [11], and Yuan et al. [32] are the most similar to our work. These approaches perform similar steps: collect run-time data in the trace or profile form, analyze it to identify anomalies, and possibly explain the cause of the anomaly. As discussed in Section 7, the key strength of our approach is that it meets all three of the following properties: easy deployment in production environments via self-propelled instrumentation, ability to integrate prior knowledge into analysis when available, and fine-grained localization of a problem to the level of an individual function.

We applied our prototype solution to identification of faults in a large production environment running a distributed job scheduler called SCore [27]. In three months of deployment of our framework on a public computational cluster at the Tokyo Institute of Technology, we witnessed several failures of the scheduler and successfully located their causes via trace examination. These results suggest that the combination of autonomous instrumentation and statistical trace analysis is a promising approach to localizing faults in complex distributed systems.

2 Fault Model

By looking at differences in the control flow across processes, we can find several important classes of problems. Some of those problems are bugs, others are performance problems. For each described problem, we can identify the failed process and the function where the failure happened.

Non-deterministic fail-stop failures. If one process in a distributed system crashes or freezes, its control-flow trace will stop prematurely and thus appear different from those processes that continue running. Our approach will identify that process as an anomaly. It will also identify the last function entered before the crash as a likely location of the failure.

Infinite loops. If a process does not crash, but enters an infinite loop, it will start spending an unusual amount of time in a particular function or region of the code. Our approach will identify such process and find the function where it spends its time.

Deadlock, livelock, and starvation problems. If a small number of processes deadlock, they stop generating trace records, making their traces different from processes that still function normally. Similarly, if a process waits for a shared resource indefinitely while other processes have their requests granted, they are likely to spend time in different parts of the code and we can identify the starving process. A function where the process is blocking or spinning will point to the location of the failure.

Load imbalance. Uneven load is one of the main performance problems in parallel applications. By comparing the time spent in each function across application nodes, our approach may be able to identify a node that receives unusually little work. Functions where the node spends unusually little time may help the analyst find the cause of the problem.

Note that there are several types of problems that may not be detected with our approach. First, we may not be able to find massive failures. If a problem happens on all nodes in the system, our approach would consider this normal behavior. Fortunately, such problems are typically easier to debug than anomalies. Second, we may not detect problems with no change in control flow. If an application fails on a particular input, but executes the same instructions as in a normal run, we would not be able to locate the root cause of the problem. In our experience however, such problems are relatively uncommon. Third, we may not be able to locate problem causes that occur long before the failure as we only retain a fixed number of recent trace entries. In the future, this limitation can be addressed by performing analysis online or saving older traces to disk instead of discarding them.

3 Data Collection

To collect control-flow traces from an application, we use self-propelled instrumentation [20]. When an application starts, we inject a small autonomous fragment of code, called *the agent*, into its address space. To inject the agent, we compile it as a shared library and use the *Hijack* mechanism [33] to cause the application to load the library. The agent modifies the application's binary code to insert trace statements at all function call sites. This activity is performed incrementally—functions are instrumented right before the flow of control reaches them for the first time. When executed, each trace statement generates a timestamped in-memory log record for the corresponding function entry or exit.

If an application contains more than one process running on one or more hosts, we inject a copy of the agent into each process. The copies do not communicate to each other and collect per-host traces independently. The copies are controlled by a single coordinator process that watches for a *deactivation event*—an application node crash or another user-specified exceptional condition. When such a deactivation event happens, the coordinator saves the per-host traces to disk to make them available for analysis.

To retain the in-memory trace data after a sudden application crash, we put the trace buffer in a shared-memory segment. The segment is mapped both into the application process and a helper tool, so that when the application process dies, the segment is left around and the helper tool can save its contents to disk. This technique does not protect the data from the crash of the operating system or a hardware failure, but it is useful for surviving application crashes, which are typically more frequent than whole-node crashes. Note that other tools use a shared-memory segment for fast accesses to performance data from an external process [18, 31]. However, to our knowledge, spTracer is the first tool that uses this technique for achieving trace durability.

The size of the trace depends on activities in a traced application. At the extreme end, a single process can make several hundred million calls to an empty function in a second on modern hardware. While such high rates are infrequent in practice, function call traces can still grow large, consume substantial amounts of memory, and adversely affect execution of all processes in the system. To limit this perturbation, we restrict the size of the trace buffer and organize it in a circular way—after reaching the end of the buffer, the agent starts adding new trace records from the beginning. This technique retains the most recent events that occurred before the crash.

In our experiments, we traced job management daemons, `scored`, of the SCore distributed environment [27]. A ten-megabyte trace buffer was sufficient for capturing several minutes of normal execution of each daemon. Furthermore, the run-time impact of tracing such daemons on execution of user jobs proved negligible. The end-to-end slowdown of applications in the NPB suite [29] was less than 1% while the daemons were traced.

4 Data Analysis: Finding a Misbehaving Host

Once the traces are obtained, saved to local disks, and gathered at a central location, we start analyzing them. The first step in our analysis is to find the misbehaving host. For this purpose, we use two techniques: identification of a node that stopped generating traces first and identification of traces that are the least similar to the rest. The first technique locates hosts with fail-stop problems; the second focuses on non-fail-stop problems. Note that none of these techniques use an external fault-detection mechanism. Instead, they locate the misbehaving host from the traces themselves. External fault detectors typically look for known symptoms, such as network timeouts or node crashes [6]. In contrast, our approach is able to find silent failures and failures with symptoms never seen before.

4.1 The Earliest Last Timestamp

A simple technique for finding the failed host is to identify the host that stopped generating trace records first. The trace whose last record has the earliest timestamp is reported to the analyst as an anomaly. In general, the technique may be effective for detecting fail-stop problems, such as application crashes or indefinite blocking in system calls. It proved to work well for several of our experimental studies.

To compare last timestamps across hosts, we convert them to the absolute time, assuming that the system clocks of all hosts are well synchronized. Next, we compute the mean and standard deviation of last timestamps across all hosts. If the earliest timestamp is substantially different from the mean and the delta exceeds the attainable clock synchronization precision [19], we assume the fail-stop scenario and report the host with the earliest timestamp to the analyst. Otherwise, we assume that the problem has the non-fail-stop property. Such problems as performance degradations, livelocks, starvation, infinite loops in the code are often harder to debug than fail-stop crashes, and we locate them with the following technique.

4.2 Finding Behavioral Outliers

To find anomalous hosts that exhibit non-fail-stop behavior, we identify *outlier traces*, i.e. individual traces or small collections of traces that appear different from the rest. To identify such traces, we use a distance-based outlier detection approach [4, 24]. First, we define a *pair-wise distance metric* that estimates the dissimilarity between two traces. Then, we construct a *suspect score* that estimates the dissimilarity between a trace and a collection of traces that we consider normal. We use the suspect score as a *rank* of a trace. Traces with the highest rank are of interest to the analyst as they correspond to hosts whose behavior is most different from normal.

Note that outliers can correspond to either true anomalies or to unusual but normal behavior. For example, the master node of a master-worker MPI application may behave substantially different from the worker nodes. Workers perform units of work, while the master distributes the units among the workers. Approaches that look for outliers would flag the normal behavior of the master as an anomaly.

To eliminate such false positives, our approach utilizes data from known-correct previous runs, if available. If a previous correct run contained a trace similar to that of the MPI master node in the current run, the master will not be flagged as an outlier. Our analyses provide a uniform framework to handle both the case when we have data only from the failed execution (the *unsupervised* case [13]) and when we also have data from a known-correct previous run (the *one-class ranking* case [28]).

4.2.1 Pair-wise Distance Metric

To define the distance between traces for two hosts, g and h, we use a two-step process similar to Dickinson et al. [11]. First, we construct per-trace function profiles that serve as a summary of behaviors in each trace. Then, we define the distance metric between the profiles. The profile for host h is a vector $\boldsymbol{p}(h)$ of length F, where F is the total number of functions in the application.

$$\boldsymbol{p}(h) = \left(\frac{t(h, f_1)}{T(h)}, \ldots, \frac{t(h, f_F)}{T(h)} \right) \tag{1}$$

The i^{th} component of the vector corresponds to a function f_i and represents the time $t(h, f_i)$ spent in that function as a fraction of the total run time of the application, $T(h) = \sum_{i=1}^{F} t(h, f_i)$.

Note that unlike the count profiles of Dickinson et al., we use time profiles. In our experience, time profiles are able to detect a wider range of behavioral problems. For example, if an application blocked indefinitely in a system call, the anomaly will be immediately visible in the time profile, but not the count profile. Also, time profiles are a natural match for detecting performance problems.

We can treat different call paths to each function as separate functions, so our definition of the profile can seamlessly include *call path profiles*. For example, assume that an execution trace contains two paths to function C: $(A \rightarrow B \rightarrow C)$ and $(D \rightarrow E \rightarrow C)$, where $(A \rightarrow B)$ denotes a call from A to B. We can consider these two paths as two different functions, f_1 and f_2. The time $t(h, f_1)$ attributed to f_1 is equal to the time spent in C when it was called from B when B was called from A, and the time $t(h, f_2)$ is equal to the time spent in C when it was called from E when E was called from D. Since the algorithms presented below are

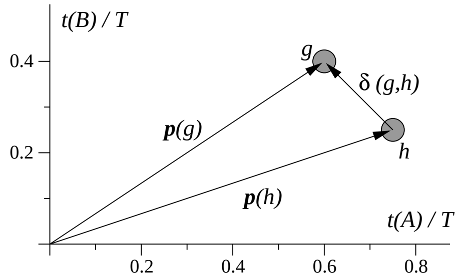

Figure 1: Geometric meaning of profiles. *The two data points correspond to hosts g and h with profile vectors $\boldsymbol{p}(g)$ and $\boldsymbol{p}(h)$. The coordinates of the points are determined by the relative contributions $t(A)$ and $t(B)$ of functions A and B to the total time T on each host. $\boldsymbol{\delta}(g,h)$ is the difference vector between $\boldsymbol{p}(g)$ and $\boldsymbol{p}(h)$.*

independent of the type of profile used, we will refer to the components of profile vectors as functions for simplicity of presentation. Section 6 presents experimental results for the path-based method.

The profile vector defines a data point in an F-dimensional space, where the i^{th} coordinate is $t(f_i)/T$, that is the time spent in function f_i as a fraction of the total time. Figure 1 shows the geometric interpretation of profiles for an example application that has two functions, A and B (i.e., $F = 2$). Host g has a profile of $(0.6, 0.4)$, representing the fact that it spent 60% of the time in A and 40% in B, and host h has a profile of $(0.75, 0.25)$.

The distance between traces g and h, $d(g,h)$, is then defined as the Manhattan length of the component-wise difference vector between $\boldsymbol{p}(g)$ and $\boldsymbol{p}(h)$, $\boldsymbol{\delta}(g,h) = \boldsymbol{p}(g) - \boldsymbol{p}(h)$. Our experiments with the Euclidean metric showed similar results.

$$d(g,h) = |\boldsymbol{\delta}(g,h)| = \sum_{i=1}^{F} |\delta_i| \qquad (2)$$

If g and h behave similarly, each function will consume similar amounts of time on both hosts, the points for g and h will be close to each other, and the distance $d(g,h)$ will be low. The advantage of computing distances between profiles rather than raw traces is that profiles are less sensitive to insignificant variations in behavior between hosts. For example, the same activity occurring on two hosts at different times may make their traces look significantly different, while the profiles will remain the same. The main disadvantage of profiles is that they may disregard subtle but real symptoms of a problem. In our experience however, once a failure occurs, the behavior of a host typically changes substantially and the change is clearly visible in the profile.

4.2.2 Suspect Scores and Trace Ranking

Our goal is to compute a suspect score for each trace. Traces with the largest scores will be of primary interest to the analyst as probable anomalies. We present two algorithms for computing such scores. The first algorithm applies to the unsupervised case where we have trace data from only the failed execution. The second algorithm applies to the one-class classifier case [28] where we have additional data from a known-correct previous run. This additional data allows us to identify anomalies with higher accuracy. Both algorithms are based on the same computational structure.

In the unsupervised case, we operate on a set of traces T collected from the failed run. We assume that T consists of one or more types of common behaviors and a small number of outliers. Ramaswamy et al. detected outliers by ranking each data point according to its distance to its k^{th} nearest neighbor [24]. We use this idea to compute our suspect scores in the unsupervised case and extend it to handle the one-class classified case.

To determine a good value of k, we evaluated the sensitivity of the algorithm to k on our data sets. As expected, values of k less than the total number of outliers (our data sets had up to three anomalies) produced false negatives, ranking some anomalous traces as normal. High values of k potentially can produce false positives, ranking some normal traces as anomalies, but we have not seen such cases in our data sets. The algorithm worked well for all k larger than 3 and up to $|T|/4$, where $|T|$ is the number of traces in T. Furthermore, the cost of a false positive to the analyst (examination of an additional normal trace) may be significantly lower than that of a false negative (overlooking a true anomaly). Therefore, we have adopted a conservative value of $k = |T|/4$.

For each trace $h \in T$ in the unsupervised case, we construct a sequence of traces $T_d(h)$ by ordering all traces in T according to their distance to h:

$$T_d(h) = \left\langle h_1, h_2, \ldots, h_{|T|} \right\rangle \qquad (3)$$

Here, $d(h, h_i) \leq d(h, h_{i+1}),\ 1 \leq i \leq |T|$. The suspect score for trace h is then defined as the distance of h to its k^{th} nearest neighbor h_k:

$$\sigma(h) = d(h, h_k) \qquad (4)$$

A trace is considered normal if it has a low suspect score and an anomaly if it has a high suspect score. Informally, $\sigma(h)$ defined by Equation 4 tries to quantify the dissimilarity between h and the nearest common behavior, since we consider all

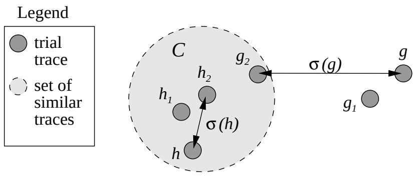

Figure 2: Unsupervised case.
Computing the suspect scores $\sigma(g)$ and $\sigma(h)$ for an anomalous trace g and normal trace h. C is a large collection of traces with similar behaviors; g_i and h_i are the i^{th} trial neighbors of g and h respectively; n_1 is the first normal neighbor of g.

common behaviors as normal. Parameter k determines what is a common behavior: a trace must have at least k close neighbors to receive a low suspect score and be considered normal.

Figure 2 shows two examples to illustrate the algorithm when $k = 2$: Trace h is part of a large set C, representing one of the common behaviors; h has a large number (more than $k = 2$) of close neighbors. As a result, its k^{th} neighbor, h_2, will be relatively close, the suspect score $\sigma(h) = d(h, h_2)$ will be low, and h will be considered a normal trace. In contrast, trace g is an outlier. It is far away from any common behavior and has only one (i.e., less than k) close neighbor. As a result, its k^{th} neighbor, g_2, will be far away, the suspect score $\sigma(g) = d(g, g_2)$ will be high, and g will be considered an anomaly.

For these two cases, the unsupervised algorithm produces desired results. It reports a high suspect score for an outlier and a low suspect score for a common behavior. However some outliers may correspond to unusual but normal behavior and they are of little interest to the analyst. Now we present a modified version of our algorithm that is able to eliminate such outliers from consideration by using additional traces from a known-correct previous run.

We add a set of *normal* traces N to our analysis. N contains both common behaviors and outliers, but all outliers from N correspond to normal activities. As a result, if an outlier g from T is close to an outlier n from N, g is likely to correspond to normal behavior and should not be marked as an anomaly. To identify true anomalies, i.e. traces that are far from any large collection of traces from T and also far from any trace in N, we compute the suspect scores as follows.

Similar to sequence $T_d(h)$ defined by Equation 3, sequence $N_d(h)$ arranges all traces in N in the order of their distance to h:

$$N_d(h) = \langle n_1, n_2, \ldots, n_H \rangle \tag{5}$$

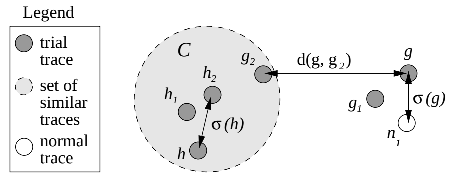

Figure 3: One-class case.
Computing the suspect scores $\sigma(g)$ and $\sigma(h)$ where both g and h are normal, but g is unusual; n_1 is the closest normal neighbor of g.

Here, $d(h, n_i) < d(h, n_{i+1}),\ 1 \le i \le |T|$. Suspect score $\sigma(h)$ is now defined as the distance of h to either its k^{th} neighbor from T or the first neighbor from N, whichever is closer:

$$\sigma(h) = \min\{d(h, h_k),\ d(h, n_1)\} \tag{6}$$

Figure 3 shows two examples to illustrate this idea: Similar to Figure 2, trace h will be considered normal, as it is part of a large collection of trial traces. Trace g is an outlier. It is far away from any common behavior and has only one (i.e., less than k) close trial neighbor. However, it has a normal trace n_1 nearby. Therefore, its suspect score $\sigma(g) = d(g, n_1)$ will be low, and g will also be considered normal.

Note that the one-class approach can also benefit from examples that may contain prior *faulty* behaviors, if available. Such data can be used to eliminate *generic symptoms*, i.e. anomalies common to many unrelated problems. Since such symptoms are not specific to any problem, identifying them may not help the analyst locate the actual problem. However, by treating examples of previous unrelated failures as normal behavior, our one-class algorithm is able to eliminate generic symptoms and identify symptoms that were unique to the problem at hand.

5 Data Analysis: Finding the Cause of the Anomaly

Once the anomalous trace is found, we provide several techniques for locating a function that is a likely cause of the problem. To define the concepts used in this section, we consider the following steps of problem evolution [1]. The first step is the occurrence of a *fault*, also referred to as the *root cause* of a problem. In the second step, the fault causes a sequence of changes in the program state, referred to as *errors*. Finally, the changes in the state cause the system to fail: crash, hang, or otherwise stop providing the service. We refer to such an event as a *failure*.

5.1 Last Trace Entry

A simple technique for identifying a function that caused the found host to behave abnormally is to examine the last entry of the host's trace. The technique can be viewed as a natural extension of our earliest last timestamp approach from Section 4.1. We identify not only the host that stopped generating trace records first, but also the last function executed by that host. Such a function may be a likely cause of the failure. The technique is often effective for identifying the causes of crashes and freezes, but may not work for non-fail-stop problems.

5.2 Maximum Component of the Delta Vector

To locate the causes (or at least symptoms) of non-fail-stop problems, we developed an approach that is a natural extension to the outlier-finding approaches described in Section 4.2. After locating outlier traces, we identify the symptoms that led us to declare those traces as outliers. Specifically, we find a function whose unusual behavior had the largest contribution to the suspect score for the outlier trace.

Recall that in Equations 4 and 6, the suspect score of a trace h is equal to the distance $d(h,g)$ of h to another trace g, where g is either the k^{th} trial neighbor of h or the first known-normal neighbor of h. In turn, the distance $d(h,g)$ is the length of the delta vector $\boldsymbol{\delta}(h,g)$. Component δ_i of $\boldsymbol{\delta}(h,g)$ corresponds to the contribution of function f_i to the distance. Therefore, by finding the δ_i with the maximum absolute value, we will identify a function with the largest difference in behavior between h and g:

$$anomFn = \operatorname*{argmax}_{1\le i\le F} |\delta_i| \qquad (7)$$

This technique worked well in our experiments. Our outlier-finding approaches were able to accurately identify the anomalous traces; examination of the maximum component of the delta vector explained their decisions and located the anomalies.

5.3 Anomalous Time Interval

Note that our approach identifies the location of the failure, but not necessarily the root cause of a problem. For example, if function A corrupted memory and caused function B to enter an infinite loop, we will locate B, but A will remain undetected. To help the analyst transition from the failure to the fault, we extended our analysis to identify the first moment when the behavior of the anomalous host started deviating from the norm. Namely, we partition traces from all hosts, normal and anomalous, into short fragments of equal duration. Then, we apply our outlier detection algorithm to such fragments rather than complete traces and identify the *earliest* fragment from the anomalous host that is marked as the top outlier. Knowing the time interval where the change in behavior occurred provides additional diagnostic precision.

6 Experimental Results

To evaluate the effectiveness of our techniques, we used the spTracer prototype to locate the causes of bugs in a large production environment running a distributed cluster management system called SCore [27]. The installation of SCore on a public computational cluster suffered from occasional failures with varying symptoms. Here, we describe the key features of SCore and show how we used spTracer to collect and analyze traces in such environment.

6.1 Overview of SCore

SCore is a large-scale parallel programming environment for clusters of workstations. SCore facilities include distributed job scheduling, job checkpointing, parallel process migration, and a distributed shared memory infrastructure. It is implemented mainly in C++, and has a large code base: it has more than 200,000 of lines of code in 700 source files.

A typical SCore usage scenario looks as follows. First, the user submits a job to the central scheduler. Then SCore finds an appropriate number of compute nodes and schedules the job on the nodes. As the job runs, `scored` daemons on each node monitor the status of the job's processes executing on the same node. Finally, when the job terminates, SCore releases acquired nodes.

To detect hardware and software failures, cluster nodes exchange periodic keep-alive *patrol* messages. All `scored` processes and a special process called `sc_watch` are connected in a uni-directional ring; when a process in the ring receives a patrol message from one neighbor, it forwards the message to the other neighbor. The `sc_watch` process monitors the patrol messages; if it does not receive such a message in a certain time period (ten minutes, by default) it assumes that a node in the ring has failed and attempts to kill and restart all `scored` processes. Note that the patrol mechanism does not allow `sc_watch` to identify the faulty node. Upon a failure, `sc_watch` knows that at least one node has failed, but it does not know which one.

In three months of monitoring a public installation of SCore v5.4 on a 129-node computational cluster at the Tokyo Institute of Technology, we witnessed several failures. To investigate the causes of such failures, we applied spTracer to collect function-level traces from `scored` processes on every node in the cluster. When each `scored` process starts, we inject our tracer agent into its address space and it starts collecting records of function calls and returns made by `scored`. When `sc_watch` times out waiting for a patrol message, spTracer saves the accumulated trace buffers to disk. Later, we analyze collected data to locate the failed nodes and identify the cause of the anomaly.

6.2 Network Stability Problem

The network link stability problem exhibited the following symptoms. The system stopped scheduling jobs, and `sc_watch` detected the failure after ten minutes and restarted the `scored` management daemons on all nodes without errors. The failure happened multiple times in two months, making it important to find its cause.

Our earliest last timestamp approach described in Section 4.1 determined that the failure exhibited a clear fail-stop behavior. We identified that host `n014` stopped generating trace records more than 500 seconds earlier than any other host in the cluster. We examined the last trace entry on host `n014` and found that `scored` terminated voluntarily by calling the `score_panic` function and eventually issuing the `_exit` system call. However, we could not find the caller of `score_panic` due to the fixed-size trace buffer. The entire buffer preceding `score_panic` was filled with calls to `myri2kIsSendStable`, evicting the common caller of `myri2kIsSendStable` and `score_panic` from the buffer. Future versions of our tracer will address this limitation by maintaining a call stack for the most recent trace record and reconstructing the call stacks for earlier records.

For the problem at hand, we used the source code to find that `myri2kIsSendStable` and `score_panic` were called from `freeze_sending`. This finding suggests that `scored` waited until there were no more in-flight messages on the host's Myrinet-2000 NIC.

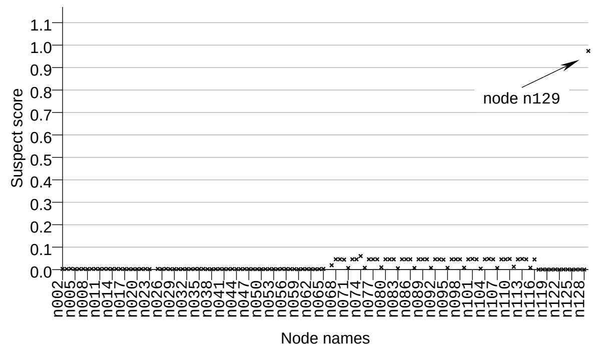

Figure 4: Suspect scores in the scbcast problem

When this condition did not occur after numerous checks, `scored` aborted by calling `_exit`. We reported the results of our analyses to the SCore developers. Their feedback confirmed our findings. They had observed such symptoms in Ethernet-based networks, but our report showed them that a similar problem exists in Myrinet networks.

6.3 Scbcast Problem

Another problem occurred when an SCore component, called `scbcast`, stopped responding to requests from `scored`. `Scbcast` is a broadcasting service in SCore that runs on the supervisor node. It aggregates monitoring information produced by `scored`, to which client programs can connect to retrieve this information rather than contacting individual `scored`s directly. While this technique eliminates some load from `scored` processes, it introduces an additional point of failure. In one execution, an `scbcast` process stopped responding to incoming requests and the entire SCore system stopped functioning. Here we describe how spTracer was able to establish that `scbcast` was the cause of that failure.

First, we decided that the problem did not exhibit a fail-stop behavior: the maximum difference between the last timestamps was only 20 seconds; the earliest host terminated less than a second earlier than the second earliest. Having identified that the problem was not fail-stop, we used our path-based ranking algorithms with results shown in Figure 4. For each point, the x-coordinate corresponds to the name of a node, and the y-coordinate to its suspect score. As we see, the suspect score of node `n129` is substantially higher than those of other nodes, making it a likely candidate for detailed examination. To obtain Figure 4, we used traces from the failed run, and also added reference traces from previous runs. The reference traces for previous faulty runs with unrelated symptoms proved especially useful as they allowed us to eliminate a second node, whose anomalous behavior was a response to more than one type of problem, and therefore could not point to a specific problem cause.

In contrast, the behavior of `n129` has not been seen in other executions. Our algorithm explained the cause of such behavior by identifying that the path (`output_job_status` → `score_write_short` → `score_write` → `__libc_write`) had the largest contribution to the node's suspect score. Of 12 minutes of execution, `scored` spent 11 minutes executing in `__libc_write` on that path. By visualizing the trace with the Jumpshot tool [5], we found that `scored` entered `score_write` and started calling `__libc_write` in a loop, never returning from `score_write`. As a result, `sc_watch` received no patrol message for more than 10 minutes, killed all `scored` processes, and tried to restart them. We see that `n129` was a true anomaly; our outlier-identification approaches were able to locate it, and the maximum-component approach found the correct symptom. Inspecting all traces manually would have required substantial effort.

Examination of the source code revealed that `scored` was blocking while trying to write a log message to a socket connected to the `scbcast` process that we did not trace. Such behavior would occur if `scbcast` froze and stopped reading from the socket [16]. If other `scored` nodes output a sufficient number of log messages, they would have also blocked. The fact that `scored` was not prepared to handle such failures points to a bug in its code. We reported the bug and it is being fixed for a future release of SCore.

7 Related work

Debugging of parallel and distributed applications has been an active area of research for several decades. Here, we survey only those approaches that perform some of the analysis automatically. The work of Dickinson et al. [11], Yuan et al. [32], Pinpoint [6], and Magpie [2] are most similar to our work. These approaches collect run-time information and use data mining techniques to simplify debugging of complex software. Unlike their *data collection* approaches that require static instrumentation, ours is dynamic. It can be activated and deactivated at any time and works on unmodified applications. Though we note that our dynamic approach could be applied to their systems. Below, we compare our *data analysis* approaches.

7.1 Dickinson et al.

In beta testing, the analysts examine execution reports coming from the users to determine the causes of failures. The work by Dickinson et al. [11] aims to reduce the number of reports an analyst has to inspect as follows. First, the users run instrumented programs to collect call profiles. Next, similar profiles are clustered together using several distance metrics. Finally, the analyst examines one or several profiles from each cluster to determine whether the entire cluster corresponds to correct or failed executions.

Similar to Dickinson et al., we operate on function profiles and use a distance metric to estimate similarity between profiles. Their clustering approach could also be applied to our goal of identifying anomalous behaviors: we could locate anomalies by finding clusters that contain one or a small number of nodes. In practice however, we found that this technique, like other unsupervised techniques, generates false positives reporting unusual but normal behaviors as anomalies. Although not a serious problem in their scenario that assumes manual data examination, false positives present a challenging obstacle for fully-automated online diagnosis.

Another difference between our approaches is that our method further automates problem diagnosis. After identifying an anomaly, our algorithm attempts to find a single function or a call chain that is the root cause of the problem (see Section 5 for details). In contrast, the approach of Dickinson et al. requires manual effort from the analyst to identify the cause of each problem.

7.2 Yuan et al.

Yuan et al. propose a technique for classifying failure reports from the field to quickly diagnose known problems. They collect a system call trace from an application that demonstrates a reproducible failure. Next, they apply a supervised classification algorithm to label the trace using its similarity to a collection of previous labeled traces. The label of the trace determines the type of observed problem, and therefore, the root cause of the problem.

Yuan et al. operate on system call traces that have coarser granularity than our function traces. In our experience, once a problem happens, the behavior of an application changes significantly, so that even system call traces may appear anomalous. This property is often used in anomaly-based intrusion detection (e.g., see [12, 30]). Although useful for detecting anomalies in system applications, it remains to be seen whether system call traces can accurately represent the behavior of other applications, such as parallel numeric codes. Moreover, system call traces may not be sufficient for finding many performance anomalies and their causes. In general, if the cause of a problem is in a function that does not make system calls, finding its location from system call traces may not be possible.

The approach of Yuan et al. would also work on our function-level traces. It could be applied to our problem at two different stages. First, it could be used to classify our function-level traces into two classes: normal and anomalous. A similar technique has also been used by Cohen et al. [9] to correlate aggregate metrics, such as system CPU utilization, with violations of performance objectives. Both approaches are supervised, requiring examples of previous traces to be labeled into normal and anomalous. In contrast, our approach is able to operate without any reference or with examples of only one class.

Second, the approach of Yuan et al. could be used at the root-cause identification stage to classify an anomalous trace into the most similar failure category. However, it would require manual effort from the analyst to classify traces for previous failures. Furthermore, this technique targets the root causes of known problems and would be unable to diagnose new failures.

Another difference between our approaches is that Yuan et al. use cross-run repetitiveness of an application's behavior. Traces from different runs (on the same or different machines) are compared to each other to identify root causes of failures. In contrast, our approach can use both the cross-run and within-run cross-process repetitiveness inherent in large-scale distributed systems. As a result, we can perform diagnosis in a single application run. Previous approaches have not explored within-run repetitiveness, as it is specific to large-scale environments.

7.3 Magpie

Magpie builds workload models in e-commerce request-based environments. Its techniques can be used for root cause identification as follows. First, Magpie collects event traces for client requests. It represents requests as strings of characters, where each character corresponds to an event, such as a context switch or an I/O event. Second, similar request strings are clustered together according to the Levenshtein string-edit distance metric [10]. In the end, strings that do not belong to any sufficiently large cluster are considered anomalous requests. Finally, to identify the root cause (the event that is responsible for the anomaly), Magpie builds a probabilistic state machine that accepts the collection of request strings [3]. Magpie processes each anomalous request string with the machine and identifies all transitions with sufficiently low probability. Events that correspond to such transitions are marked as the root causes of the anomaly.

Such data analysis techniques can be applied to our function-level data by representing function calls as Magpie's events. A key difference between our approaches is that Magpie groups raw traces based on the edit distance between them. We first summarize traces to compute profiles and then analyze similarity between the profiles. In our environment, application nodes perform similar activities in the long run, but their instantaneous behaviors can be radically different. The advantage of profiles in such situations is that they would still be similar while the difference between raw traces would be significant.

A disadvantage of profiles is that we may overlook subtle but real symptoms of a problem. In our experience however, the behavior of a node typically changes substantially upon a failure. The failed node and the location of the failure can then be easily identified from the profiles. Whether either approach can detect latent faults that occur before the failure remains to be seen.

7.4 Pinpoint

Pinpoint [6, 7, 8] detects faults in client-server systems and locates their root causes as follows. First, it records traces of components involved in processing each client request. Traces are obtained from application-specific or middleware-specific instrumentation present in many commercial systems. Second, Pinpoint determines whether each request completed successfully or failed. It looks for faults with known symptoms (e.g., network timeouts) using an auxiliary fault detector. It is also able to detect statistically significant deviations from the norm using PCFG (probabilistic context-free grammars). This approach is similar to that of Magpie, but Pinpoint applies it to fault detection rather than to root cause identification. Finally, Pinpoint uses two independent techniques, clustering and decision trees [8], to look for correlations between the presence of a component in a request and the failure of the request.

To identify an anomalous host, we could use Pinpoint's external fault detectors. However, they would not allow us to find silent problems, such as the one described in Section 6.3. Alternatively, we might be able to apply their PCFG-based approach. However, as discussed in Section 7.3, this approach may not be effective on raw function-level traces due to their variability.

To locate the root cause of a problem from classified traces, we could use Pinpoint's decision trees or clustering of coverage data. When applied to function-level coverage data, both approaches only would be able to detect a narrow class of problems. If a problem is not manifested by a difference in function coverage across traces, it will not be detected.

7.5 Other Diagnostic Approaches

Another approach for root cause identification is being investigated by the CBI (Cooperative Bug Isolation) project [15]. CBI instruments an application to collect the values of three types of predicates for each run: whether each conditional branch was taken or not, whether a function returned a negative value or not, and whether one scalar variable is greater than another. Further, each run is labeled as either failed or successful, depending on whether the application crashed or not. In the last step, CBI uses statistical techniques to analyze the collected data sets from numerous runs and identify predicates that are highly correlated with failed runs. Such predicates allow the analysts to focus their analysis on specific parts of the code.

If we introduce a different set of predicates that determine whether each function was present in the trace or not, the CBI analysis can be applied to data collected by spTracer. The key difference between our approaches is similar to that between spTracer and the first approach of Pinpoint: we do not rely on an externally-provided classification of traces into normal and anomalous and identify anomalies from the traces themselves.

Aside from the mentioned projects, there exists a variety of more specialized approaches that aim at locating a particular kind of problem. Since they do not attain the level of generality of approaches discussed above, we only survey them briefly. Several tools aim at detecting memory-related errors in applications [14, 21, 23]. Such tools can detect buffer overruns, memory leaks, attempts to use uninitialized memory, `free` operations on an already-freed memory region, and similar errors. There is also an extensive body of work on finding race conditions in multithreaded programs [21, 22, 25, 26]. These techniques monitor memory accesses as well as lock and unlock operations performed by different threads to make sure that all shared memory locations are guarded by locks.

8 Conclusion

We presented an automated approach that combines dynamic instrumentation and trace analysis for explaining failures in large-scale distributed environments. The approach looks for anomalies rather than massive failures as anomalies are often harder to locate than massive failures. We identify both fail-stop and non-fail-stop anomalous behaviors and further attempt to explain the cause of the anomaly. Evaluation of our prototype in a real-world distributed environment demonstrated the effectiveness of described techniques.

Acknowledgements

We wish to thank Somesh Jha for helpful discussions of root cause analysis techniques, Xiaojin (Jerry) Zhu for insight into machine learning approaches, Satoshi Matsuoka for valuable discussions and suggestions of failure diagnosis in clusters, and the Global Scientific Information and Computing Center, Tokyo Institute of Technology for allowing us to conduct the experiments on their production system.

This work is supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510, and Office of Naval Research grant N00014-01-1-0708, and in part by Japan Science and Technology Agency as a CREST research program entitled "Mega-Scale Computing Based on Low-Power Technology and Workload Modeling". The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] A. Avizienis, J.-C. Laprie and B. Randell, "Fundamental Concepts of Dependability", *Research Report N01145*, Laboratory for Analysis and Architecture of Systems (LAAS-CNRS), Apr. 2001.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for Request Extraction and Workload Modelling", in *6th Sym-*

posium on Operating Systems Design and Implementation (OSDI), San Francisco, CA, Dec. 2004.

[3] P. Barham, R. Isaacs, R. Mortier, D. Narayanan, "Magpie: real-time modelling and performance-aware systems", in *9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, May 2003.

[4] S.D. Bay and M. Schwabacher, "Mining distance-based outliers in near linear time with randomization and a simple pruning rule", in *9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington, D.C., Aug. 2003.

[5] A. Chan, D. Ashton, R. Lusk, W. Gropp, "Jumpshot-4 Users Guide", Mathematics and Computer Science Division, Argonne National Laboratory, http://www.mcs.anl.gov/perfvis/software/viewers/jumpshot-4/usersguide.html

[6] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox, "Pinpoint: Problem Determination in Large, Dynamic, Internet Services", in *International Conference on Dependable Systems and Networks*, Washington D.C., Jun. 2002.

[7] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based Failure and Evolution Management", in *1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[8] M. Chen, A.X. Zheng, M.I. Jordan, and E. Brewer, "Failure Diagnosis Using Decision Trees", in *International Conference on Autonomic Computing (ICAC)*, New York, NY, May 2004.

[9] I. Cohen, J. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control", in *6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[10] W.W. Cohen, P. Ravikumar, and S. Fienberg, "A Comparison of String Metrics for Matching Names and Records", in *KDD Workshop on Data Cleaning and Object Consolidation*, Washington D.C., Aug. 2003.

[11] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles", in *23rd International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, May 2001.

[12] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A sense of self for unix processes", in *IEEE Symposium on Security and Privacy*, Los Alamitos, CA, May 1996.

[13] T. Hastie, R. Tibshirani, J. Friedman, "The Elements of Statistical Learning: Data Mining, Inference, and Prediction", Springer-Verlag, 2001, ISBN 0-387-95284-5.

[14] R. Hasting and B. Joyce, "Purify: Fast detection of memory leaks and access errors", in *Winter Usenix Conference*, San Francisco, CA, Jan. 1992.

[15] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan, "Scalable Statistical Bug Isolation", in *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[16] Linux Manual Page, "send, sendto, sendmsg - send a message from a socket".

[17] M. Litzkow, M. Livny, and M. Mutka, "Condor—a hunter of idle workstations", in *8th International Conference on Distributed Computing Systems*, San Jose, CA, Jun. 1988.

[18] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tool", *IEEE Computer*, **28**, 11, Nov. 1995, pp. 37-46.

[19] D.L. Mills, "The network computer as precision timekeeper", Precision Time and Time Interval (PTTI) Applications and Planning Meeting, Reston VA, Dec. 1996.

[20] A.V. Mirgorodskiy and B.P. Miller, "Autonomous Analysis of Interactive Systems with Self-Propelled Instrumentation", in *12th Multimedia Computing and Networking (MMCN)*, San Jose, CA, Jan. 2005.

[21] N. Nethercote and J. Seward, "Valgrind: A program supervision framework", in *3rd Workshop on Runtime Verification (RV)*, Boulder, CO, Jul. 2003.

[22] R.H.B. Netzer and B.P. Miller, “Improving the Accuracy of Data Race Detection”, in *3rd ACM Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, Apr. 1991.

[23] B. Perens, “Electric Fence”, http://perens.com/FreeSoftware/

[24] S. Ramaswamy, R. Rastogi, and K. Shim, “Efficient algorithms for mining outliers from large data sets”, in *ACM SIGMOD International Conference on Management of Data*, Dallas, TX, May 2000.

[25] M. Ronsse, B. Stougie, J. Maebe, F. Cornelis, K. De Bosschere, “An Efficient Data Race Detector Backend for DIOTA”, in *International Conference on Parallel Computing (ParCo)*, Dresden, Germany, Sept. 2003.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multi-threaded programs”, in *16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.

[27] Y. Ishikawa, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, F. O’Carroll, and H. Harada, “RWC PC Cluster II and SCore Cluster System Software—High Performance Linux Cluster”, in *5th Annual Linux Expo*, Raleigh, NC, May 1999.

[28] D.M.J. Tax, “One-class classification”, PhD thesis, Delft University of Technology, http://www.ph.tn.tudelft.nl/ davidt/thesis.pdf, Jun. 2001.

[29] R.F. Van der Wijngaart, “NAS Parallel Benchmarks Version 2.4”, *NAS Technical Report NAS-02-007*, Oct. 2002.

[30] D. Wagner and D. Dean, “Intrusion Detection via Static Analysis”, in *IEEE Symposium on Security and Privacy*, Washington, D.C., May 2001.

[31] R. Wismuller, J. Trinitis, and T. Ludwig, “OCM—A Monitoring System for Interoperable Tools”, in *SIGMETRICS Symposium on Parallel and Distributed Tools*, Welches, OR, Aug. 1998.

[32] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, W.-Y. Ma, “Automated Known Problem Diagnosis with Event Traces”, *Microsoft Research Technical Report MSR-TR-2005-81*, Jun. 2005.

[33] V. Zandy, “Force a Process to Load a Library”, http://www.cs.wisc.edu/∼zandy/p/hijack.c