Automated Fault Localization in Large-Scale Computing Systems

by

Naoya Maruyama
naoya.maruyama@is.titech.ac.jp

Submitted to the
Department of Mathematical and Computing Sciences,
Graduate School of Information Science and Engineering,
Tokyo Institute of Technology

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

March 2008

Abstract

This dissertation presents two scalable, automated approaches to simplifying fault localization in large-scale computing systems that view localization as anomaly detection in system behaviors. Both approaches always capture system behaviors by obtaining function call traces, and identify anomalous behaviors through automatic data analysis of the collected traces. To find anomalies in scalably and automatically, they assume processes in typical distributed software systems have behavioral similarities, and find violations in the assumed similarities as anomalies.

The first approach, *outlier-detection-based localization*, localizes faults by assuming that the target system consists of distributed processes with similar behaviors. Specifically, once a failure occurs, it identifies anomalous processes and functions by comparing the failure traces and finding outliers among them. Traces are compared by using their function-execution times. By finding outliers based on these times, this approach can localize faults such as performance bugs, deadlocks, and livelocks.

The second approach, *model-based localization*, localizes faults by assuming that all processes exhibit similar behaviors to those observed in the past. By using traces collected during normal operations, it derives an execution model that estimates the call probability of each function. Once a failure occurs, it finds anomalous processes and function calls by comparing the failure traces against the derived model. We consider the following cases anomalous when: 1) high-probability functions are not called, and 2) low-probability functions are called. This approach is especially effective in localizing program logic bugs by finding these functions.

Experimental studies done on real-world large-scale environments indicate the effectiveness of the proposed techniques. Our outlier-detection-based localization almost automatically found the causes of several nondeterministic failures in a distributed cluster middleware running on a 129-node production cluster. The model-based localization also substantially simplified the localization process of a failure that occurred in a three-site, 78-node Grid environment.

Acknowledgments

This dissertation would not have been possible without the expensive support that I received from various individuals in both research and non-research matters. Although I cannot possibly acknowledge everyone that have contributed, I can extend my special thanks to a few.

First, I would like to thank my dissertation advisor Professor Satoshi Matsuoka. His timely and effective mentoring played an immense role in enabling me to pursue my research goals and establish my career as a researcher in computer science. Throughout the years I have spent as a graduate student within his research group, he has always been supportive and encouraging and given me numerous opportunities of interacting with prominent researchers in various fields.

The other members of the dissertation committee, Professors Masataka Sassa, Etsuya Shibayama, Osamu Watanabe, and Shigeru Chiba, provided me with insightful comments and questions on my research. Their feedback enabled me to think about my research from various different perspectives, and to improve the overall quality of this dissertation.

I have also been fortunate to have had many gifted graduate students as collaborators and friends. Many thanks go to the former and present members of my research group at the Tokyo Institute of Technology, especially, Hirotaka Ogawa, Fuyuhiko Maruyama, Hidemoto Nakada, Toyotaro Suzumura, Satoshi Shirasuna, Hiroyuki Komatsu, Yoshiaki Sakae, Hideyuki Jitsumoto, and Shinichiro Takizawa. They made my graduate student years much more interesting as well as enjoyable.

I have also had many experiences with interacting and working with various researchers and students. My first research collaboration was with Professor Hidehiko Masuhara of the University of Tokyo. Through his guidance, I first learned about how a computer scientist should think in pursuing solutions to intellectually challenging problems. I also enjoyed discussing various topics with Professor Shigeru Chiba of the Tokyo Institute of Technology. I have been repeatedly impressed by his insightful comments on a wide variety of research topics.

Professor Bart Miller and his computer sciences research group at the University of Wisconsin, Madison kindly hosted me as a visiting graduate student from September 2004 for one year (Thanks again to my dissertation advisor for giving me such an invaluable opportunity). It was then that I was first exposed to international research environments. Without such experiences, my interests and career directions would have been more insular and confined to domestic research opportunities. In particular, I have truly enjoyed collaborating with Alex Mirgorodskiy. I was impressed with his deep understanding of low-level programming techniques and insights into system software, and admire his "magic" dynamic instrumentation technique. The first part of the contributions of this research is from my collaboration with him and Professor Miller. My life in Madison, which was my first long stay in the USA, was also supported by my many friends there, especially Eli Collins, Matt Legendre, Jaime Frey, and Hao Wang. I continue to miss the beautiful lakes, gorgeous summer weather, as well as the freezing winter weather, the delicious Babcock ice cream, and Madison's academic atmosphere.

I also received support from the CREST Mega-Scale project in both research assistance and funding. The project members, especially Professors Hiroshi Nakashima, Hiroshi Nakamura, Taisuke Boku, and Mitsuhisa Sato, provided me with a wide variety of discerning suggestions in my discussions with them throughout this research.

Case studies using real-world production systems were extremely valuable for assessing the effectiveness of our proposals, and their results actually allowed us to gain useful insights. The Global Scientific Information and Computing Center (GSIC) at the Tokyo Institute of Technology kindly allowed us to apply outlier-detection-based fault localization to their SCore cluster environment. The InTrigger distributed computing platform by the Info-Plosion project also enabled us to evaluate our model-based fault localization. I wish to thank the InTrigger development team, especially Professor Kenjiro Taura, Hideo Saito, and Yoshikazu Kamoshida for their efforts in developing such a large-scale playground for computer science research. Cooperation with both GSIC and the InTrigger team has greatly strengthened the impact of this dissertation.

Last, but not least, I wish to thank my family for their unconditional support. My parents are likely to know little about what I have done in this research; nonetheless, they have unreservedly believed in me throughout my prolonged study. For the last three years, I have also received the highest degree of support from my wife, Akiko, in almost everything except the research itself. Without her continuous encouragement and belief in me, my final and most difficult years in completing this dissertation would have been much more daunting and less stimulating. I am genuinely grateful to her for her understanding and encouragement throughout.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

System failures have been one of the biggest obstacles in operating today's large-scale computing systems. *Fault localization*, i.e., identifying direct or indirect causes of failures, is a complicated and time-consuming problem, and thus automating such cause identification processes represents an important research challenge. This dissertation shows that anomaly-detection-based techniques can automatically localize important classes of faults in large-scale systems, allowing problem analysts to focus on significantly smaller parts of the overall system.

1.1 Difficulty in Fault Localization in Large-Scale Computing Systems

Fault localization in large-scale computing systems, such as clusters and Grids, is a notoriously difficult problem to solve. The two main reasons for this difficulty are:

- *Scalability barriers imposed by ever increasing scale of modern HPC systems and*
- *Rare, nondeterministic faults magnified by diverse system configurations using commoditized software and hardware components.*

Finding the root cause of a particular failure is becoming more challenging as system scale increases. The TOP500 list as of November 2007[1] shows that 86% of total systems ranked on the list have more than a thousand processors. For example, the number of processors in the world's fastest supercomputer on the list, the IBM Blue Gene/L at Lawrence Livermore National Laboratory (LLNL),

[1] http://www.top500.org/stats/list/30/procclass

has reached hundreds of thousands. The consequences of this trend are twofold. First, most conventional methods in these large-scale systems, such as examining crash dumps or various system logs, or attaching to faulty systems with interactive debuggers, are not always feasible in practice because they inevitably require manual input. A crash dump can provide a snapshot of system behaviors immediately before a system crashes, and can provide useful information on system diagnosis on a single machine. However, on clusters with thousands of machines, where the number of crash dumps would also be in the thousands, simply having such a large amount of data would be too overwhelming to extract any useful information. Logs could reveal the history of critical system behaviors, but, as is the case with crash dumps, they alone do not always provide sufficient visibility into system behaviors for localizing faults. Also, interactive debuggers represent a proven technology as effective debugging tools for a single process; even with multiple processes, debuggers with a parallel extension allow multiple processes to be introspected simultaneously as in single-process debugging [54, 90]. However, while they are indeed effective in examining particular program states, they are of no help to analysts in finding where they should examine faults. Thus, finding their root cause in such a large number of processes remains a difficult problem. In fact, Oliner et al. reported that root causes of approximately 17% of failures were not determined in the Blue Gene/L [66]. Second, timing-related bugs, more likely to occur as the scales of systems increase, can preclude the use of these manual approaches due to their non-negligible perturbations.

Allowing diverse system configurations also presents several difficult challenges to software reliability [34, 61]. First, it makes standard engineering disciplines such as in-house pre-release testing less effective. Since real production systems are too varied, it is unlikely to have all the same testing environments as diverse as real production systems. Thus, even if a software component works flawlessly within testing environments, this does not necessarily mean that it will work as well on production systems. For example, as discussed in Chapter 6, a popular parallel-programming library was observed to fail to operate in a Grid environment, whereas this problem did not occur while running within a single cluster. Such environment-dependent faults are hard to eliminate with conventional pre-release in-house testing because of the difficulty of reproducing them.

Second, such faults need to be localized in particular settings due to their dependence on these specific environments. However, in reality, the actual administration policies in these environments might preclude such time-consuming post-mortem analysis as that involved in manual diagnosis with an interactive debugger, since rapid recovery is typically preferred by merely restarting production systems to thoroughly diagnose their faults, which requires extra downtime. As such, analyzing crash dumps or various system logs post-mortem is typically the only possible approach in such environments; however, as previously stated, simply having such a voluminous amount of data does not necessarily lead to rapid fault localization.

Several trace-based techniques of detecting anomalies have been proposed to tackle these problems with fault localization [8, 17, 42, 60, 71]. However, the main limitation with these approaches has been that they employ a centralized algorithm to find anomalies, assuming that traces distributed over remote machines can be collected by a centralized trace analyzer with no limits in scalability. For example, Magpie identifies correlated system events occurring in distributed nodes using several manual and automatic techniques, and allows one to understand high-level control and data flows spread across distributed nodes [8]. Although such a centralized approach could allow us to comprehend the distributed system as a whole, and not just a mere collection of uncorrelated multiple processes [8, 17, 42, 60, 71], their application to the previously mentioned large-scale current HPC clusters is still unclear.

1.2 Approach

To simplify fault localization in large-scale computing environments, this dissertation proposes two scalable, automated techniques of fault localization where localization is regarded as the detection of anomalies in system behaviors. Both the outlier-detection-based and model-based techniques capture system behaviors by always recording the function call traces of the system being observed, and identify anomalous behaviors by automatically analyzing data from the collected traces. Always-on tracing allows system behaviors, including those that are rare and nondeterministic, to be recorded. The key challenge in achieving such trace-based fault localization, therefore, is how to develop a scalable, automated technique of detecting anomalies that can effectively localize faults in large-scale environments.

To meet this challenge, we use a *behavioral rule* that holds during normal operations, but does not when failures occur. Once a failure occurs, we find behaviors violating the rule by automated data analysis of function traces, and identify anomalous symptoms correlated with the failure. Furthermore, to achieve scalability, we design the data analysis algorithm so that it can operate on the scale of current and future large-scale computing systems.

This dissertation presents two derivations of this general approach. Specifically, for behavioral rules, we exploit two observations in the behaviors of typical distributed software for large-scale systems, i.e., *spatial* and *historical similarities*.

Spatial similarities Typical distributed software for clusters and Grids, such as job schedulers and parallel file systems, consist of a small number of collections of identical programs running on each member node. Any software that employs a master-worker architecture should consist of two identical process groups, i.e., the master and the rest of the processes. Even software with a more complicated hierarchical structure should also have a small number of process groups.

Historical similarities Many distributed systems consist of member processes that exhibit similar execution behaviors to themselves in the past. The reason behind this is that they typically employ event-driven architectures, where several different event-processing routines are multiplexed into a single event loop. For example, a batch-job scheduler for clusters could employ a daemon process on each node whose responsibilities include monitoring jobs under the node and handling requests from the master job scheduler. A typical software architecture for such purposes would model job-status changes and incoming requests from the master as events, and consist of an infinite loop of event-processing routines. Thus, we could expect that member processes in the long run would exhibit similar behaviors to themselves in the past by running the same routines in an infinite loop.

Based on the observation of spatial similarities, we propose the first localization technique, *outlier-detection-based fault localization*, which is specifically targeted to cluster-type distributed systems [58]. It uses the observation of spatial similarities as a behavioral rule, and considers rare behaviors that are different from the majority, thus violating the observation, as anomalies. The second proposed technique, *model-based fault localization*, assumes the observation on historical similarities of processes in distributed systems [52]. It considers behaviors that are not observed before, thus violating the observation, as anomalies. Note that both techniques share the same principle: they obtain function call traces from running systems, and attempt to identify anomalies by finding violations against their respective behavioral rule. The rest of this section overviews both methods.

1.2.1 Outlier-Detection Based Fault Localization

The first method assumes spatial similarities are being observed, which allows us to view the localization problem as an outlier-detection issue. That is, we localize anomalous processes and functions by finding a process or a small group of processes that exhibits significant deviations from the rest of them. More specifically, outlier-detection-based fault localization consists of two steps. First, it executes process-level localization that finds an outlier process by comparing the per-process function traces of the failed run. Second, it applies function-level localization to the trace of the identified process to locate the most anomalous function.

This dissertation presents techniques of comparing traces for both *fail-stop anomalies* and for *non-fail-stop anomalies*. Fail-stop anomalies are a common case of faults in distributed environments, where an anomalous process stops execution prematurely while the others are still running. For example, such cases as infinite blocking on system calls and termination due to illegal operations would identify themselves as fail-stop anomalies. We find such anomalies by detecting the earliest last timestamp in the per-process traces of the failed run. While this technique is very simple, we demonstrate its effectiveness through a case study with a real production system in Chapter 5. For more complex, non-fail-stop anomalies, we use the following technique of trace comparison.

For non-fail-stop, latent anomalies, we find them by locating behavioral outliers, i.e., processes that have substantial differences in function-call traces compared to others. To do so, we apply a *distance-based outlier-detection* technique [9], where the pair-wise distance is defined by the difference in the execution times of functions. For a given pair of per-process traces, we give a long distance when one of the pair takes substantially longer to execute a certain function, or, even longer when they execute different functions. Finally, based on the distance definition, we identify outlier processes by finding a trace or a small group of traces that is significantly far from its k^{th} nearest neighbor.

Once an anomalous process is found, we find the functions that best explain its outlierness to further assist the problem analyst in localizing the fault. Specifically, for the fail-stop case, we report to the problem analyst the function of the last trace entry of the anomalous process. For a non-fail-stop case, we rank the functions appearing in the trace according to their contribution to the distance to the k^{th} nearest neighbor, and report the highest-ranked function to the problem analyst. In other words, the function with the largest difference in its execution time between the anomalous process and its k^{th} nearest neighbor is considered to be the most suspicious.

Unlike the second approach described below, this localization approach does not require prior learning of the normal execution behaviors of the target system. Instead, it only performs post-mortem analysis of the traces of the failed run with no *a priori* knowledge on the target system (i.e., *unsupervised analysis* [85]). However, if we already have prior traces collected from previous runs, we can improve localization accuracy by excluding outlying but legitimate processes from further localization (i.e., *one-class analysis* [87]). We show its effectiveness through several case studies with a large real-world production cluster.

1.2.2 Model-Based Fault Localization

Based on the observation of historical similarities, the second approach localizes faults by modeling normal execution behaviors and detecting violations in fault traces against the model. Specifically, we first learn per-process function-calling behaviors, and generate a concise per-process model called a *process model*. Next, we aggregate all process models into a single *global model* that reflects the behaviors of the overall system, and deploy to all nodes of the target system. Finally, once a failure occurs, we detect anomalies in the failure traces by comparing them with the global model, and report the detected anomalies to the problem analyst for further analysis of the root cause.

We focus on a class of distributed systems that employ an event-driven architecture, and design an accurate model specifically targeted to the class. To derive the process model, all the function traces are first divided into sub-traces, or *execution units*, based on their associated event sources, and then a different model is derived for each event source. Of various types of events, this dissertation focuses on network events, which we believe would be the most important events in distributed computing systems. The source of network events is their connection; thus we treat a sequence of function calls corresponding to the same network connection as a single execution unit. Next, we derive a model for each connection by constructing a call tree for every function appearing in its associated units, and assigning each function an estimated probability of occurrence. We estimate the probability of a function by dividing the number of its occurrences by the total number of the same events. For instance, if a function always appears when a message arrives at a connection, we give a function probability of 1. Note that creating separate models for different connections can improve the accuracy of estimating the probability, since different connections are likely to have different function-calling behaviors. Finally, we derive the global model by merging the process models whose processes are inferred to have played the same role in the system.

Once a failure occurs, we assist human analysts by localizing the root cause of the failure by comparing its failure traces with the derived model. Given failure traces, we first divide them into execution units as we did in deriving the model. For each execution unit, we compute a *suspect score* that quantifies how likely a particular part of call traces is correlated with the failure. Functions with a small probability in the model yield high suspect scores when called. Frequently called functions in the model also yield high scores when not called. Finally, we report the execution units annotated and sorted with the suspect scores to the human analysts, assisting them to narrow down the localization process to a small part of the entire system.

Note that, since only locally observable information is used, i.e., function-call traces, both modeling and detection of anomalies can mostly be decentralized, allowing good scalability to be achieved with increasing system scale. Specifically, our modeling requires globally coordinated operations only when gathering the process models to generate the global model. Fault localization using the global model can be performed locally by deploying the model to all the nodes in advance. As previously discussed, this decentralized approach differs from many of the previous approaches that reconstruct distributed flows by matching distributed logs and traces [8, 17, 42, 60, 71].

The observation behind the approach is that many anomalies manifesting themselves over distributed nodes still exhibit *locally observable deviant behaviors* as well. For instance, a bug discussed by Reynolds et al. prevented an event-handler function from being called, failing to serve incoming requests [71]. Also, the Tokyo Stock Exchange, one of the largest stock exchange markets in the world, suffered from a severe service outage from February 8, 2008 for three consecutive days [89]. It was reportedly caused because a certain part of newly allocated memory was not initialized. While we do not know the internal logic of the memory initialization, it is likely that a certain function related to initializing the memory area was not called only at the failed case. These behaviors could be detected by only using locally observable information, i.e., function-calling behaviors in these particular examples.

1.3 Contributions

This thesis presents several contributions towards automated fault localization in large-scale computing environments. Among them, the primary contributions are as follows:

- We propose two automated fault localization approaches based on anomaly detection in per-process function call traces. The outlier-detection based approach uses an unsupervised, distance-based outlier-detection technique to find anomalous processes and functions in the target system. We also present an extension of the technique that exploits past knowledge on system behaviors for more accurate and precise localization. The model-based approach further extends the use of past data: It derives a concise execution model that reflects normal system behaviors using past traces collected during the system operates correctly. When a failure happens, it compares the traces of the failed run against the derived model and automatically finds anomalies by detecting deviant behaviors in the failed run. Both approaches are based on their respective similarity assumption to detect anomalies, i.e., the spatial similarity for the outlier-detection based approach and the historical similarity for the model-based approach. We present discussions on the difference of the two approaches in Chapter 7.

- We show the effectiveness of our proposals through several experimental studies that apply them to failures in large-scale computing environments. Our outlier-detection based localization almost automatically found the cause of several nondeterministic failures in a distributed cluster middleware running on a 129-node production cluster. The model-based localization also significantly reduced a localization process of a fault that often, but not always, happens in a three-site, 78-node Grid environment.

1.4 Dissertation Outline

This dissertation is divided into eight chapters. The rest of it is structured as follows:

Chapter 2: Background To set the context, we first provide definitions of key terms used throughout this dissertation, and describe target environments, i.e., large-scale computing environments, and typical faults in such environments.

Chapter 3: Related work Next, we review related studies in such areas as fault localization, fault detection, statistical techniques for detecting anomalies, and performance debugging. We specifically focus on issues of the previous approaches with respect to scalability and automation. We also review approaches similar to ours, but applied to different problem domains such as viruses and intrusion detections.

Chapter 4: High-level overview of proposed approaches To tackle the problems stated in this chapter, we propose two approaches to fault localization. This chapter presents the underlying ideas and observations behind the approaches.

Chapter 5: Outlier-detection-based fault localization The first fault-localization approach is based on outlier detection in the member processes of the system being observed. This chapter describes the algorithm and experimental evaluation of the outlier-based fault localization.

Chapter 6: Model-based fault localization The second approach to localization derives a model reflecting the execution behaviors of the target system, and localizes faults by detecting violations of the derived model in failure traces. This chapter presents the algorithm and an experimental evaluation with real-world faults in distributed environments.

Chapter 7: Discussion Next, we discuss what we have learned through research on this dissertation, including the limitations of the approaches and possible solutions to them. We also compare the proposed approaches in terms of their applicabilities and limitations.

Chapter 8: Conclusions Finally, we summarize the contributions made by this dissertation and discuss possible directions for future research.

Chapter 2

Background

This chapter sets the context of our research by presenting the definitions of several key terms and introducing target environments, i.e., modern large-scale computing systems. Finally, we review several examples of faults occurring in various large-scale environments reported in the literature to enable a better understanding of the characteristics of faults.

2.1 Definitions of Faults, Failures, Errors, and Fault Localization

This dissertation refers to several standard terms in the field of dependable computing systems [6, 82]. We briefly introduce their definitions to avoid confusion in the rest of this dissertation. For more complete explanations, refer to Avizienis et al. [6] and Steinder and Sethi [82], among others.

Failures in system components are states that occur when they generate externally visible incorrect output [6]. *Errors* denote a class of system states that cause a discrepancy between a real computed output and a specified correct output. An error can cause subsequent errors in the same component, or a failure, if it manifests itself as visible to outside components. An error is initially caused by a *fault* within the same component or external input. A *causality path* is a directed path from a fault to errors to a failure.

For example, consider a situation where a Web server occasionally fails to return correct responses due to a coding bug. Here, the state failing to return correct responses is a failure of the Web server; the bug is the cause of the failure, thus it is a fault. The bug might cause different discrepancies in the Web server other than dropping user requests; these are errors caused by the bug. Note that the Web server does not always fail to operate even if a fault is present in the code. Such faults are called *dormant*; they are present in a system but do not manifest themselves to the system. In the Web-server example, if the bug is not exercised by incoming requests, the system exhibits no failures; thus the bug is dormant. When it actually causes a failure, it becomes *active*.

Fault localization is a term that refers to the process of deducing the exact source of a failure from the set of observed failure indications [82]. In other words, it identifies the causality path from a fault to its failure using observed failure symptoms. Note that we use the term in a weaker sense, i.e., narrowing the causality path to a small set of possible locations. Other closely related terms used in this dissertation are *fault detection* and *failure detection*. The former refers to the process of capturing unintended behaviors that may or may not lead to errors and failures; the latter refers to the detection of occurrences of externally visible faults. Fault localization itself does not detect failures, but only attempts to identify possible causes of a given failure that is detected by a failure detection mechanism. This dissertation explores techniques of simplifying fault localization for given failures while minimizing the human effort involved.

2.2 Today's Large-Scale Computing Platforms for HPC

Large-scale supercomputers are the most common platforms for high-performance computing. As can be seen from the TOP500 lists [1], the trend toward improving supercomputing performance has been to increase the number of processors by employing a scalable system architecture, such as through clustering and massively parallel processing (MPP). Figure 2.2 traces the development of average and maximum processor counts of machines ranked in the TOP500 lists from 2000 to 2007. The average number of processors increased approximately 14 fold, from 231 in June 2000 to 3296 in November 2007; the maximum number increased approximately 20 fold, from 9632 to 212992.

Cluster supercomputers are the current most common architecture. In fact, more than 80% of machines ranked in the TOP500 list as of November 2007 are clusters. One example of the largest clusters is the TSUBAME supercomputer at the Tokyo Institute of Technology [24]. It consists of 655 Sun Fire X4600 nodes, each of which is equipped with eight AMD Opteron dual-core CPUs, resulting in 10,480 cores in total. The nodes are interconnected with Voltair Infiniband networks, and are further powered by ClearSpeed CSX600 accelerator boards. The supercomputer also supports high-performance access to 1 PB of storage with the Lustre parallel file system [18]. It uses a mixture of various standard-based software components, including, among others, SuSE Enterprise Linux as a com-

[1] http://www.top500.org/

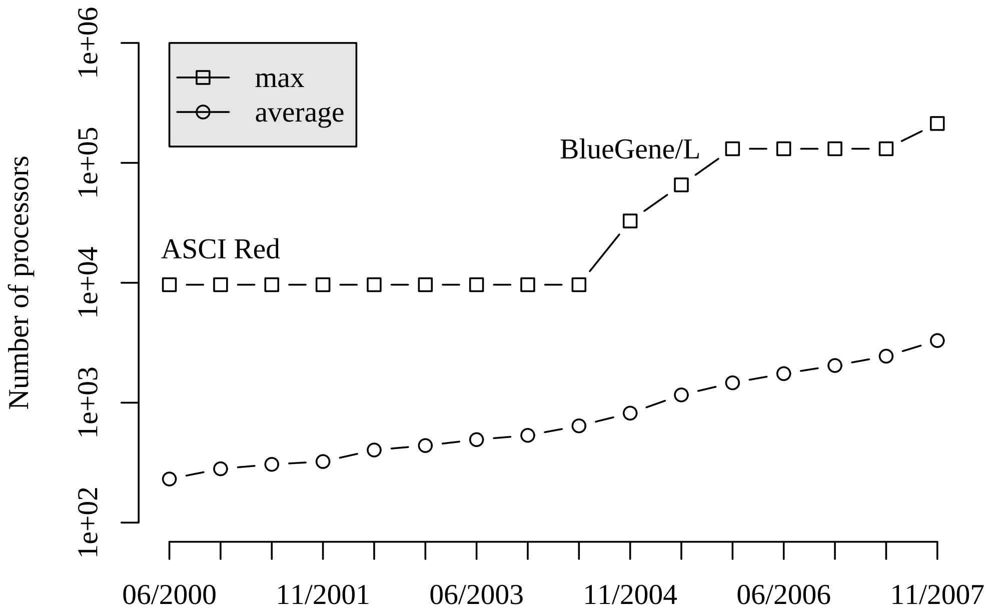


Figure 2.1: Average and maximum processor counts of machines ranked in TOP500 lists from June 2000 to November 2007.

pute node OS, RedHat Enterprise Linux as a storage node OS, Lustre parallel file system for scalable high-performance storage, Sun Grid Engine for distributed resource management, and various other ISV applications. This multi-vendor hardware and software organization exemplifies a commoditized component architecture that can complicate the diagnosis of problems in such dynamic environments.

Massively parallel processing systems (MPPs) are also a popular supercomputing architecture that are typically more powerful than the abovementioned cluster supercomputers. The Blue Gene/L supercomputer by IBM at LLNL consists of more than a hundred thousand nodes, and was the fastest supercomputer in the Top500 supercomputer list as of December 2007 [25]. The Earth Simulator consists of 640 compute nodes, each of which is equipped with eight vector processors, reaching 5,120 vector processors in total [80]. It had been the fastest supercomputer from 2002 to 2004 on the TOP500 list by achieving 35 GFlops [2].

Grid computing is another viable large-scale computing platform. It allows heterogeneous, geographically distributed computing resources to be used se-

[2] http://www.top500.org/system/ranking/5628

curely and in coordination. For example, InTrigger provides a large-scale computing platform for researchers in various fields. It consists of more than 300 compute nodes distributed over six Japanese universities and laboratories (as of December 2007) with further extensions planned [75]. Other examples include Grid'5000, DAS-3, Grid3, and EGEE. A French project called Grid'5000 has been providing a large-scale Grid computing platform consisting of a collection of computing clusters distributed across nine French universities and laboratories, featuring more than 3000 processors in total as of December 2007 [14]. A unique characteristic of Grid'5000 is that users have complete control of compute nodes from their operating systems to the application software by having had a scalable remote mechanism for system installation provided to them. DAS-3 is another Grid infrastructure distributed over five sites in The Netherlands, providing 272 compute nodes in total [67]. Furthermore, DAS-3 and Grid'5000 are now being interconnected to provide a more powerful larger-scale platform. Grid3 is a US Grid infrastructure consisting of more than 30 sites and 4500 CPUs. It provides large-scale computing resources mainly for physics experiments [23]. The EGEE project also provides a very large-scale Grid infrastructure, which consists of 41,000 CPUs distributed over 240 sites across 45 countries in Europe and other regions [29].

Most of these supercomputers and Grid infrastructures share the same reasons for difficulties with fault localization, i.e., system scale and diversity. The prevalence of these large-scale platforms makes automated fault localization more important and even essential to achieve reliable and dependable computing with them.

2.3 Fault Characteristics in Large-Scale Computing Systems for HPC

Gray reported on failures that affected Tandem fault-tolerant systems [33]. He roughly classified failures into four categories: administration, software, hardware, and the environment, where their ratios corresponded to 42%, 25%, 18%, and 14%. He suggested that the most important area for improving system availability was to reduce administrative mistakes, while hardware failures could be mitigated through redundancy as is done in Tandem systems. Although his survey dates back 20 years, the situation in today's computing systems has not differed much. Administrative mistakes can even be more problematic since current systems are much more complex from both hardware and software perspectives. Furthermore, large-scale systems for HPC, such as supercomputers, usually do not benefit from hardware fault tolerance mechanisms such as redundant compute nodes due to increased cost; thus, hardware can be much more problematic. The rest of this section reviews some published studies on faults occurring in large-scale environments.

Schroeder et al. reported on failures that were manually recorded over nine years in 20 different systems at LLNL, and categorized the root causes into six factors: human, environment, network, software, hardware, and unknown [77]. The number of nodes in the systems that they studied varied from just one to 1024, with the number of processors ranging from four to 6152. In their classification, hardware was the largest source of failures, ranging from 30% to more than 60% depending on what systems were studied, followed by software (5% to 24%). Another noteworthy finding was that the ratio of failures whose cause was not identified ranged from 20% to 30%, close to the ratio of software failures but less than that for hardware. Such a high ratio illustrates the difficulty of localizing faults in large-scale systems. They also reported that failure rates varied significantly across systems, ranging from only 17 failures per year to an average of 1159 failures per year, and that these rates were approximately proportional to the size of each system.

Liang et al. investigated failures observed in the Blue Gene/L at LLNL for 100 days [46]. Their findings included temporal correlations of memory-related failures and spatial correlations of network-related failures, e.g., an intermediate network switch failing to function would cause all child nodes connected to other nodes with the switch to be unreachable. They presented techniques of predicting occurrences of failures by exploiting these correlations. While the techniques presented this dissertation generally focus on software faults, such correlations would also be useful for fault localization.

Oliner et al. also reported on failures occurring in five supercomputers, including Thunderbird, Red Storm, Spirit, and Liberty, which are located at Sandia National Laboratories (SNL), and the Blue Gene/L at LLNL [66]. The machines differ greatly in process counts, ranging from Liberty's 512 to Blue Gene/L's 131072. They collected standard system logs that were available by default in these systems for durations ranging from 104 to 558 days, and classified the alerts into three types of hardware, software, and indeterminate. In contrast to the study by Schroeder et al., the most common category was software (64%), followed by hardware and indeterminate, both of these accounted for approximately 17% of failures approximately. This could be because of the difference in log collection methods; the data studied by Schroeder et al. were in a database that collected failures reported manually by system administrators, while the data studied by Oliner et al. were from computer-generated logs. However, both of them reported that approximately 20% of failures had root causes that were unknown. They also suggested that the system logs that they studied did not always contain sufficient information for localizing faults, which supports and motivates the need for collecting richer, more detailed information such as function call traces.

Faults occurring in Grid environments have also been studied and published in the literature. Kola et al. reported on faults observed in a Grid computing platform at the University of Wisconsin-Madison [43], which uses idle computers distributed across the campus, running the Condor distributed resource management system [49]. Various anomalies involving both software and hardware caused the observed faults, including corrupted data due to faulty hardware, hanging processes, misleading return values, misbehaving machines, and unavailable resources. They found the cause of hanging processes was related to an NFS operation, but the exact cause is still unknown. An application was found to return values indicating success even when it actually failed. Misbehaving machines accepted jobs but did not completed or return the results. Unavailablity of resources, such as intermittent network outages, also caused failures in jobs scheduled to use resources that became unavailable. They reported that most of these faults took a long time to diagnose due to the scale and distribution of the platform.

Another study on faults in Grid environments was presented by Medeiros et al. [53]. They reported the results of surveys given to actual Grid platform users and operators asking about experiences with and management of failures. The survey found that configuration errors due to operator mistakes were the most frequent causes of failures, while the second-most frequent causes of failures were software bugs in middleware and user applications, followed by hardware faults. These results are consistent with the studies on supercomputer faults previously discussed. Another interesting answer in the surveys was that the most difficult problem in fault management was localizing them, which supports the problem statement and motivates our proposal for automated fault localization.

It can be helpful to know what effects these faults have on the observable states of applications to localize those occurring in these environments. If anomalous behaviors similar to the known effects of a fault are observed, the cause of failure could be narrowed down to the fault. Lu et al. studied what effects transient software and hardware memory errors had on MPI applications [21]. To study such effects, they injected faults into application programs using MPICH, a popular MPI implementation. They injected bit flips into the memory area of user applications, and modified receive packet payloads by trapping the recv system call. Their overall results revealed that the applications studied were largely affected by the injected faults, including incorrect outputs, application crashes, and hanging applications. Thus, such anomalous behaviors could be inferred as having been caused by corrupt memory data. Their technique can be classified as a supervised method of learning to build an expert system for localizing known faults, while our techniques are more oriented toward localizing unknown faults.

In summary, we can see that faults do occur more frequently in larger-scale systems and finding the root cause of a fault is indeed an important issue, as indicated by the ratio of undetermined failures and the surveys obatained by Medeiros et al. These observations motivate the need for more advanced techniques of localizing faults than the standard approaches currently employed such as manual `printf` debugging.

Chapter 3

Related Work

This chapter describes past research efforts related to this work. We first introduce past approaches on fault localization in distributed systems in Sections 3.1 and 3.2. The former describes the differences in automated approaches and the latter in the other approaches, especially focusing on scalability and automation of analysis. Other key aspects of our research include efficient and effective recording of system behaviors and scalable analysis of recorded behaviors. We introduce research on both areas in Sections 3.3 and 3.4. Although the target problem domain is the distributed computing area, this research is also related to fault localization in sequential programs; we review past projects on that research area in Section 3.5. Finally, we survey data-analysis-based approaches in other problem domains, such as intrusion detection, in Section 3.6. We conclude this chapter by summarizing the related work in Section 3.7.

3.1 Approaches Based on Automated Data Analysis

3.1.1 Approaches Based on Flow Analysis

Identifying distributed control and data flows is a common approach in the trace-based automated performance and logic debugging, and thus many flow-based approaches have been proposed. They, however, have differed in their specific algorithms for doing so; we will first review several such approaches, and describe the key differences between our approaches and theirs.

MagPie by Barham et al. recovers distributed flows by using a user-written application-specific event schema [8] and an instrumented OS kernel and other middleware layers. Their schema is a set of rules to separate and join individual events observed in distributed components. To find anomalies in the recovered flows, they use cluster analysis [40], where the distance of a pair of flows is defined as their string-edit distance in the textual representation of the flows and the difference in resource usage.

Kiciman and Fox and Chen et al. presented a fault management framework that can detect failures and localize their faults by automated analysis of distributed flows [16, 17, 42]. Their techniques assume that the system being observed has naturally observable user request flows, such as RPC-based systems. For example, an algorithm targeted to HTTP-based distributed systems uses HTTP request logs to recover user-request flows in Web-server farms. Another algorithm is based on the assumption that the target system uses a high-level component framework, such as J2EE, and an underlying framework implementation was modified to record component interactions. By collecting these data, their analysis algorithm based on probabilistic context free grammar (PCFG) automatically learns correct flow models, and detects failures by finding deviations in observed flows from the learned models. Finally, to find the root causes of a detected failure, they correlate the failure with particular components by analyzing recovered flows with several statistical techniques, such as cluster analysis and decision trees. While their framework covers both failure detection and fault localization in request-reply based systems, their applicability to other types of systems is unclear In fact, the authors themselves discussed an example of such limitations, where a single unit of work incurs multiple RPCs [42].

Based on our earlier collaboration [58], Mirgorodskiy presented another trace-based flow-recovering algorithm that imposes little manual burden by using automated binary instrumentation across node boundaries [60]. For all send and receive calls, the code inserted by his instrumentation records the size of the message sent or received, allowing post-mortem analysis to correlate trace records in separate processes. Similar to this work, he uses function call traces, and quantifies the differences between identified function-call flows to find anomalous calls.

Reynolds et al. proposed an approach, called Pip, to assisting application programmers to detect unexpected system behaviors [71]. Similar to our model-based approach, they first infer the expected program behaviors from test runs that include interactions between distributed components, and generate textual representations of the expected behaviors. The auto-generated expectations, which the programmers can extend for more accurate analysis, are checked against the traces of trial runs.

Aguilera et al. [2] presented two statistical algorithms that require no *a priori* knowledge on the target system, while trading off the accuracy of the recovered flows. One of the two algorithms, called the nesting algorithm, is based on the assumption that the flows to be recovered are derived from RPCs, and it stitches together the calls by finding the parent callers of all RPCs. Another algorithm, called the convolution algorithm, is not based on the assumption of RPC-based systems, and find more free-form message flows by statistically computing the correlation of arriving and leaving messages on each node in distributed systems.

The key difference between these past projects and ours is that our approaches do not use distributed flows, but, instead, exploit the observations that allow local behaviors to be viewed as effective data analysis units as discussed in Chapter 4. Namely, our outlier-detection-based approach uses the observation on spatial similarities, and detects anomalies in the per-process traces by comparing them. Our second approach, the model-based approach, is based on the observation on historical similarities, and detects anomalies by comparing their past behaviors. While their flow-recovering algorithms work in a centralized fashion, we designed the localization algorithm in the model-based approach to be mostly decentralized. As shown by Roth et al. [72], decentralized processing is essential to work at scale of today's HPC systems, including the 106,496-node BlueGene/L at LLNL and the 655-node Tsubame supercomputer at the Tokyo Institute of Technology [24].

Another notable difference is that flow-based approaches require the correlation of message send and receive operations to be determined by message counting [60] or tagging [16, 17, 30, 42]. While counting is relatively simple to implement, it cannot be used for UDP connections since messages can be reordered and dropped. Message tagging works for both TCP and UDP connections, but the perturbation due to embedding tags into all messages can be too large to capture nondeterministic, timing-related bugs.

3.1.2 Approaches Based on Spatial Similarities

Other approaches related to the outlier-detection-based method include those by Zheng et al. [98] and Arnold et al. [4]. Zheng et al. presented an anomaly detection algorithm that is also based on the assumption on spatial similarities [98]. Unlike the outlier-detection-based approach, they use standard performance metrics, such as CPU and memory load. We could also improve the usefulness of our approach by using such metrics as well. For instance, bugs such as memory leaks would manifest themselves clearly in the memory-usage metric. However, their method can only reveal process-level anomalies, unlike our function-level analysis that identifies anomalous functions.

Arnold et al. proposed a bug detection method using stack trace sampling [4]. Similar to this work, their primary focus is scalability with increasing numbers of nodes. Thanks to MRNet, a tree-based overlay network [73], they achieve very low-latency collection of samples of call stacks from thousands of processes; we could also make use of such a scalable overlay network to generate a global model. They aggregate the call stacks of all processes into a single call graph by allocating the same node for the same function called in multiple processes, and attributing all edge with cumulative call counts over the distributed processes. By reducing the call stacks over a large-number of processes into a concise single call graph, they aim to reduce the problem exploration space into a small manageable class. A key difference between their approach and ours is that, since they use a sampling-based analysis, it is difficult to capture rare, nondeterministic faults, such as timing-related bugs.

3.1.3 Approaches Based on the Historical Similarity

Comparing performance metrics data across time is another often used technique to find anomalous behavior. Yuan et al. presented a technique of detecting anomalous application performance by comparing observed performance values with their moving-window averages [95]. For example, consider a scientific application that executes the same loop iteratively until some conditions are met. Many such applications should complete a nearly-constant number of iterations per a unit of time. Thus, if the application executed a much different number of iterations than the average, such behaviors could indicate an anomaly affecting the application performance. However, raw performance metrics such as CPU and memory loads can be affected by not only the application itself, but also background management tasks that are typically activated by system tools such as *cron* at some fixed intervals. Since the performance perturbation by periodic background tasks is generally unavoidable, they filter out such legitimate perturbation by using Fourier transform. More specifically, their anomaly detection first converts collected performance data from the time domain to the frequency domain by Fourier transform, and then filters out those frequencies that have high amplitude. Finally, the filtered performance data is then converted back to the time domain.

While their approach can be effective for detecting anomalous application performance even with background noises, it requires to know a performance metric that nearly stays constant over the course of its execution. Such metrics may be easily found for simply structured applications; however, it does not necessarily exist for more general distributed applications, and even if it exists, its discovery inevitably requires internal knowledge on the application.

3.1.4 Approaches Based on Supervised Classification

Another approach related to fault localization is supervised classification, e.g., the work done by Cohen et al. [20], Yuan et al. [96], and Vetter [92]. Cohen et al. proposed an algorithm to predict the occurrence of high-level performance faults, or SLA violations, from raw system performance metrics, such as CPU loads, cache miss ratios, and memory usage [20]. In the offline stage, they train a SLA violation model that predicts the occurrence of particular violation and provides a correlation of the SLA with system performance metrics by using a variant of Bayesian networks, called Tree-Augmented Naive Bayesian networks [28] with normal and failure profiles. At run time, the model can quickly predict SLA violations and their correlated performance metrics.

Yuan et al. also proposed a supervised classification algorithm for quickly determining the causes of problems using system call traces [96]. They train a correlation model using the system call traces of both normal and failed runs. They derive the model by discretizing the traces into N-grams and by applying a Support Vector Machine (SVM) classification technique [12]. At run time, as in Cohen et al., given a failed system call trace, their derived model can locate the specific system-call sequence of a particular known failure, and identify the learned cause of the sequence.

Vetter proposed a supervised technique of finding communication inefficiency in cluster environments [92]. Similar to the detection of SLA violations by Cohen et al. [20], it derives a classifier that notifies of the occurrence of inefficient communications in parallel-program executions. It trains a decision tree by learning sample traces of normal and inefficient parallel programs that contain well known patterns of inefficient communications in message-passing parallel programs, including “late send” and “late receive”.

These supervised approaches are closely related to our model-based approach, since it also learns a model through sample runs. A key difference with our approach and the above supervised approaches, however, is that we do not require the execution data of failed runs for model learning. Supervised classification generally requires a large volume of sample data from all classification categories to learn accurate and robust models, i.e., the execution data for both normal and failed runs in the above cases. Furthermore, they require samples for each specific category of faults for failed runs. Our model learning, conversely, only requires traces of normal runs, which is an important property for localizing rare, nondeterministic faults. Taking sample traces of nondeterministic behaviors ahead of time is difficult in a probabilistic sense. Nonetheless, our approaches are orthogonal; once the trace is collected, we could improve the usefulness of model-based fault localization with their techniques of known-problem diagnosis.

3.1.5 Other Automated Approaches

A tool for measuring performance, called Paradyn, by Miller et al. aims to automate the localization of performance bottlenecks in parallel applications [55]. Paradyn’s Performance Consultant module automatically searches for possible bottlenecks by dynamic binary instrumentation of parallel programs. Performance Consultant first finds the cause of a performance problem by testing pre-defined hypotheses on performance data gather by instrumented parallel programs. One example hypothesis is: “When synchronization blocking time is greater than 20%

of the execution time, the cause of the bottleneck is synchronization." Having identified the cause of the problem, Performance Consultant further automates locating the bottleneck by dynamically narrowing down instrumentation points into finer-grained candidates for bottlenecks.

Their use of hypotheses are similar to our assumption of observations in our approaches: we also initiate localization by finding violations of observations. However, a key difference is that our observations are more abstracted. We only claim that all member processes should behave similar to one another or their behaviors in the past, while their hypotheses include more concrete facts about program behaviors, such as "20%" in the above example. Part of the reason for the difference is that our objective of localization is to find anomalies, while theirs is to discover more specific performance problems. Of course, having specific knowledge on faults would improve localization accuracy; however, deriving such knowledge is not trivial, or may even be impossible for rare, nondeterministic faults. Our contribution, compared to theirs, is that such hard-to-anticipate faults can be localized using simpler observations even without more specific knowledge.

3.2 Other Approaches to Fault Localization

3.2.1 Replay-Based Distributed Debugging

Replay debugging can be a powerful debugging technique by allowing the user to examine not only arbitrary program states as when a symbolic debugger, such as GDB [27], is used but also at arbitrary times by rewinding and forwarding executions [30, 31, 50, 76, 81]. Konuru et al. presented such a debugging technique for distributed Java applications[45]. By modifying a standard Java Virtual Machine (JVM), they record all run-time events that are required to deterministically replay the executions later, including thread scheduling switches, synchronization operations, and establishing and closing network connections. The recorded execution can be replayed later in exactly the same order as the original. Another replay debugger, called liblog, by Geels et al. provides replay debugging for C/C++ native distributed applications by using a user-level library interpositioning technique [30]. The liblog debugger records all the interactions that all processes perform during recording, even incoming network messages for complete deterministic replay, without any modifications to the underlying operating system or application code. They also presented an approach that allows the user to validate global predicates on data and control flows that involve distributed processes, e.g., a predicate that states an overlay ring topology must always actually form a ring [31]. WiDS Checker by Liu et al. also provides replay-based debugging for applications written with their own API for constructing distributed applications called WiDS [50]. It is more powerful than Friday because WiDS allows applications to be simulated and thus WiDS Checker can search for possible predicate violations with brute force simulation runs.

Compared to fault localization based on automated data analysis replay debugging can be seen as an orthogonal technique: Once a suspicious symptom is found with our anomaly detector, further root-cause identification could be greatly eased with the use of such replay debuggers, if they could be applied with affordable overhead. While tools being able to completely replay executions of distributed systems with both deterministic and nondeterministic behaviors can be very useful for debugging complex systems, their application is restricted to lightly loaded applications due to its comprehensive recording of states. Replaying a large number of processes with a single process as in WiDS Checker also limits its application to executions with a small number of processes. While Liu et al. showed that their technique can replay multiple processes efficiently, their evaluations are limited to cases using very small numbers of processes such as four. In addition, as suggested by Geels et al. [31], simply being able to replay a buggy distributed application cannot always help the programmer detect its bugs, due to the daunting complexity and scale of the target application.

Automated predicate checking by Friday and WiDS checker partially solves this problem; however, deriving effective predicates still requires *a priori*, expert knowledge, and even worse, predicates to check the absence of emergent misbehaviors are even harder to derive, if at all possible. Our anomaly detection could come into play here; anomalies found with our techniques could be an effective starting point for more thorough diagnosis with replay debugging.

3.3 Efficient Techniques of Recording and Representing System Behaviors

Raw traces, no matter whether they are textual or binary, are unstructured, voluminous, and difficult-to-comprehend. Thus, any form of trace analysis can be divided to confront two common challenges: efficient trace recording and effective information retrieval from the traces. We review related work on these issues in the rest of this section.

Reiss et al. presented two modeling schemes for function call traces, the first based on context-free grammar (CFG) and the second based on finite state automata (FSA) [69]. Their main objective is to compact voluminous traces by detecting those that are repetitive. To do so, CFG-based modeling repeatedly finds a frequent sequence of traces, and assigning a new non-terminal to it. FSA-based modeling assigns a node to all traces, but, at the same time, merges any pair of nodes whose trailing *k* traces are the same. They demonstrated that both approaches are quite effective in compacting traces in various applications.

Similar compaction schemes could also have been used in our approaches; however, merging traces could sacrifice the resolution of localization. In our approach based on outlier detection, we derive a time profile for each per-process trace, which is a vector of the normalized time spent in each function (see Chapter 5). If two functions are merged into a single non-terminal or state, as in Reiss et al., we need to merge the execution time information of both functions as well, decreasing the resolution to differentiate which function is the true source of the anomaly. The same problem could also appear in our model-based approach.

Verbowski et al. presented a technique of compacting traces online for always-on tracing of persistent-state interactions [91]. Examples of such traces include I/O access to files, changes to operating-system configurations, and process management. Based on their observations on the characteristics of such traces, they achieve highly-efficient trace formatting and its compression algorithm, which allowed traces to be collected from more than 10,000 machines and an 8TB RAID system to retain the traces of a window for one month. The key observation allowing such compaction is similar in our observations on function-call traces. That is, while there can be a huge number of persistent-state interactions, the number of distinctive traces is typically much smaller (This was only 0.2 to 5.4% in their case study on a large Web-server farm.). Unlike this work, they further reduce the storage requirements by applying a standard block-compression algorithm; we expect such an approach would also be effective in our normal-trace collection phase as well, although this needs to be evaluated.

3.4 Scalable Techniques for Performance Analysis

Another related research area, especially with the model-based approach, is scalability for analyzing performance in large-scale HPC systems [3, 10, 32, 36, 72]. Manually finding performance bottlenecks in current large-scale machines is a daunting task; we review past approaches that simplify such analyses through automation.

Roth et al. presented a scalable framework for analyzing performance [72]. They collect performance profiles at run time by constructing a tree network of distributed nodes using MRNet [73, 74]. Their highly decentralized framework of analysis allowed searches by Paradyn [55] to find performance bottlenecks with moderate loads on CPUs and networks even in more than a thousand processes, whereas a centralized analyzer could not handle such a large number of processes. A tree-based overlay, like MRNet, would also be useful in our derivation of global models, even though we have not yet seen bottlenecks in collecting process models.

While Roth et al.'s technique exploits the parallelism in the target computing environment to manage a large volume of performance data, others have reduced the volume by using data mining methods, such as cluster analysis. Nickolayev et al. presented a cluster-analysis-based technique to reduce the volume of execution traces at run time [64]. The traces are clustered into a small number of groups, on a premise similar to our observation of spatial similarities, and only cluster centroids are saved into storage. Ahn et al. presented a technique based on data mining to analyze performance counter values in large-scale computing systems [3]. Extracting useful information from performance counter values taken during executions of parallel programs is not a trivial task because the volume of collected performance data can be overwhelmingly large. They found that clustering data can reduce the exploration space into a manageable size, allowing the analyst to quickly discover performance bottlenecks. Huck et al. further advanced such performance analysis by integrating various data-mining methods into Tuning and Analysis Utilities (TAU), which is a framework for analyzing the performance of parallel applications [36, 79].

Geimer et al. proposed a parallel algorithm for finding communication patterns that have sub-optimal performance [10, 32]. They use the same set of nodes as the target system, and attempt to discover inefficient communication patterns scalably by replaying all communication events on the same node and identifying distributed message correlations. We could have used such a parallel correlation-based technique in our model derivation phase as well. For example, while the current modeling assigns a single model to each network connection, we could have differentiated function-calling patterns by using the call stack of the sender as a key. By doing so, we could have generated a model for a pair made of a unique connection and its sender (see Section 6.1), which would have improved the accuracy of the resulting model. However, this in turn would have required distributed message correlations to be found by methods such as message counting and tagging. Since the overhead incurred by message correlation can be too large to detect timing-related bugs, the effectiveness of correlation-based analysis applied to our problem domain remains unclear and is a subject for future work.

3.5 Approaches to Fault Localization of Sequential Programs

We review several related research efforts on automated fault localization in sequential programs. While our approaches, especially that based on the detection of outliers, share some key techniques for localizing faults, the key difference is that, while their approaches only need to consider intra-process exploration space as possible locations of faults, we have to search through different processes as well, which can be tens of thousands in modern large-scale environments. The rest of this section elaborates on the similarities and differences between their efforts and ours.

Dickinson et al. presented a technique based on cluster analysis to help the system developer examine the reports from beta testing [22]. Similar to this work, they instrument the target program so that it generates function call profiles. The program testers run the instrumented version to collect per-process call profiles. Similar to Arnold et al. [4] and Ahn et al. [3], their objective is to automatically reduce the exploration space by aggregating a large number of profiles into a concise summary. They use cluster analysis so that the developer can only examine a small number of profiles in each cluster, instead of all the testing reports. However, theirs still requires per-process profiles for all clusters to be manually examined, while ours further automates the localization of faults into function-level granularity.

Renieris et al. proposed a nearest-neighbor based technique to aid the analyst in localizing faults using known-normal and known-abnormal execution profiles [70]. They localize faults by detecting differences between the behaviors of a failed run and those of its nearest-neighbor normal run. While we do not assume that we know whether each process executed normally or abnormally, they require such knowledge on every execution profile. Examining all processes to label whether they are normal or abnormal would be too costly in large-scale computing environments without automated techniques like ours.

Cooperative Bug Isolation by Liblit et al. [47, 48] determines the predicates in program executions that correlate with the failure of the program. For example, it uses a predicate that the return value of a certain function will always be greater than 0. They instrument the target program to evaluate a wide range of fine-grained predicates, such as where a branch is taken or not taken, and record their results at run time. Unlike our tracing approaches, theirs is based on sampling to record fine-grained predicates with little performance overhead at run time. Furthermore, they distribute sampling to the actual users of the program to collect sufficient sample data to derive reliable statistical results while keeping the overhead low.

Compared to our approaches, we could improve the accuracy and resolution by considering finer-grained execution traces than function-level traces. For example, the return values of certain functions in C-based applications could become effective indicators of faults, since such applications typically return non-zero or negative values on failure. However, sampling does not ensure that the execution data will always be captured when failures occur, which is important in localizing nondeterministic faults. Although the chance of selecting the specific samples can be increased with the distributed sampling approach, it is still unclear how effective the cooperative approach would be for general distributed computing systems. Another difference is that our analysis does not require processes to be identified as normal or abnormal, which is especially important in large-scale environments.

3.6 Similar Approaches to Different Problem Domains

This section reviews several related research projects that employ various data analysis techniques that are similar to ours but for different problem domains.

3.6.1 Security Issues

One of the most important security problems is to detect malicious executables, often sent to computer users as email attachments. Traditional techniques use a database of known-normal and known-malicious programs to identify the signatures of these malicious programs. They generate binary scanners with the signatures that classify programs as malicious or benign. While such techniques can very effectively classify known executables, generating effective scanners for unseen new malicious executables is still difficult, and requires expert knowledge and significant manual effort.

Schultz et al. presented a data-analysis-based method of automatically generating malicious executable classifiers [78]. They use supervised classification techniques with sample programs that were malicious and benign, including an inductive rule learner and Naive Bayes learners. To train classifiers, they extract executable features for classification by finding several static properties as DLL calls, ASCII strings, and bytecode sequences. Their experimental evaluation using publicly-available Windows executables demonstrated significant improvements over traditional techniques.

Their classification technique is similar to our model-based technique since both of them use previously available information to train a model for detecting problems in the future. The major difference between their method and ours is which program properties to use to derive the model. As previously described, they only use statically available information, while we monitor run-time behaviors by tracing function calls. While using only static information is important for being able to detect malicious executables without actually executing them, we do not have such restrictions since our purpose is to automate the localization of faults post-mortem. Monitoring run-time behaviors allows us to capture actual fault behaviors that may be impossible to detect using only static information, especially those that are nondeterministic.

Detecting malicious attacks, i.e., intrusion detection, is another important problem in the security domain. Wagner et al. presented a technique of generating an intrusion detection system (IDS) [93]. Their technique is aimed at detecting the execution of corrupted code through the exploitation of buffer overruns and other security holes in normal applications. To do so, they generate a model that describes the normal execution behavior of the application being observed by statically analyzing the executable code. Specifically, they use several forms of automata, including nondeterministic finite automata (i.e., regular languages) and pushdown automata (i.e., context-free languages), and demonstrated that for moderately-sized applications their models effectively detected all the attacks tested. The major difference between their research and ours is the objectives: detecting executions of unknown code versus localizing faults such as logic and performance bugs. For example, they only consider application's system calls to detect malicious executions, since they assume that executions of unknown code would depend on using certain system calls. While we agree that their assumption is valid to attain their objective, it is too simplistic for localizing program logic bugs and performance problems, since they can manifest themselves without perturbing system calls. Another difference is the data for learning models: source code versus call traces. Using only source code is advantageous in the sense that it does not require actual executions of applications, and detecting unknown executions of system calls would be feasible even only with source code, as demonstrated by their evaluation results. However, it is difficult to infer, for example, how long a function execution would take by only using such static information, and thus we expect that using only source code would be of limited effectiveness in localizing faults.

3.7 Summary

The contributions of this dissertation have been build on previous work in various fields, especially fault localization, execution logging, system behavior learning, and scalable techniques of performance analysis. First, we reviewed the most closely related research area, i.e., fault localization, particularly focusing on the scalability and automation. Most of the previous approaches on this subject have attempted to recover distributed flows from distributed traces in deterministically [8, 16, 20, 42, 60, 71] or statistically [2]. As we have discussed, however, they have not addressed the scalability issue that arises when doing so in large-scale computing systems consisting of thousands of machines. Correlating local events to build distributed flows in such large-scale environments is no trivial task. Although the technique of parallel analysis of MPI traces proposed by Geimer et al. [32] could help to some extent, it is still unclear whether such an approach would be possible with an affordable overhead in more complex systems software.

Other automated approaches that are similar to this work include those based on similarities and supervised classification, although our approaches requires less manual effort. While not as automated as them, replaying distributed applications is another focus of much research in fault localization. Techniques such as Friday [31] and WiDS Checker [50] are indeed powerful debugging tools for application programmers; however, they have the same scalability issue as the flow-based approaches.

We have also reviewed previous work on recording and representing system behaviors, scalable performance analysis, and detection of intrusions and malicious executables. In particular, the studies on increasing the scalability of various online performance analysis [3, 10, 32, 36, 72] had inspired and affected our particular focus on the scalability of fault localization.

Chapter 4

High-Level Overview of Our Proposed Approaches

To tackle the problems stated in Chapter 1, we propose trace-based approaches to fault localization that simplify the identification of root-causes in large-scale computing environments. This chapter first discusses the design goals that we set to solve the problems in Section 4.1. Section 4.2 elaborates on several key observations and assumptions on the target environments that we exploit to design our localization techniques. Although these prerequisites limit the application of our approaches to environments where they can actually hold true, we argue that they are sufficiently common to assume without significantly reducing the real usefulness of our proposals. Finally, in Section 4.3, we present an overview of our approaches that achieve automated fault localization in large-scale environments by taking advantage of the observations and assumptions.

4.1 Design Goals

Fault localization in large-scale environments presents several difficult challenges, including scalability barriers and diversity in commoditized component systems (see Section 1.1). To solve these challenges, we have set three design goals, namely, 1) automated analysis, 2) fast, scalable analysis, and 3) localization of unknown faults. The rest of this section discusses these goals in more detail.

4.1.1 Automated Analysis

The first goal is to automate fault localization to the extent possible. In large-scale computing environments, one of the most dominant approaches is still to localize faults by ad-hoc logging of various system states that are explicitly inserted by the programmer. Effective localization with such manual logging requires extensive knowledge about the internal architecture of the system being observed. Even if logs are a valuable source if properly collected, identifying effective system states to record and extracting useful information from the recorded logs remain to be a tedious manual task. Worse, since the identification of important states is neither trivial nor easy, manual logging tends to result in either too verbose or too little content, both of which do not provide effective means for fault localization per se. The ever increasing scale of modern HPC systems can further increase the difficulty of extracting useful information, since the volume of collecting logs from the system increases as the system scale becomes larger.

Unlike these existing approaches, we set a goal where fault localization has to be automated to the utmost extent. The manual collection of system logs as is done in many practical cases is so ad-hoc and tedious that it is not a viable option for our purposes.

4.1.2 Fast, Scalable Analysis

The second goal is fast, scalable analysis in current and future large-scale environments, which can be as large as tens of thousands of machines. As discussed in Section 3.1.1, the dominant approaches recover distributed flows by analyzing system execution traces deterministically or statistically. A critical limitation in these approaches is their applications in large-scale systems. For example, a simple implementation of a message-tagging-based algorithm for recovering flows would first collect each per-machine (or per-process) trace into a central repository to match sender and receiver tags. However, collecting and matching traces from tens of thousands of processes in a central repository is not likely to scale well. Specifically, reading through the traces and matching tags would take linear time with the increasing number of processes. While this might not be a problem in relatively small-scale environments where the number of machines ranges from tens to hundreds, it could impose prohibitively large costs in present and future large-scale environments.

Unlike these past approaches, we set our goal to achieve fast, scalable analysis with no scalability barriers even with tens of thousands of machines. Distributed flows could be effective sources for localizing faults, especially those that manifested themselves over distributed components; however, such analysis is not likely to scale well in the large scale environments that we envision.

4.1.3 Localization of Unknown Faults

The third goal is to localize faults that have been little or never observed before. Such faults are more difficult to diagnose than known faults, since problem analysts may not have much past experiences that could be exploited to localize them. Large-scale systems are more prone to intermittent, timing-related faults that only occur very infrequently; thus, being able to localize such faults is an important criterion for localizing faults in large-scale systems.

As discussed in Chapter 3, some previous research projects have demonstrated the effectiveness of supervised learning of system behaviors in fault localization [20, 92, 96]. For example, decision trees, which can be trained using both success and failure execution logs, can provide useful indications about the faulty system properties [92]. Other supervised techniques of learning, such as support vector machines and naive Bayes classification have also been demonstrated to be effective for finding the causes of several types of failures [20, 96]. However, the limitation of these supervised approaches is that they require to obtain a large volume of training data from both succeeded and failed executions. Observing many instances of succeeded executions may be implemented for systems that are already deployed to real production environments, if not trivial. However, production systems should not frequently yield failures, making it time-consuming to obtain a sufficient volume of failure training data. In addition, since large, real production systems are inherently complex, the number of failure patters are also large, making the data collection for effective training further difficult.

Instead of attempting to obtain training data of both succeeded and failed executions, we design analysis algorithms so that they do not rely on the availability of failure data. This design decision allows us to avoid anticipating possible faults beforehand and to develop localization algorithms that can be used for unknown faults.

4.2 Observations and Assumptions

To accomplish fault localization while achieving these three design goals, we exploit the following two observations on large-scale computing systems, i.e., spatial and historical similarities.

4.2.1 Spatial Similarities

The observation of spatial similarities is that many classes of distributed software for clusters and Grids, such as job schedulers and parallel file systems, consist of a small number of collections of equivalent programs running on each member node. For example, typical clusters for HPC consist of a single *master node* and a collection of *compute nodes*, and use a *master-worker* parallelization scheme to coordinate the compute nodes. A master-worker system designates a single process running on the master node as a *master process* and lets it manage *worker* *processes* running on the compute nodes. For example, a batch job scheduler for clusters, called Torque [19], consists of a master process called `pbs_server` and per-node worker processes called `pbs_mom`. The `pbs_server` process handles incoming user requests to submit new jobs, query job statuses, and withdraw submitted jobs. These requests are then forwarded to the appropriate `pbs_mom` processes, which manage jobs under their own node, and return the answers to the `pbs_server` process. Another example of the master-worker architecture is a parallel-job manager, called the Multi-Purpose Daemon (MPD), which is shipped with a popular Message-Passing Interface (MPI) implementation (MPICH) [35]. MPD consists of a collection of equivalent daemon programs running on each node and connected to one another through a ring-topology network. One of the daemons in the ring operates as the master process, and handles user requests and spawns specified parallel jobs in the managed nodes by forwarding the requests to the worker process in the ring network.

4.2.2 Historical Similarities

The observation of historical similarities is that each process in target systems is likely to behave similarly to the way it did in the past. This observation is based on a common event-driven software architecture for network systems. An *event* is a software abstraction that notifies changes in some internal or external states. In typical event-driven distributed systems, each member process would run infinitely executing *event handlers* for new events. For example, each worker in a master-worker system would execute event handlers to serve incoming requests from the master process. We expect that these systems would exhibit similar behaviors to those they had in the past, since their behaviors could be characterized by the continuous execution of the event handlers.

Note that specific mechanisms for event handling can vary from system to system since modern operating systems support several programming models for such operations. One of the most commonly used is the `select` system call [83] to multiplex the processing of multiple events into a single *event loop*. The process would block on a call to `select` to wait for status changes in resources associated with the given file descriptors, and for each file descriptor whose status was changed, it would execute the event handler itself or spawn a new process or thread to execute the handler on behalf of itself. Another common mechanism is to use polling to detect new events, e.g., using the `poll` system call [83]. While specific implementations would differ as described above, our observation of historical similarities would still hold true.

4.3 Overview of Proposed Approach

We propose two techniques of localizing faults in distributed systems, i.e., outlier-detection-based and model-based techniques. The underlying approach of both techniques is to automate the fault-localization process through the detection of violations against the above observations in system behaviors. Specifically, to localize a given fault, we start with either of the above observations of similarities, monitor the system by collecting execution data where we expect that the observation should hold true if the system operates normally (*data collection*), and then find violations, or *anomalies*, in the behaviors against the observation (*anomaly detection*). If we find any violations, we consider them as *symptoms* of the fault, and report them to the problem analyst, potentially ordered by the anomalousness of each symptom. The rest of this section first describes the data collection mechanism in more detail, and presents a brief overview of the anomaly detection.

Note that we do not advocate that the process for identifying the root cause could be completely automated; rather, our aim is to significantly simplify the current manual process through our automatic techniques of detecting anomalies. Part of the objectives of this dissertation is, therefore, to explore effective techniques of anomaly detection for localizing faults, and study their effectiveness when applied to real-world systems.

4.3.1 Data Collection: Always-on Function Call Tracing

We decide the methodology for data collection based on four criteria. First, it has to be implemented without much manual effort to make the localization process as automatic as possible. Second, collected data have to exhibit similarities in observations of normal operational states, since this is one of the primary assumptions that our anomaly detection techniques are based on. Third, they have to reveal information that permits problem analysts to effectively identify the root cause and take corrective steps, even for unknown faults. Fourth, the performance overhead due to data collection has to be kept minimal, since too much overhead could preclude our technique from being applied to performance-sensitive systems.

Considering these criteria, we record function calls by always-on tracing as depicted in Figure 4.1. Each line represents an entry for a call to or return from a function whose address is specified in the second column. The other columns represent the process and thread IDs of the thread that executed the call or return at the time specified in the last column. Note that while the figure illustrates the traces in textual form, our prototype actually generate them in a more-concise binary format.

The primary reasons to perform always-on function tracing are four-fold. First, we can expect that function-call traces would exhibit spatial and historical simi-

```
ENTER 0x819967c pid 5095 tid 4 timestamp 12131002746163258
LEAVE 0x819967c pid 5095 tid 4 timestamp 12131002746163936
ENTER 0x819967c pid 5095 tid 4 timestamp 12131002746164571
LEAVE 0x819967c pid 5095 tid 4 timestamp 12131002746165197
ENTER 0x819967c pid 5095 tid 4 timestamp 12131002746165828
LEAVE 0x819967c pid 5095 tid 4 timestamp 12131002746166395
LEAVE 0x80de590 pid 5095 tid 4 timestamp 12131002746166938
ENTER 0x819967c pid 5095 tid 4 timestamp 12131002746167573
LEAVE 0x819967c pid 5095 tid 4 timestamp 12131002746179202
ENTER 0x80de750 pid 5095 tid 4 timestamp 12131002746180027
ENTER 0x811b070 pid 5095 tid 4 timestamp 12131002746180691
ENTER 0x8138710 pid 5095 tid 4 timestamp 12131002746181359
LEAVE 0x8138710 pid 5095 tid 4 timestamp 12131002746185934
...
```

Figure 4.1: Textual representation of function call traces. Each line represents an event entering or leaving the function specified by the second column. The remaining six columns show the process and thread IDs of the thread that generated the trace, and the timestamp when the trace was recorded.

larities, since they reflect the control flow of a monitored process. Simple master-worker-based distributed systems would consist of two classes of processes, i.e., the master and worker processes. While the master and the workers would execute different functions, all the workers would execute mostly the same set of functions, exhibiting spatial similarities in function traces. Besides, event-driven software architectures in many distributed systems would have certain patterns of function calls that are derived from the event-processing functions. Such patterns should appear repeatedly in the course of executions of the target system, exhibiting historical similarities in function traces.

Second, function-level data collection is often a good balance between data granularity and collection overhead. Being able to locate anomalous functions was indeed effective in identifying the root causes of faults in the experimental studies presented in Chapters 5 and 6. Function traces can often be obtained with an acceptable performance overhead. Again, one experimental study that we conducted with the SCore cluster-management system indicated that the end-to-end slowdown of applications in the NAS Parallel Benchmark (NPB) suite [94] was less than 1% while SCore daemons were being traced.

Third, function traces are relatively easy to obtain compared to finer-grained information such as branch-level or basic-block-level tracing. While collecting such finer-grained information would be possible by instrumenting programs or executing them under modified interpreters, modern programming languages, especially VM-based ones such as Java and Python, often provide built-in function-tracing capabilities, thus allowing applications to be traced semi-automatically with little changes to application code. Furthermore, even gcc, a popular C compiler [26], provides a function-call hook mechanism as a compile-time option. Alternatively, we can use binary instrumentors, such as Dyninst [11], Pin [51], and Valgrind [63], to generate function traces of binary programs.

Fourth, we believe that always-on tracing is the only viable option to accomplish localization of unknown faults. Always-on recording of execution behaviors eliminates the necessity of reproducing faults to localize, which will significantly simplify the difficulty of localizing unknown faults. However, the performance overhead due to always-on tracing might preclude the use of our techniques in certain systems, depending on particular proving points and techniques. For example, function call tracing might cause a prohibitively high overhead in CPU-intensive scientific applications. Yet, even in scientific applications, function-call tracing of I/O-intensive applications that spend most of their execution time blocking on I/O operations would not cause too high an overhead. Moreover, much of the distributed middleware for clusters and Grids, e.g., batch job schedulers and inter-site authentication services, are lightly loaded in terms of CPU usage. In fact, as previously noted, the decrease of application performance while the SCore daemons were traced was less than 1%. This dissertation explores how applicable the trace-based always-on monitoring we propose is within the context of large-scale computing systems.

In addition to always-on function-call tracing, we considered several alternative methods of collecting execution data; however, none of them does not meet the design goals.

Collecting Data at Different Granularities

We considered finer-grained data collection, such as tracing basic blocks, branches, and even instruction counters; however, they would have caused a significantly larger performance overhead than function tracing, restricting application only to less-lightly loaded systems. Such large overheads could be reduced by sampling data collection as in Liblit et al. [47, 48]. However, sampling in turn makes it difficult to capture system behaviors induced by rare faults.

We also considered tracing coarse-grained information, including nodes, processes, and software components, as has previously been done [8, 17, 42]. While such coarser-grained tracing would reduce the performance overhead, it would in turn reduce the effectiveness of locating root causes.

Using Standard Performance Metrics

Other easily obtainable data include standard performance metrics, such as the CPU load, memory usage, and network I/O, since operating systems usually provide standard interfaces to collect such metrics (e.g., the `uptime` command in Unix variants). However, the main drawback compared to function-call traces is that, even if a performance anomaly were found in such performance metrics, it would be difficult to correlate this to specific locations of programs, such as functions. Thus, correcting detected anomalies would require additional localization steps. Nonetheless, we might be able to find a wider variety of performance bugs by extending function-call tracing with these performance metrics. Specific techniques and evaluations of this extension are beyond the scope of this dissertation and remain a subject for future work.

Post-Mortem Fault Reproduction with Less Run-Time Monitoring

Any system monitoring inevitably incurs some degree of overhead in end-to-end user-perceived performance, and should be avoided if the best achievable efficiency is the most critical concern. Furthermore, the longer the system is monitored, the more data the analyst needs to examine. In particular, performance-sensitive production systems might favor minimal run-time monitoring, and, if a failure occurs, find its cause by reproducing the exactly same behavior with more-extensive data collection.

However, such post-mortem fault reproduction is not a trivial task even for system experts due to the following two reasons. First, since any additional probing inevitably disturbs the original system execution behaviors, reproducing the exactly same fault as the original would require carefully-selected probing points, perhaps in a trial and error manner. In fact, nondeterministic, timing-related bugs might not appear at all while the system were monitored. Second, even if it could reproduce a particular fault, it might take very long time for the fault to appear again. A fault that occurs in an extremely small probability might take days or even longer to manifest itself in the system. The long downtime due to its reproduction would not be an acceptable option in most of real production systems.

Replay debugging tackles this difficulty of reproducing faults by recording all nondeterministic events, such as network message ordering and random number generation, and replaying them deterministically. However, reproducing *latent* faults that only occur long after the system is started would still take a considerable amount of time. Combining state recording with process checkpointing as in Geels et al. [31] could reduce this reproduction time, but the large overhead due to checkpointing would limit the application to very lightly loaded applications.

4.3.2 Fault Localization through Anomaly Detection

We present two fault localization techniques that detect anomalies in function traces by assuming either of the above observations of similarities. The first technique, *outlier-detection-based fault localization* assumes spatial similarities, and automatically detects anomalies by finding outlying processes in the per-process traces as follows. When a failure occurs in the system being observed, it collects all the per-process function traces, and analyzes how long each process spent executing a particular call stack by simulating the calls and returns of the process. Based on the time spent in each call stack, it then maps each trace to an N-dimensional Euclid space, where N is the number of distinctive call stacks. It finally finds outlying processes by detecting traces that are isolated from the rest of the traces [9, 68]. Chapter 5 presents more details on this technique.

The second technique, *model-based fault localization*, assumes historical similarities, and detects anomalies by finding differences between each process's behaviors in normal runs and failed runs. To find differences, it derives an execution model that reflects the normal execution behaviors of the target system. To derive a model, it uses known-normal traces collected while the system operates as intended. When a failure occurs, it detects anomalies by finding different function-calling behaviors between the learned model and the failure traces. Note that unlike the outlier-detection based technique, it requires an *a priori* learning phase before it is able to localize faults. However, once the models are learned, it operates in a completely decentralized way without interacting remote components. Thus, it can achieve more rapid and scalable localization than the outlier-detection-based approach, which requires process-by-process comparisons. Chapter 6 presents more details on this technique.

Table 4.1 compares the two techniques with respect to our design goals, i.e., 1) automated analysis, 2) fast, scalable analysis, and 3) localization of unknown faults (see Section 4.1). The technique based on outlier detection is superior to the model-based one to attain the goal of automated analysis, since the latter requires a model to be generated *a priori* with known-normal execution traces, while the former operates without requiring such preparations. However, we expect that the model-based technique will outperform the outlier-detection based one to attain the speed and scalability goals because once a model is learned, anomaly detection is a completely localized per-process operation. However, the technique based on outlier detection needs to compare per-process traces, which can be as many as tens of thousands in large-scale environments. Finally, our case studies presented in the subsequent chapters demonstrated that 1) function tracing can capture unknown, non-deterministic faults in large-scale environments; and 2) detecting anomalies in such data can substantially simplify fault localization of such faults.

Table 4.1: Comparison of Outlier-Detection-Based and Model-Based Techniques.

Design Goals	Outlier Detection	Modeling
Automated analysis	*Good*	*Moderate*
Fast, scalable analysis	*Moderate*	*Good*
Localization of Unknown Faults	*Good*	*Good*

Chapter 5

Outlier-Detection-Based Fault Localization

The outlier-detection-based technique simplifies fault localization through a three-step automated process as outlined in Figure 5. Each step performs the following:

Step 1: Data collection The first step involves collecting per-process function-call traces from the system being observed, as described in Section 4.3.1.

Step 2: Process-level localization The second step finds anomalous processes in the collection of distributed processes by analyzing per-process function traces. To do so, it exploits the observation of spatial similarities: We consider processes whose behavior is significantly different from the rest anomalous. As we discussed in Section 4.2.1, many classes of distributed systems for clusters and Grids should exhibit similarities, often because of the master-worker architecture employed to organize distributed processes. Based on the similarity assumption, we discover anomalous processes by finding the earliest last timestamps or applying a distance-based outlier detection method. The former aims to localize fail-stop faults, while the latter aims to localize non-fail-stop faults. In distance-based outlier detection, we map each per-process trace into an Euclid space, and define a *suspect score* based on the distance to the k^{th} nearest neighbor. We consider traces that have the highest suspect scores anomalous.

Step 3: Function-level localization The final step finds the functions, or call stacks, that best explain the anomalousness in the processes detected at the second step. For fail-stop faults, we report to the problem analyst the last trace entry of the fail-stop process. For non-fail-stop faults, we calculate the contribution made to the suspect score of each function, and report to the problem

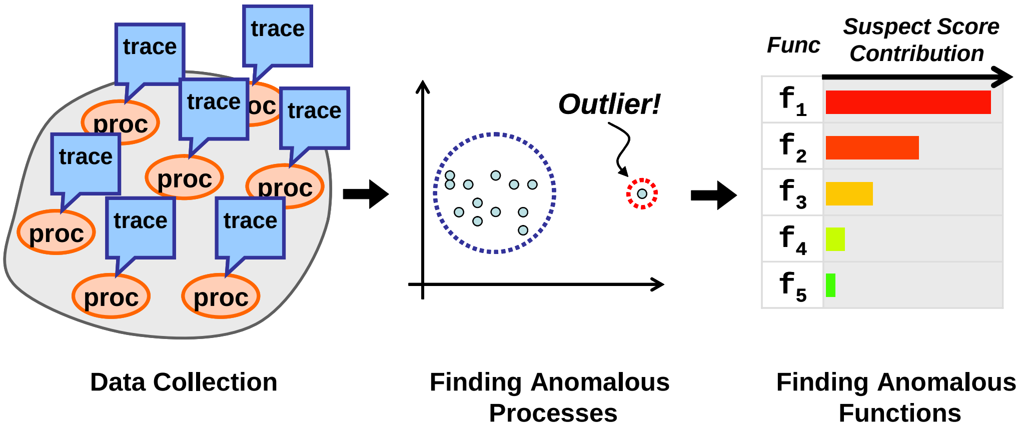

trace
proc
Outlier!
Func
Suspect Score Contribution
f_1
f_2
f_3
f_4
f_5
Data Collection
Finding Anomalous Processes
Finding Anomalous Functions

Figure 5.1: Overview of the outlier-detection-based fault localization. The first step, annotated with "*Data Collection*", involves collecting function-call traces from the target application. The next step, annotated "*Finding Anomalous Processes*", localizes the fault to the anomalous processes. The final step, annotated with "*Finding Anomalous Functions*", ranks the functions according to their contribution to the suspect score.

analyst the functions ordered by the contributions to emphasize more suspicious ones.

We evaluate how effective the proposed technique is by applying it to localize real faults that occurred in a 129-node production cluster running the SCore cluster middleware [39]. Our fault-localization technique successfully identified several anomalous function-call behaviors observed while the system was operating incorrectly, which led us to find unknown bugs. These results suggest that our proposed technique as well as observing similarities are effective in real-world computing systems.

The rest of this chapter first describes the details of the techniques for process-level and function-level localization in Sections 5.1 and 5.2. We also discuss the scalability of process-level localization with respect to the number of processes in Section 5.1. Finally, we present our experimental results in Section 5.3, followed by a brief summary in Section 5.4.

5.1 Process-Level Localization

To locate anomalous processes, we use two algorithm: 1) the identification of the process that stopped generating traces first (Section 5.1.1), and 2) the identification of processes whose traces are the least similar to those of the other processes (Section 5.1.2). The first technique locates processes with fail-stop problems; the second focuses on non-fail-stop problems. As noted in Section 2.1, both techniques involve the assumption that a failure itself has been detected by an external *failure-detection* mechanism (see Section 2.1 for the definition of "failure detection"), and attempt to find the processes that caused the given failure. To do so, we first apply the localization for fail-stop problems to the traces collected when the failure occurred, and determines whether the failure is actually fail-stop. If not, we use the second localization algorithm to find behavioral anomalies.

5.1.1 Fail-Stop Problems

The first technique finds outlying processes by comparing the last timestamp of all per-process traces. If one process or a small number stopped generating traces significantly earlier than the others, the function that generated the last trace record may be correlated with the root cause of the problem. For example, application crashes or infinite blocking in system calls would manifest themselves as such fail-stop symptoms. Such outlying processes are likely to be of interest to the problem analyst.

To find a fail-stop process, we first adjust the last timestamps of the processes so that they can be comparable with one another. Since a timestamp is recorded by reading a cycle counter in the CPU of each local node, we convert it to the local clock time by using the `gettimeofday` system call. Of course, local clocks can differ in distributed nodes, and thus simply comparing them could produce wrong results. We assume that they are well synchronized through such time-synchronization protocols as NTP [56]. Minar reported that only 10% of 647,401 nodes on the Internet had an offset more than 20ms, and only 1% have offset greater than 1s [57] . We expect that in a cluster environment the possibility of offset being greater than 1s can be negligible if NTP is properly configured. We conservatively assume that the resolution of local clocks is as coarse as one second.

Next, we compute the mean and standard deviation of the adjusted timestamps, and find the earliest timestamp among them. If the earliest timestamp is substantially different from the mean with a small standard deviation, we decide that the failure is caused by a fail-stop fault. For example, if the earliest timestamp is several minute earlier than the mean with a standard deviation of less than a few seconds, we determine that the process with the earliest timestamp is an outlier.

Otherwise, we assume that the problem is caused by a non-fail-stop fault, such as performance degradations, livelocks, deadlocks, and infinite loops. We localize them with the second technique, which analyzes the whole traces of processes to find outliers.

Scalability with Number of Distributed Processes Computing the average and standard derivation of the last timestamps of all processes can be implemented by gathering all the timestamp data to a single node and simple arithmetic operations with the gathered data. With a tree topology, the gather operation can be done in $O(n \log n)$ steps, where n is the number of processors. Computing the difference between each timestamp and the average takes $O(n)$ floating-point subtractions. Even if n were as large as hundreds of thousands, it would only take a millisecond or less with a modern CPU. Thus, we expect that the number of processes is unlikely to cause scalability problems in this algorithm.

5.1.2 Non-Fail-Stop Problems

To find outlying processes that exhibit non-fail-stop behavior, we look at the difference in the execution time spent in each function. Specifically, we define a *pair-wise distance metric* that estimates dissimilarity between two traces according to the function-execution time. Then, we compute a *suspect score* for each trace by employing an algorithm for distance-based outlier detection [9, 68]. The suspect score of a trace quantifies the difference between the trace and a collection of traces that we consider normal. Since we assume the spatial similarity about process behaviors, we consider that the higher the score is, the more likely the process is to have caused the fault. Thus, the goal in this step is to compute the suspect score for each trace and to construct the trace ranking by using the score.

While this approach attempts to narrow the root cause of a failure down to a small number of outlier processes, false positives are possible if legitimate processes exhibit outlying behaviors. For example, in a master-worker style distributed application, the master process would execute different functions from the workers, and thus would have a large distance to any of them. The above approach to outlier detection, while it does not require any *a priori* knowledge on the target system (i.e., an *unsupervised* algorithm), would consider the master process an outlier.

Our extension of the unsupervised approach attempts to eliminate such false positives by exploiting previous traces, if available. When constructing a trace ranking, our *one-class* [87] algorithm considers an outlier trace normal if it has a similar past trace. For example, if previous traces include a similar trace to that of the master process, the one-class algorithm gives the master process a low suspect score so that it is not considered an outlier. The rest of this section describes the pair-wise distance metric and our uniform analysis framework that handles both the unsupervised and one-class ranking.

5.1.2.1 Pair-wise Distance Metric

The process behavior that we exploit in this step is how each process spent its execution time in functions. We assume the spatial similarity with respect to the function execution time behavior. Thus, we define the distance of two processes so that it reflects the dissimilarity with respect to the behavior. Specifically, we first summarize the raw trace of a process into a function profile that represents the function execution times. The profile for process h is a vector, $\boldsymbol{p}(h)$, of length F, where F is the total number of functions in the application.

$$\boldsymbol{p}(h) = \left(\frac{t(h, f_1)}{T(h)}, \ldots, \frac{t(h, f_F)}{T(h)} \right) \tag{5.1}$$

Here, $t(h, f_i)$ represents the time spent in function f_i by process h, and $T(h)$ the total execution time of the process, i.e., $T(h) = \sum_{i=1}^{F} t(h, f_i)$. The i^{th} component of the vector represents the fraction of the time spent in function f_i by process h.

Unlike the coverage profiles used in the model-based approach presented in Chapter 6, the above function profile uses time profiles. While time profiles may be more suitable for localizing performance faults, they can be more prone to false negatives if the function has high variability in its execution time. For example, assume that a function is always called and takes a time of nearly zero seconds to several minutes. In time profiles, even if the function is not called (thus zero execution time), it can be marked as normal, since the ordinary execution time ranges points near to zero. In contrast, coverage profiles only retain the fact that the function is always called; thus, the absence of calls is immediately determined as unusual. In our analysis framework, the specific definition of the distance metric is replaceable; both the time and coverage metrics can be used in the same manner as described below.

Based on the function profiles, we can define the distance between two processes by the Manhattan distance of the profile vectors. A profile vector, $\boldsymbol{p}$, is a point in F-dimensional Euclid space, where F is the number of functions in the application. Let $\boldsymbol{\delta}(g, h)$ be the vector of the component-wise difference, $\delta_i(g, h)$, of the two profiles, $\boldsymbol{p}(g)$ and $\boldsymbol{p}(h)$. The distance between the two profiles, $d(g, h)$, is defined as:

$$d(g, h) = |\boldsymbol{\delta}(g, h)| = \sum_{i=1}^{F} |\delta_i(g, h)| \tag{5.2}$$

That is, the distance between two vectors is the sum of the component-wise difference between the two vectors.

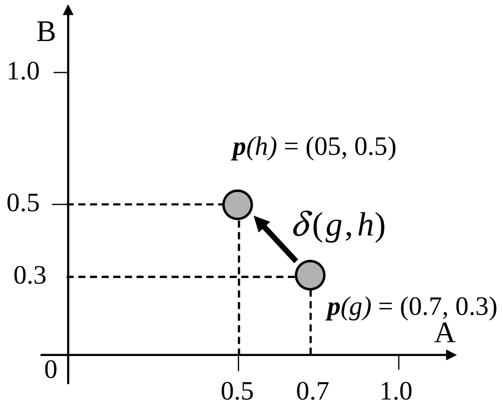


Figure 5.2: Simple example of function profiles. The points $\boldsymbol{p}(g)$ and $\boldsymbol{p}(h)$ represent the profile vectors for the process g and h. The arrow from $\boldsymbol{p}(g)$ to $\boldsymbol{p}(h)$ represents the difference vector between the two points.

Figure 5.2 illustrates a simple case, where the application only has two functions, A and B. The two processes of the application depicted in the figure are the processes g and h, whose profile vectors correspond to $(0.7, 0.3)$ and $(0.5, 0.5)$. The profile vector $(0.7, 0.3)$ means that the process spent 70% of the time in function A and 30% in function B. The distance between g and h is thus calculated as:

$$\begin{aligned} d(g,h) &= |(0.5 - 0.7, 0.5 - 0.3)| \\ &= |-0.2| + |0.2| \\ &= 0.4 \end{aligned} \tag{5.3}$$

We will next describe *call-stack profiles*, an extension of function profiles with context-sensitive analysis, and compare the Manhattan distance and other distance metrics.

Call-Stack Profiles Instead of function profiles, we can use call-stack profiles as well. We can differentiate each function call based on its call stack, since we can simulate the call stack of the process from the function trace. For example, assume that an execution trace contains two stacks 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 stacks 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. By differentiating the functions by their call stacks, the derived context-sensitive profiles can be more precise summaries of the execution behaviors of the application. Since the algorithms presented below are independent of the type of profile used, we will refer to the components of profile vectors as functions for simplicity of presentation. Section 5.3 presents experimental results for the stack-based method.

Manhattan Distance vs. Other Distance Metrics As previously described, we use the Manhattan distance as the distance metric for function profiles, rather than other metrics, such as the Euclidean distance (i.e., the square root of the sum of the squares of the component-wise differences). Both the Manhattan and Euclidean distances are special cases of the Minkowski distance [40]:

$$d_m(g,h) = (\sum_{i=1}^{F} |\delta_i|^m)^{1/m} \tag{5.4}$$

However, we use the Manhattan distance because in the function profile space there should be no shortcuts for calculating the distance between two points in the function profile space. To illustrate this, Figure 5.3 depicts an example scenario, where profile g spent 100% of its execution time in function A, profile h spent 50% in each of A and B, and profile k spent 100% in function C. With the Manhattan distance, the distance between g and h is calculated as $d(g,k) = 2$, and the distance between h and k is also $d(h,k) = 2$. The fact that the two distances are the same is intuitive; after all, both processes g and h share nothing with process k with regards to their execution-time distribution. In contrast, the two distances with the Euclidean metric are not the same: $d(g,k)$ is $\sqrt{2}$, while $d(h,k)$ is $\sqrt{1.5}$. The other distance metrics derived from the Minkowski metric have the same disadvantage compared to the Manhattan distance.

5.1.2.2 Unsupervised Trace Ranking

The unsupervised trace ranking computes suspect scores with traces collected when the application failed, but with no previous traces. To compute scores with no *a-priori* knowledge, we employ an unsupervised algorithm for detecting outliers, called the k-nearest neighbor (k-NN) [68]. The k-NN finds outliers based on the distance between each data point to the k^{th} nearest neighbor in the data set. We define suspect scores by the distance to k-NN and compute the trace ranking as follows.

First, we determine the value of k, which is the threshold between normal and outlier traces. Too large values of k could cause normal traces to be marked as outliers, while too small values could miss true outliers. For example, assume that a fault affected t processes of the application so that their behaviors were substantially different from the others, yet similar to one another. In that case, any values of k less than t are likely to miss k anomalous processes. Since we have no

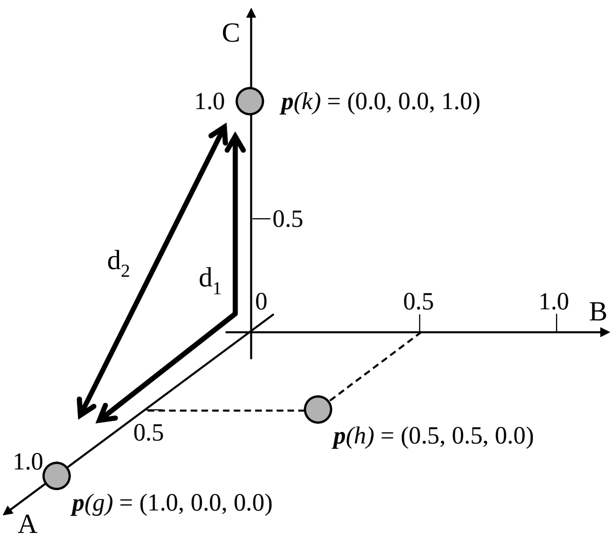


Figure 5.3: Comparison of the Manhattan and Euclidean distance metrics. The three points are the profile vectors for processes g, h, and k. The axes represent functions, A, B, and C, that the application executed. The arrows annotated with d_1 and d_2 indicate the distances between $\boldsymbol{p}(g)$ and $\boldsymbol{p}(k)$ defined by the respective Manhattan and Euclidean distance metrics.

knowledge on the actual number of affected processes, we cannot determine the minimum value that k should be greater than. However, since the assumption in this approach is that most of the processes in the application behave similarly, we can expect that normal processes can be clustered into a small number of tightly formed clusters. Thus, relatively large values of k would still be able to classify normal traces correctly. In fact, in our experimental studies, we have observed that the algorithm worked well for all k's larger than 3 and up to $|T|/4$, where $|T|$ is the number of traces, T. Thus, we use $|T|$ as the value of k in this study.

Next, for each trace g, we find k-NN, $g_k \in T - \{g\}$, by computing a sequence of traces $g_i \in T - \{g\}$, $1 \leq i \leq |T| - 1$ based on the distance from g.

$$T_d(g) = \langle g_1, g_2, \ldots, g_{|T|} \rangle, \text{s.t. } d(g, g_i) \leq d(g, g_{i+1}) \qquad (5.5)$$

Thus, g_k is the k-th component of the sequence above. The suspect score, $\sigma(g)$, for trace g is then defined as the distance to k-NN:

$$\sigma(g) = d(g, g_k) \qquad (5.6)$$

We compute the ranking of traces based on the $\sigma(g)$ and further localize the fault into specific anomalous functions by the third step described in Section 5.2.

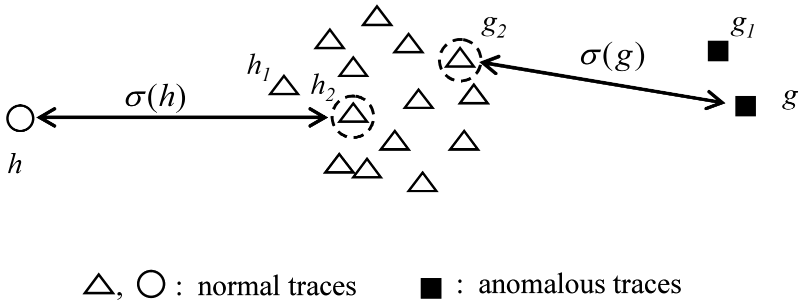


Figure 5.4: Example of the unsupervised trace ranking. Each object represents the trace of a failed run of an application process. The open triangles and circle designate that these objects are normal, while the closed rectangles are anomalous. g is one of the true anomalous traces, while h is a normal trace, even though it is an outlier. The arrows for g and h represent the distances to k-NN, where $k = 2$.

Figure 5.4 illustrates an example suspect-score computation where a majority of traces are located closely within a small area and three traces are far away from the majority. Assume that the right-most one, g, is a true anomaly; g_1 is the first and g_2 is the second nearest neighbor. Assume that we set $k = 2$. The suspect score of g is the distance between h and h_2, which is annotated with $\sigma(g)$ in the figure. As can be implied from the figure, $\sigma(g)$ is one of the largest in the traces depicted; thus, trace ranking results in a successful identification of the true anomaly, g. The left-most, on the other hand, trace illustrates a false-positive case; assume that the trace, h, is an outlier, but is a correct trace. Again, trace h_1 is the first h_2 is the second-nearest neighbor of trace h. The suspect score of h is the distance between h and h_2, annotated with $\sigma(h)$, and, although legitimate, it is also relatively large compared to those of the other traces. The resulting ranking causes the trace to be incorrectly marked as anomaly.

In our experimental studies, the unsupervised algorithm presented here have successfully computed high trace ranks for anomalous processes. However, as in the previous example, we also observed that some legitimate processes did receive a high suspect score; e.g., the master process in a master-worker system was given a high score, causing it to be considered as an outlier. Any such legitimate processes that have larger suspect scores than the true anomalous process are false positives for the problem analyst. We reduce the false-positive rate by using another ranking algorithm, which is described below.

5.1.2.3 One-Class Trace Ranking

The one-class trace-ranking algorithm computes suspect scores using previous traces in addition to traces collected when the fault occurs. We call the former *reference traces*, and the latter *trial traces* to avoid confusion. By using reference traces, the one-class algorithm aims to mark unusual, but correct traces as normal traces, thus reducing the false positive rate.

As reference traces, we use not only known-correct traces, but also any traces that have behaviors that are of little interest to the problem analyst, no matter whether they are correct or incorrect. For example, finding a behavior that commonly appears as a result of many classes of different faults would not help the analyst to localize a particular fault. We include such traces, if any, to improve the precision of localizing faults.

Note that our use of the term one-class classification is not correct in the strictest sense. One-class classification generally refers to classifying a single class and distinguishing it from all other data by only using sample data from the classified class [87]. In the domain of anomaly detection, the classified class means the set of correct behaviors of the application; thus, analysis using both correct and incorrect traces does not mean one-class classification. Below, we use the term in the sense where we detect any behaviors that are of interest using available traces, no matter whether they are correct or incorrect.

The one-class ranking computes a suspect score for each trial trace by taking the minimum of both the k-NN in the trial set and the nearest neighbor in the reference set. For each trial trace, g, we compute the suspect score, $\sigma(g)$, as follows. First, as in the unsupervised ranking, it first finds the k-NN in the trial set (i.e., $g_k \in T_d(g)$). Next, we find the trace, r_1, in the reference set, R, that is nearest to the trial trace:

$$r_1 = \operatorname*{argmin}_{r \in R} d(g, r) \tag{5.7}$$

Based on g_k and r_1, we define the suspect score, σ, for trace g by the minimum distances to them:

$$\sigma(g) = \min\{d(g, g_k),\ d(g, r_1)\} \tag{5.8}$$

By taking the minimum, we can ensure that all traces are given as low a suspect score as in the unsupervised-ranking case. In addition, if the reference set includes correct, but unusual traces, we can award low scores to traces similar to these unusual ones as well. The problem in unsupervised ranking is that a legitimate outlier can be assigned a high suspect score since it has few similar traces; we avoid such false positives by considering the distance to the nearest neighbor in the reference set as well.

Figure 5.5 illustrates an example of computing the one-class suspect score, which is based on the previous unsupervised example, but has several reference

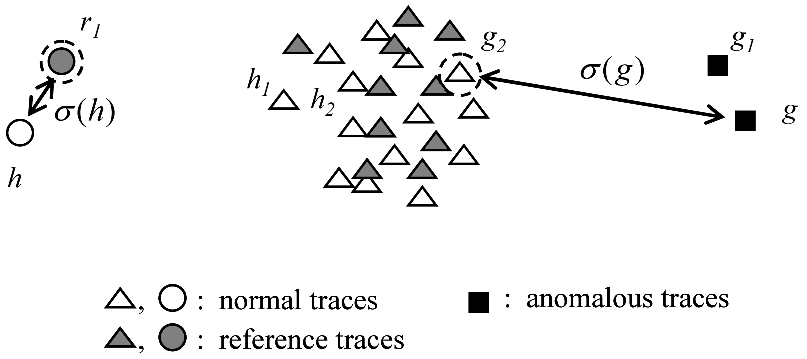


Figure 5.5: Example of one-class trace ranking. Each object represents a trace of an application collected from previous runs or from the diagnosed failed run. As in Figure 5.4, the open triangles and circle designate that the objects are normal, while the closed rectangles are anomalous. Unlike the unsupervised example, this case also includes reference traces, which are designated in gray.

traces added from previous runs (shown in gray). As we can see from the figure, $\sigma(g)$ has the same value as the unsupervised case since there are no reference traces close to g. Thus, we can assign a high trace rank to g in this case as well. However, unlike the unsupervised case, the suspect score of the unusual, but correct trace, h, has a closely located reference trace, annotated with r_1. Thus, we assign a low value to $\sigma(h)$, as indicated by the distance between h and r_1.

Scalability with Number of Distributed Processes Both the unsupervised and one-class algorithms scale well with the increasing number of distributed processes, because the suspect scores of all traces can be computed in parallel as follows. First, we compute the time profile for each process in parallel by exploiting the underlying distributed nodes. Each node computes the profile for the process that was running on the node. Next, each node exchanges the computed profile with the rest of the nodes. Such all-to-all data gathering can be performed in $O(\log n)$ steps, where n is the number of processes, using the recursive doubling algorithm [88]. The amount of data exchanged depends on the number of distinctive functions or call stacks appearing in the profiles. In our case studies, we used a 10-MB fixed-length buffer, and observed a typical number being in the hundreds. Therefore, the data size exchanged at each step would be as large as n KB, which would take less than a second to transfer among cluster nodes interconnected with a Gigabit Ethernet. Next, each node computes the suspect score for its process by using either the unsupervised or one-class algorithms, which takes $O(n)$ comparisons of the profiles. Finally, we gather the scores computed in parallel to a central location, and compute the ranking for the processes by sorting them. The scores can be gathered in $O(\log n)$ steps and sorted in $O(n \log n)$ steps. All in all, the dominant step in terms of computational complexity is the final sorting step; however, in practice, we expect that the profile exchange step would take longer even for as many processes as a million. In fact, sorting a million floating-point values on a machine with a dual-core 2-GHz CPU with 3 GB of RAM took only half a second. Overall, we expect that the profile comparison at each local node would be the most dominant step, and thus the localization time for the proposed algorithms would increase linearly with the number of processes.

5.2 Function-Level Localization

The final step of our analysis localizes the fault at hand to specific functions or call stacks. We will describe a function-level localization algorithm for each of the fail-stop and non-fail-stop faults. For the former, we assume that the previous step found anomalous processes whose last timestamps are substantially earlier than the rest. For the latter, we find anomalous functions or call stacks for the processes with the highest suspect scores. We present the problem analyst with the functions found in this step to assist further manual diagnosis.

5.2.1 Fail-Stop Problems

For a fail-stop fault, we present the problem analyst with the function that generated the last trace entry in the anomalous process identified by the previous step. The information should be useful for faults such as application crashes and deadlocks. For example, illegal memory access such as dereferencing `NULL` in C programs would result in a crash. Our analysis attempts to pinpoint the specific function that performed the illegal access. As another example, let us suppose that a process had infinitely blocked on a synchronization primitive (i.e., a deadlock); our analysis would locate the function that resulted in this infinite blocking. For non-fail-stop problems, we use the following analysis.

5.2.2 Non-Fail-Stop Problems

To localize a fault down to anomalous functions, we find the functions that contributed most to the ranking of traces in the last step. For all anomalous processes, we compute a sequence of the functions, $f_i \in F, 1 \leq i \leq |F|$, ordered by the difference to the nearest-neighbor process. The functions that have the largest differences explain why the process is an outlier; by presenting the function sequence, we can expect that the problem analyst can focus on more suspicious functions first, possibly resulting in faster determination of the root cause.

We compute the function sequence as follows. As previously described, the suspect score of trace g is defined as the distance between g and another trace h that is either the k-NN or the nearest-reference trace, depending on the specific algorithms employed to compute the trace ranking. Thus, from Equation (5.2), the suspect score, $\sigma(g)$, is defined as $\sum_{i=1}^{F} |\delta_i(g, h)|$, where $\delta_i(g, h)$ is the difference between the i-th components of $\boldsymbol{p}(g)$ and $\boldsymbol{p}(h)$. Based on the component-wise differences, we can compute the sequence of functions, $F_\delta(g)$, for process g as:

$$F_\delta(g) = \langle f_1, f_2, \ldots, f_{|F|} \rangle \text{, s.t. } \delta_i(g, h) \leq \delta_{i+1}(g, h) \tag{5.9}$$

5.3 Experimental Results

To evaluate how effective the approach based on outlier detection discussed above, we conducted a case study with a distributed cluster management system called SCore [39]. The installation of SCore on a public computational cluster at the Tokyo Institute of Technology has suffered from occasional failures with varying symptoms. Here, we will first present an overview of SCore, and then describe how we collected function-call traces from it. Finally, we will describe how we localized two representative faults: the network stability problem and the scbcast problem.

5.3.1 Overview of SCore Environment

SCore is a large-scale parallel programming environment for clusters, consisting of a single master node and multiple compute nodes. It provides job-execution facilities for several parallel-programming paradigms, including MPI, PVM, OpenMP, MPC++, and distributed shared memory, with the support of distributed scheduling, checkpointing, and migration. It is implemented mainly in C++ user-level code with several kernel-level extensions, and has a large code base; it has had more than 200 KLOC in 700 source files as of version 5.4.

The user of SCore submits a job to the central job scheduler with a description specifying its hardware and software requirements. The scheduler matches the requirements and available computing resources, and starts the job on the allocated resources. SCore uses a per-node daemon process, `scored`, to spawn and manage jobs on remote nodes. It monitors the jobs under its node, and redirects their output to the user's terminal.

SCore implements a heartbeat mechanism that periodically checks the liveness of the `scored` on each node. All `scored` processes running on the compute nodes and the sc_watch process on the master node form a ring-topology network by being connected to one another. The sc_watch process on the master node sends an initial heartbeat packet to its neighbor scored process. Each scored, receiving a heartbeat packet from a neighbor, forwards the packet to another neighbor, which should finally return to sc_watch through the ring network. sc_watch waits for the packets to be returned by another neighbor scored with a timeout of a specified duration (ten minutes, by default). If it does not receive any heartbeat packets for the duration, it assumes that a scored in the ring has failed and automatically attempts to kill and restart all scored processes. Note that the heartbeat mechanism does not know the specific daemon that caused the timeout or the reason for this.

In three months of monitoring the SCore environment, we witnessed several timeouts. The system administrator of the cluster quickly identified the root causes for some of the timeouts by manually examining various log messages and the status of hardware. We applied our framework to analyze the other timeouts, where manual efforts were not able to identify the root causes, using the tracing mechanism described below. We report the results of analysis in the following sections.

The platform for the cluster is as follows. It consists of 129 compute nodes with a dedicated master node, running the SCore v5.4. Each node runs Linux v2.4 (Red Hat v7.2) on dual Pentium III 1.4-GHz processors with 1 GB of RAM. The compute nodes are interconnected with both a 100BASE-TX Ethernet and Myrinet2000.

5.3.2 Function Tracing in SCore Environment

We take the function-call traces of all scored processes running on the compute nodes to analyze the SCore environment. We modify scored to write a trace statement into a trace buffer before and after each function call. To modify the program, we instrument each function-call site using a dynamic binary instrumentation tool called spTracer [59]. spTracer injects an instrumentation agent to the process in a similar manner as the process hijacking technique described by Zandy et al. [97]. Once injected to the target process space, it rewrites each function-calling instruction to a jump instruction to a *trampoline*. The trampoline is an instruction sequence that in turn calls the original target function as well as executes tracing routines before and after the call. The agent performs the instrumentation incrementally: once activated, it modifies each call site only in the currently executed function so that at every call site the process yields control to the agent. Each time the agent is activated again, it rewrites the call site to the jump instruction to the trampoline, and modifies the target function in the same manner as the caller. By incrementally following the control flow of the application, it reduces perturbation due to dynamic instrumentation, since doing all functions at once could pause the execution of applications for a non-negligible duration, especially for large programs.

We create a fixed-length circular trace buffer on an IPC shared memory segment; the instrumented scored writes a trace entry at each function call and return into the trace buffer. By limiting the length of the buffer to a fixed size (10 MB in this particular study), we avoid exhausting the free memory available on each node. Every time the traced process reaches the end of the buffer, it continues to write traces by overwriting the oldest entries. By allocating the chunk of memory for the trace buffer on a shared memory, we retain the contents of the trace buffer even if the application process suddenly exits. This property is especially important in our case study; common failures in the SCore environment did not cause the operating system to crash, but the scored daemons did.

As illustrated in Figure 4.1, each trace entry consists of five fields, including the trace type, function address, process ID, thread ID, and timestamp, with an optional variable number of fields. The type of trace is either call or return; the combination of the process and thread IDs uniquely designates the thread that generates the trace; the timestamp field holds the value of the processor-cycle counter; and the optional fields are used for recording argument and return values for particular functions, such as the return value from system calls. Our 10-MB buffer usually held a 10-minute execution of scored.

The automated localization starts operating once a failure is detected; for the SCore environment, we extended sc_watch so that when a timeout occurs (i.e., a failure), it dumps the contents of the trace buffer into the local disk on each node. The automated analysis in turn collects all dumped trace files from the compute nodes and start applying the analysis previously described.

While our tracing framework can be used for almost all types of applications with no *a priori* internal knowledge or special annotations, we make two assumptions to simplify recording two of the above trace fields: thread IDs and timestamps. First, the technique of obtaining thread IDs depends on specific thread libraries used in target applications; thus, we need to know how to collect thread IDs in the employed threading library. scored is a multithreaded program implemented with its custom threading library MPC++ [38]. In this experimental study, we extend spTracer to be able to trace MPC++ programs. However, in general, we can see that such use of a custom thread library is rare: For example, the majority of multithreaded programs for Unix systems may use the POSIX Threads API [13], where thread IDs can be obtained with the pthread_self function. Thus, by supporting a small number of commonly used libraries, we can expect that many applications do not require such application-specific extensions as MPC++ support for scored. Second, our current prototype takes timestamps by reading the processor-cycle counter for a minimal performance penalty being imposed. We use the timestamps for calculating the time spent in functions and computing the function profiles for traces, as defined by Equation (5.1). Here, we assume that the cycle counter increases regularly; therefore, the absolute duration for each increment of the counter must be the same. While this assumption has been valid on traditional processor architectures, this is not necessarily so on current modern processors with dynamic voltage and frequency scaling (DVFS), where the clock speed can change dynamically. While our experimental platform did not have the DVFS technology in place, we will explore alternative low-overhead timestamping techniques in future work.

5.3.3 Network Stability Problem

The network-link stability problem exhibited the following symptoms. The system stopped scheduling jobs, and `sc_watch` detected a timeout after ten minutes and restarted the `scored` daemons on all nodes without errors. As failures occurred multiple times in two months, it was imperative to find their causes.

Our earliest last timestamp approach described in Section 5.1.1 determined that the failure exhibited a clear fail-stop behavior. We identified that host `n014` stopped generating trace records more than 500 sec 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 issued an `_exit` system call. Figure 5.6 visualizes the trace on node `n014` with the Jumpshot tool [15]. In the figure, each rectangle represents a function call with the execution time indicated by the width. However, we could not find the caller of `score_panic` because the trace buffer was of fixed size. 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 of `scored` to find that `score_panic` and `myri2kIsSendStable` were called from `freeze_sending`. The `scored` calls `freeze_sending` each time a running thread yields its execution so that there are no more in-flight messages on the node's Myrinet-2000 NIC. The `freeze_sending` repeatedly checks the existence of in-flight messages by calling `myri2kIsSendStable` in a loop until it returns true. However, if the check count exceeds one million, `freeze_sending` assumes that a network error has occurred and aborts by calling `_exit`. Our findings analyzing traces matches the case when in-flight messages were in fact never completely delivered. We reported the results of our analyses to the SCore developers. Their feedback confirmed our findings. They had observed such symptoms in Ethernet-based net-

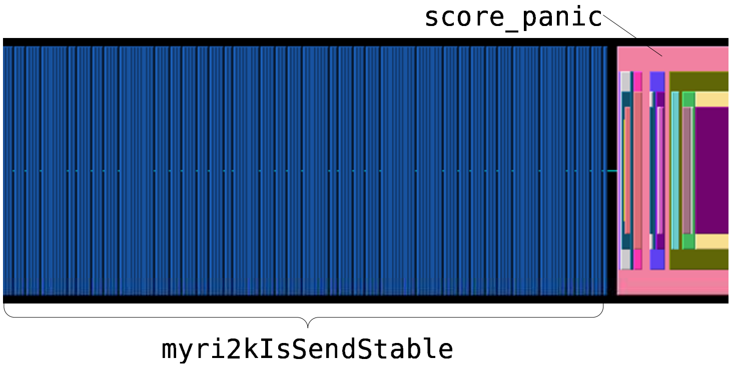


Figure 5.6: `Scored` **trace on node** `n014` **visualized with Jumpshot [15].** Each rectangle represents a function call with its execution time indicated by the width. Nested rectangles mean nested function calls. The blue rectangles illustrate the large number of calls to `myri2kIsSendStable`, and the pink one the call to `score_panic`.

works, but our report informed them that there is a similar problem in Myrinet networks.

These results illustrate the difficulty of building robust distributed systems discussed in Section 1.1. That is, our finding on faults in the mechanism of detecting timeouts was behavior that had been unexpected by the SCore developers. They have changed timeout detection so that it was more conservative in the latest versions of SCore. This change included a call to function `usleep` to suspend execution for 100 ms each time `myri2kIsSendStable` is called. The SCore development team informed us that the aim of this change was to adapt the timeout mechanism of `freeze_sending` to the modern faster CPUs. They suggested that the old timeout mechanism should work in our environment, which uses Pentium III 1.4GHz CPUs. However, with the results of our analysis, we suspect that either their estimates could have been wrong or there is another unknown bug in SCore.

In summary, our automated analysis accurately helped identify incorrect assumptions or the existence of yet-unknown bugs. Although we only inspected a small part of SCore's source code, the analysis automatically identified the crashed node and the suspicious function, `score_panic`, with no *a priori* knowledge on the internals of SCore. Furthermore, the number of functions that needed to be manually examined was only three—`score_panic`, `myri2kIsSendStable`, and `freeze_sending`—out of the total of 2319 functions in `scored`.

5.3.4 Scbcast Problem

Another problem occurred when an SCore component called scbcast stopped responding to requests from scored. The scbcast component serves as a broadcasting proxy for monitoring information collected from all scored daemons running on compute nodes. SCore provides a client program that connects to scbcast to retrieve the information, rather than contacting individual scoreds directly.

While this technique eliminates some of the load from scored processes, it introduces another potential point of failure. Our automated analysis that is described below revealed a bug in the scored that caused the entire SCore system to stop functioning. We have identified that the bug was initiated by the anomalous behavior of an scbcast process that stopped responding to incoming requests from scoreds. The automated restart by sc_watch was ineffective to rectify this case: Every time scored daemons were restarted, they immediately stopped functioning as the anomaly in scbcast persisted even after the scoreds were restarted. As a result, the system administrator had to manually restart all SCore components without knowing which component was the root cause of the problem. Here, we will explain how our analysis revealed that the problem was caused by the combination of the bug in scored and the anomalous scbcast.

First, we decided that the problem did not exhibit a fail-stop behavior. All nodes terminated at similar times—the maximum difference between the last timestamps was only 20 sec; the earliest process terminated less than a second earlier than the second earliest. Therefore, we decided that the problem was a non-fail-stop problem, which led us to use the trace-ranking algorithms.

To identify the anomalous scored daemons, we first applied the unsupervised ranking algorithm based on call-stack profiles with the results shown in Figure 5.7. For each point, the x-coordinate corresponds to the name of the node where a scored daemon ran, and the y-coordinate corresponds to its suspect score. As we can see, the suspect scores of node n116 and n129 are substantially higher than those of the other nodes. To further narrow down anomalous nodes, we used the set of traces in the previous network stability problem as reference traces. Figure 5.8 shows the results for the suspect scores computed with the one-class algorithm. Here, only node n129 has a substantially higher score than those of the others, while node n116 has been assigned a score as low as the others. These results allowed us to focus further on the behavior of node n129.

To find the cause of the anomaly in node n129, we applied function-level localization that finds the call stacks making the largest contributions to the suspect score. Figure 5.3.4 shows the contribution of each call stack, where the x-axis corresponds to a call stack (i.e., f_i in Equation (5.9)), and the y-axis corresponds to its contribution (i.e., $\delta_i(g, h)$ in Equation (5.9)). As we can see, the call

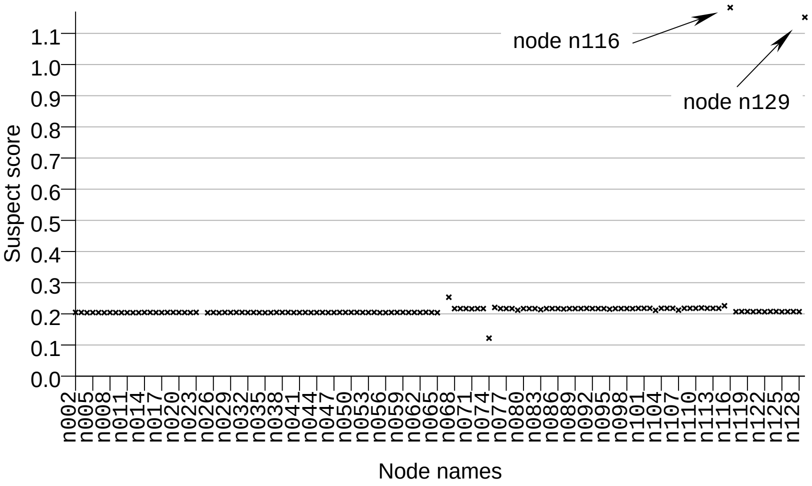


Figure 5.7: Suspect scores in scbcast problem computed by unsupervised algorithm. Each point corresponds to a `scored` daemon, where the x-coordinate represents the name of its node, and the y-coordinate its suspect score.

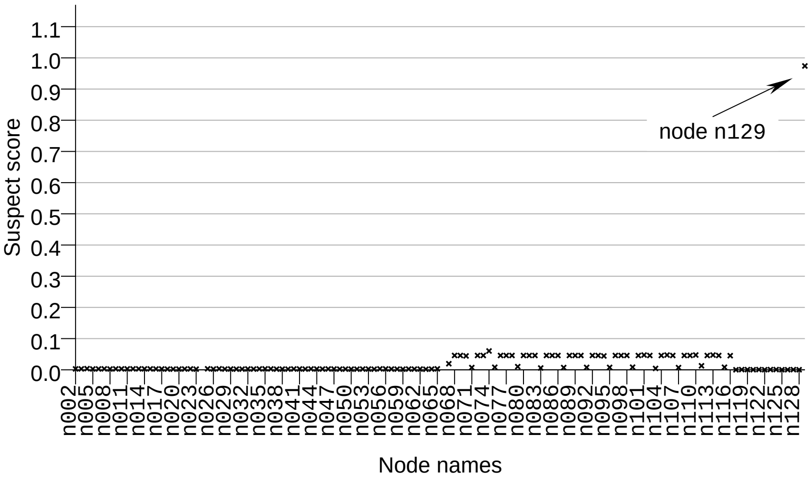


Figure 5.8: Suspect scores in scbcast problem computed by one-class algorithm. Each point corresponds to a `scored` daemon, where the x-coordinate represents the name of its node, and the y-coordinate its suspect score.

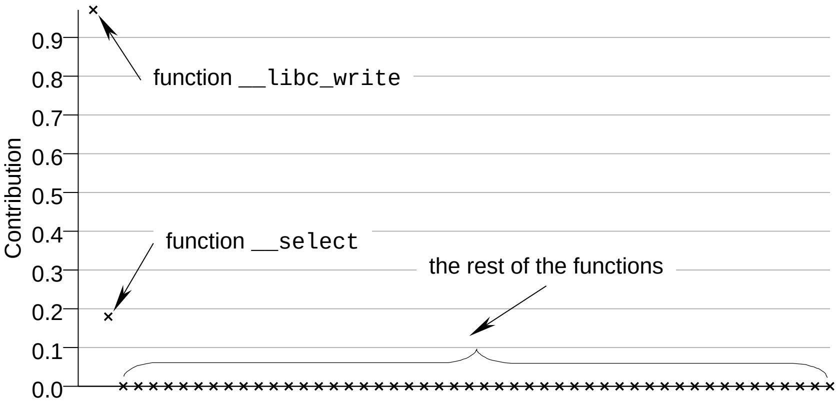


Figure 5.9: Contributions of functions made to the suspect score of `n129`.

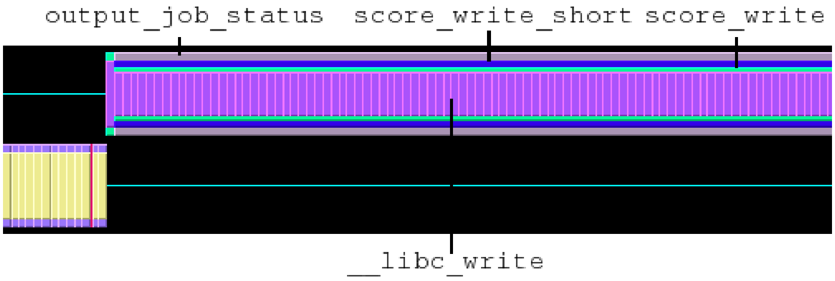


Figure 5.10: Fragment of `scored` trace from node `n129`.

stack annotated with `__libc_write`, which corresponds to `output_job_status` → `score_write_short` → `score_write` → `__libc_write`, made a substantially larger contribution to the suspect score than those of the others. Visualization with the Jumpshot tool further revealed that `scored` entered `score_write` and started calling `__libc_write` in a loop, never returning from `score_write`. Figure 5.10 illustrates the particular part of the entire trace. As a result, `sc_watch` received no patrol message for more than the timeout duration (10 min), and restarted `scored` processes. However, the restarted `scored` daemons exhibited the same behavior: Node `n129` immediately blocked again on the call to `__libc_write`. The problem was fixed only after all SCore services were restarted manually. We can see that the `scored`on `n129` was a true anomaly; our outlier-detection approaches were able to locate it, and the maximum-component approach found the correct symptom in the `scored` functions. Inspecting all traces manually would have required substantial amounts of effort.

Examination of the source code revealed that `scored` was trying to write a log message to a socket connected to the `scbcast` process. While it usually completes immediately, it had not returned for 11 min in this particular case. The __libc_write function kept blocking instead of returning even though the amount of data to be sent was relatively small (the maximum message size was 512 bytes). A write to a socket blocks if the message does not fit into the send buffer of the socket [83]. In turn, the send buffer can fill up if the remote peer (scbcast) does not read from the socket for some time. Therefore, if scbcast froze or the network connection to its node went down, all scored processes that sent job statistics to scbcast would eventually have entered an infinite loop.

Note that scbcast was not traced in this case study. As a result, we cannot determine why it stopped reading data from the scored packet. However, part of the problem is on the scored side—we believe that scored should be better prepared for handling error conditions when writes to the scbcast connections start blocking. Entering an infinite loop on such an event should be considered as a bug and our analysis proved useful in identifying its location. We reported the bug and it is being fixed for a future release of SCore.

To understand why node n116 is an outlier in the unsupervised algorithm, we found that the call stack making the largest contribution to its suspect score was fepio_read → ult_idle_hook → __select. Node n116 spent most of its time in fepio_read, repeatedly calling ult_idle_hook and __select. From the trace records on the repeated calls to __select, we suspected that it had been expecting incoming messages that had never arrived. By consulting the source code, we identified that it had in fact been waiting for a message from the scored on n129, which had been blocking on __libc_write. Thus, the outlying behavior of n116 was caused by the true anomaly in n129. To understand why the one-class algorithm gave 116 as low a score as the other nodes, we examined the reference trace that was the closest to 116, and found that it also spent most of its time in the same call stack. This result indicates that this behavior is common to failures in SCore; indeed, further examination of the source code revealed that this behavior appeared every time each scored failed to forward keep-alive messages at regular intervals. Since such unusual, but common behavior is not as relevant to the problem at hand as that more specific to the problem, our one-class algorithm proved effective in achieving more precise localization.

5.4 Summary

We have presented our automated approach to the localization of faults based on the detection of outliers in function traces. For fail-stop problems, we compare the last timestamps and finds the trace with significantly earlier timestamps. Deadlocks and crashes would often exhibit fail-stop behaviors. For non-fail-stop problems, we compare the time spent in each function to find outliers in failed runs. We compute trace ranking with the unsupervised and one-class algorithms. The former operates only with the set of traces collected from the problem at hand, while the latter improves the precision of analysis by avoiding uninteresting traces that do not need to be considered as outliers. We have presented the results of our several-month case study in a real production cluster with two sample problems: the network stability and scbcast problems. These results demonstrated the effectiveness of our techniques in fault localization.

The case studies presented in this chapter also indicate a limitation with the outlier-detection-based approach. The fact that the one-class ranking algorithm successfully eliminated uninteresting behaviors from the trial traces implies the effectiveness of comparing behaviors across time. However, it also presents a new problem on maintaining a possibly very large collection of past behaviors. In the scbcast problem, we used as a reference a set of traces that was collected when a failure occurred; however, only using traces from one failure might prove insufficient for other scenarios. For example, if traces from N different failure instances are collected, we would end up having an N-times large set of reference traces. Furthermore, larger-scale systems would magnify the problem as clusters would be ten times larger, or even scales of a hundred times compared to our testbed would be commonplace.

This observation motivated our second approach, which is based on historical similarities in process behaviors. It efficiently summarizes past behaviors as *execution models* by learning function call behaviors during normal operations, and finds anomalous behaviors by comparing failure traces with the derived models. We describe the algorithm and experimental studies involved in this approach in Chapter 6.

Chapter 6

Model-Based Fault Localization

To localize faults in distributed systems, our second approach learns normal behaviors from execution histories and detects deviations from learned past behaviors. Specifically, as in the outlier-detection-based approach, it collects the per-process function-call traces of target systems. Using the collected traces, it performs two phases of analysis: pre-failure model derivation, and anomaly detection in the failure traces using the derived model. The first phase, using traces collected under normal operations, automatically derives an *execution model* that reflects the normal function-calling behaviors of the target system. When a failure actually happens in the learned system, the second phase finds violations in the traces against the derived model. By doing so, it locates specific calling sequences in the traces that are highly correlated with the failure.

The key challenge in attaining such model-based localization of faults in large-scale computing systems is then how to perform the model derivation and fault localization using models in an automated, scalable manner. Learning accurate models requires to identify self-similar, repetitive behaviors in target systems. While repetitive behaviors can be assumed to exhibit historical similarities, automatically identifying similar behaviors in raw function-call traces is not trivial. Often raw traces are very large in size since typical processes in our target domain do not start and stop frequently as in desktop applications, but rather always run with little downtime. Furthermore, as previously stated, such model derivation must work at scale. For example, the InTrigger distributed computing platform consists of six clusters distributed among Japanese universities and research laboratories. Decentralization and scalability are key factors in analyzing faults that occur in such distributed environments.

To tackle this challenge, we exploit two observations on typical software architectures and fault characteristics in distributed systems. First, many of the distributed software for clusters and Grids, such as batch job schedulers and parallel file systems, consist of processes that employ an event-driven architecture, where

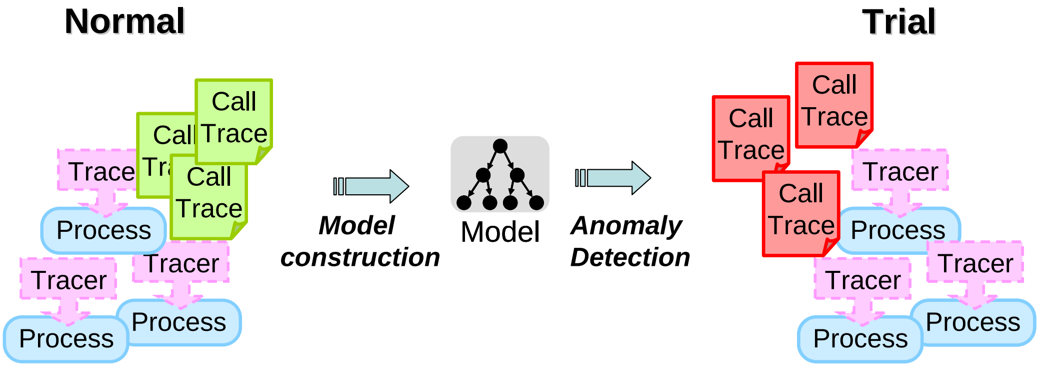


Figure 6.1: Overview of Model-Based Fault Localization. Model-based localization derives a model that represents the normal function-calling behaviors of the processes of the target system using normal traces. It localizes a fault by comparing the derived model with the trial traces.

several different event-processing routines are multiplexed into a single event loop. For example, a batch job scheduler for clusters could employ a daemon process on each node whose responsibilities include job monitoring under the node and handling requests from the master job scheduler. A typical software architecture for such purposes would model job-status changes and incoming requests from the master as events, and consist of an infinite loop of event-processing routines. Second, we have observed in our case study presented in Chapter 5 and the results by other researchers [71] that many anomalies that were observed to manifest themselves over distributed nodes also exhibited locally observable deviant behaviors. For instance, one bug reported by Reynolds et al. [71] caused an event handler function not to be called, failing to serve incoming requests; such behaviors could be detected only using locally observable information, namely function-calling behaviors in this particular example.

Based on the above observations, our approach models system executions by first learning per-process function-calling behaviors, and then aggregating them into a single model that represents the behaviors of the entire system. Specifically, for each member process, we generate a concise per-process model called the *process model* from its function call traces. Unlike the previous approaches that attempted to reconstruct distributed flows by matching distributed traces [8, 17, 42, 60, 71], we only use local information to generate the process model to eliminate centralized bottlenecks.

To derive the process model, we first decompose the overall function traces into sub-traces, or *execution units*, based on their associated event sources, and then derive a model for each event source. This paper focuses on network events among various event types, which we believe would be the most important events in distributed computing systems. The event source of network events is their connection; thus, we treat a sequence of function calls corresponding to the same network connection as a single execution unit. Next, for each connection, we derive a model by constructing a call tree consisting of every function appearing in its associated units, and assigning each function an estimated probability of appearance. We estimate the probability of a function by dividing its number of occurrences by the total number of occurrences of the execution units for the same connection. For instance, if a function always appears when a message arrives at a connection, we give the function a probability of 1. Creating separate models for different connections improves the accuracy of estimating probabilities, since different connections are likely to have different function-calling behaviors. Finally, we derive the *global model* by merging process models whose processes are inferred to have played the same role in the system.

We aid human analysts in localizing the root cause of a fault by comparing its traces with the derived model. Given failure traces, we first decompose them into execution units as in the model derivation. For each execution unit, we compute a *suspect score* that quantifies how likely a particular part of the call traces was correlated with the failure. Functions with low probabilities in the model yield high suspect scores when called in failures. Frequently called functions in the model also yield high scores when not called. Finally, we report execution units ranked with suspect scores to the human analyst, narrowing further manual localization down to a small part of the overall system.

This approach is especially effective in localizing program logic bugs For example, assume that an application performs an operation when receiving user requests. Typical request handling involves executing certain functions depending the types of the requests: every time a user request arrives at the application, some functions for the particular request type should be called. However, if the application failed to correctly serve a request, some of the functions for the request type might not be called. The model-based localization could detect such behaviors as anomalies; the problem analyst would be able to determine the source of the failure automatically.

For example, a failure occurred at the Tokyo Stock Exchange on February 8, 2008 caused a part of its commercial services unavailable for three consecutive days [89]. It was reportedly caused because a certain part of newly allocated memory was not initialized. While we do not know the internal logic of the memory initialization, it is likely that a certain function related to initializing the memory area was not called only at the failed case. These behaviors could be detected as anomalies by generating a model reflecting the correct memory initialization and comparing the failure traces with the derived model.

Both our modeling and fault localization operate in a mostly decentralized fashion. In the model-derivation phase, only the derivation of global models requires a centralized operation that is globally coordinated, while the derivation of process models analyzes raw traces in parallel using the same set of nodes as the target system. Once the global model is derived and deployed to each local node, our fault localization requires no remote operations. Therefore, our method can achieve higher scalability compared to the previous approaches based on centralized algorithms [8, 17, 42, 60, 71].

We applied our proposed method to localizing a known nondeterministic bug in a distributed job manager. Experimental results on a three-site 78-node distributed environment demonstrated that our method quickly locates an anomalous event that is highly correlated with the bug. Specifically, without our automated trace analysis, we would have needed to examine all traces of 78 nodes accounting for a complete 70-second run. Our analysis narrowed the localization only to traces from two nodes for less than a second, significantly reducing the burden of fault localization.

The rest of this chapter explains the details of model derivation in Section 6.1 and fault localization using models in Section 6.2. We then introduce our prototype implementation of the proposed technique in Section 6.3, and experimental evaluation results using the prototype in Section 6.4. We conclude this chapter with a summary in Section 6.5.

6.1 Model Derivation

Our execution modeling aims at detecting program logic anomalies. A logic anomaly is a situation where an intended operation is not performed or a non-intended operation is performed. Such misbehaviors often lead to different function coverage. For instance, a bug discussed in [71] caused an event-handler function not to be called. Another example is a bug that caused a hang in a distributed job manager discussed in Section 6.4. The bug caused a function to be called that had never been called in normal operations.

We derive the execution model so that it can find quantitative differences between traces with such logic anomalies and normal traces. To do so, we consider the following two classes of functions as having higher levels of suspicion:

- Functions that are rarely called in normal operations, but are called when a failure occurs and

- Functions that are often called in normal operations, but are not called when a failure occurs.

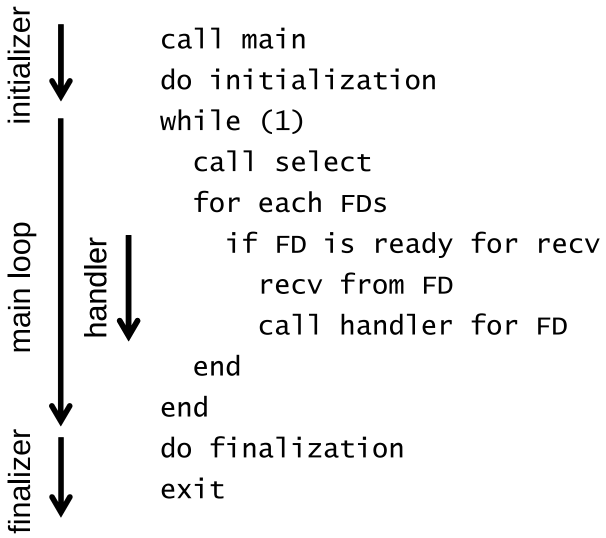
Our model quantifies how such properties hold between normal and failure traces with the estimated function-call probabilities. We compute the probabilities by obtaining known-normal function-call traces from the same system under normal operation states. Using the collected normal traces, we derive the execution model via the following three-step process: 1) decomposition to execution units, 2) derivation of the process models, and 3) derivation of the global model. The result of this process is a global model consisting of multiple process models, each of which reflects the normal function-call probabilities of a particular process group. The rest of this section describes the details for each step.

6.1.1 Function-Call Tracing

Similar to the outlier-detection-based approach, we use per-process function-call traces for learning execution models. A trace entry consists of four fields—type, timestamp, caller, and callee—and a variable number of optional fields. The type field consumes one byte and designates the type of the entry; there are currently three types: call, return, and exception. To simplify modeling and anomaly detection, we also encode unique identifiers for particular functions such as `connect` and `recv`. The timestamp field records the value of the CPU cycle counter, and consumes seven bytes. The caller field records the address of the caller and the callee that of the callee. The size of the fields depends on the architectures: two bytes on the x86 architectures and six bytes on its 64-bit extension x86-64. Note that while the x86-64 architecture allows 64-bit addressing, the current available products do not use the two top-most bytes [1, 37]. We omit those two bytes to reduce the size of traces. The optional fields encode the parameter and return values for particular functions. For example, we record the value of the file descriptor parameter of the `recv` system call to accomplish the trace decomposition described in Section 6.1.2. The largest optional field in our current implementation is 128 bytes for recording ready file descriptor numbers on returns from the `select` system call. Overall, the total size of a trace entry ranges from 16 to 148 bytes.

6.1.2 Decomposing Traces into Execution Units

Once function traces are obtained, we decompose them into sub-traces, or *execution units*, based on their associated events, assuming that the target system employs an event-driven architecture. By separating event-handling routines into different execution units, we aim at improving the resulting accuracy of the derived model. Of the various events in distributed systems, this dissertation focuses on network events such as arrivals of incoming messages since they are one of the most representative classes of events in distributed systems. Figure 6.2 illustrates



```
call main
do initialization
while (1)
  call select
  for each FDs
    if FD is ready for recv
      recv from FD
      call handler for FD
  end
end
do finalization
exit
```

Figure 6.2: Example of trace decomposition into execution units.

an example of such trace decomposition. Note that a program can include regions that are not related to any meaningful network events. For example, the function-calling behaviors in the program initialization and finalization parts should not depend on any incoming messages. We also identify such parts as different units.

We will now discuss how we decompose traces into execution units by the following three-step process of automated trace and program analyses. Here, we describe an algorithm for programs that use the `select` system call for event multiplexing. It should also be able to be easily extended to other event-processing frameworks, such as polling.

Step 1: Identification of event loop Since we assume that our target system uses `select` to multiplex message-handling routines, we can also assume that the event loop must include calls to `select` and `recv`. Based on this, we can find the event loop as follows. First, we detect loops at run time by detecting recurring call stacks in the function traces. We consider the call sequence inside the recurring stacks as a loop body trace. Next, of the detected loops, we locate the loop that has calls to `select` and `recv` in its body by analyzing the program source or binary code, and determine this as the event loop.

Step 2: Identification of handler units We decompose the traces inside the loop at every call to `accept` or `recv`, and determine a sequence of trace entries from a call to the next as a *handler unit*. The trace decomposition at `recv` calls assumes that a single event-handling routine starts by first receiving a message and performs any operations according to the content of the received message without receiving further data from the connection. Note that a `recv` system call can only return a partial message when the given receive buffer is too small to copy the whole message. Thus, a typical use case of the API repeats a call to `recv` until no more data is available on the connection. We treat consecutive calls to `recv` with the same file descriptor as a single `recv` call.

Step 3: Identification of other units The part of traces from the very beginning of the program to the start of the event loop should represent the initialization of the program, which typically includes such operations as listening-port setup and event-handler registration. We call the part the *initializer unit*. In addition, the function calls made after the event loop until the termination of the process should represent its finalization; we call this part the *finalizer unit*.

6.1.3 Process-Model Derivation

A *process model* consists of the call trees of the functions in the decomposed execution units. We derive the process model in the following automated means of analysis. Note that this model is derived in the same node as each target process, making it completely decentralized.

First, for each unit, we construct a tree representing function calls from the starting function of the unit. Each node n corresponds to a function call annotated with its call site s and callee function g, denoted as $s \to g$. Call site s denotes a unique location in the parent node's function, f. The path from the tree root to each child node corresponds to a function call stack executed at its trace-collection time. We allocate different nodes to calls from different call sites to the same function, i.e., if $n = s_1 \to g$ and $m = s_2 \to g$, then $n \neq m$. However, we allocate a single node to multiple calls to the same function from the same location, to avoid the tree from growing excessively large due to a large number of loop iterations.

Next, we merge the handler units based on their associated connections so that the process model has a unique sub-model for each event source. We define the equivalence of connections by their call stacks to the connection-establishing functions, including `bind`, `connect`, `listen`, and `accept` of the Berkeley Socket API. Specifically, let h be a handler unit, and c be the connection from which the unit received a message. Let s be the call stack to one of the connection establishing functions for c, denoted as $s = \{n_0, \ldots, n_k\}$, where the stack originates from call tree node n_0 and ends with n_k. Let $S(c)$ be the set of all the call stacks, s, associated with connection c, i.e., $S(c) = \{s_i\}$. We assume that two connections, c_1 and c_2, are equivalent if and only if their associated call stacks are the same, i.e.,

$S(c_1) = S(c_2)$. Based on this relation of equivalence, we categorize handler units into multiple groups where handler units in the same group have an equivalent connection. We create a single call for each group tree by merging the trees of the member units.

The observation behind the above merging of handler units is that if two connections are established by the same call stacks, we could expect that the handler units associated with them should include similar function calls. This is not necessarily the case; for example, if multiple connections are established by a single call site to `connect` in a loop with different socket descriptors, and these connections are actually related to different roles in the program. However, we expect that such a program structure would be rather rare; in fact, our case study presented in Section 6.4 exhibits no such behavior.

Finally, we annotate each node of the call trees by its estimated call probability. We estimate the probability of a node by counting the number of occurrences of execution units where the call-stack path of the node appears. Let f and g be calls in a tree where f is the parent of g. Also, let n_f and n_g be the number of execution units where f and g appear. We estimate the occurrence probability of g as $p_g = n_g/n_f$. Note that we do not consider how many times a node appears in a unit (i.e., frequency), but only whether it appears or not (i.e., coverage). Thus, n_g is always less than or equal to n_f.

6.1.4 Global-Model Derivation

Once process models are created, we gather them to a central location, and merge them into a single *global model* so that it includes the function-call behavior of all member processes. The reason to merge process models is for scalability with respect to the number of processes. The space cost to retain individual models, e.g., for hundreds of thousands of processes would be prohibitively high. For example, in our case study explained in Section 6.4, the size of process models was approximately 40 KB, reaching 4 GB with a hundred thousand processes. However, there would be significant duplication in learned models; not all processes would perform different operations, but several processes would play the same role in the system, thus generating similar function-call traces. For example, typical distributed software for clusters would employ a tree-style network topology with varying tree heights to organize each member process, where each node would have a different role depending on its depth in the tree. A leaf node process would only communicate with its parent process, while an internal node process would communicate with both its parent and child processes. The tree root process would be responsible for overall system management, such as responding incoming user requests and dispatching requests to the child nodes.

To derive a concise global model where these duplicated behaviors are removed, we categorize processes into groups where their roles could be expected to be the same inside each group. We infer the role of each process by the network connections established in its initializer unit. For example, in the tree-topology organization, the root process would establish connections by accepting requests from its children, while the leaf processes would do so by connecting to their parent. By looking at the call stacks that established connections during the process-initialization stage, we define the equivalence relation between processes, and derive a single model for equivalent processes. More specifically, we identify such call stacks in the initializer unit by locating calls to `bind`, `connect`, `listen`, and `accept`. Let $S(p)$ be the set of those stacks of process p. We call $S(p)$ the *process signature*, and define the equivalence of processes p and q by the equality of their signatures $S(p)$ and $S(q)$.

Figure 6.3 illustrates an example process grouping in a master-worker system. The master process, which performs operations written in the nearby rectangle, alone forms a process group. The italicized lines, including listening to connections from clients and workers, denote the process signature calls for the master process. The three worker processes, on the other hand, form a single process group, since they execute the same signature functions, i.e., connecting to the master.

Once the process models are categorized into groups, we merge the models in each group into a single process model as follows. First, we merge all initializer and finalizer models by aggregating the respective trees and annotating each tree node by the mean probability of the originates nodes. Next, we merge the handler models by finding matching pairs as in the handler-unit grouping in the process-model derivation. More specifically, for each handler model, we determine whether there is another handler model that has the same associated connection. If such a model is found, we merge the pair of models into a single handler model in the same way as the initializer and finalizer models. Otherwise, we copy the model to the resulting merged model as is.

6.2 Model-Based Fault Localization

Once the global model for a target system is derived, we deploy it to all local nodes so that following model-based localization can be applied in a decentralized way. When a failure occurs in the system, we compute a *suspect score* that quantifies the correlation of an execution unit in the traces with the observed failure by comparing the unit with the pre-deployed model. Our scoring algorithm described below yields high suspect scores to units whose behaviors have greatly deviated from the learned normal model. Finally, we gather the scores to a central location and report the scores sorted in decreasing order to the problem analyst so

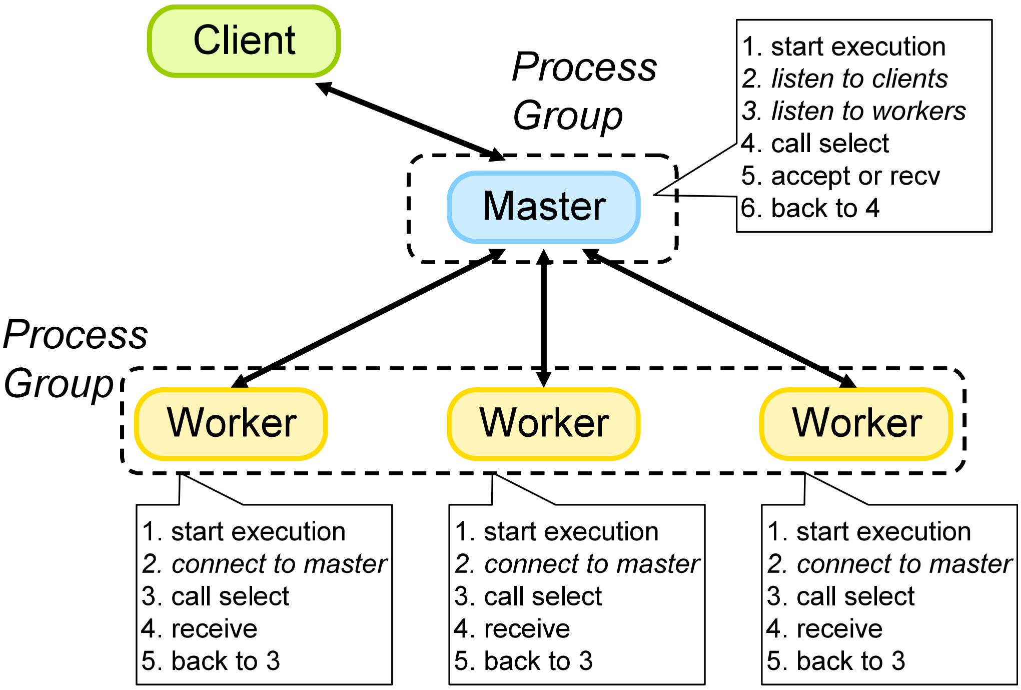


Figure 6.3: Example process grouping in master-worker system. Each of the master and worker processes performs the operations described in the nearby rectangle, where the italicized lines denote the process signature calls.

that more suspicious parts of the program can be prioritized in further localization.

6.2.1 Suspect-Score Calculation

We compute the scores by using the following three-step decentralized process:

Step 1: Decomposing traces Decompose the traces into execution units in the same way as in model derivation.

Step 2: Finding corresponding process model Find the process group with the same process signature. If no such corresponding process group is found, report the process as an anomaly to the analyst. If found, proceed to the next step.

Step 3: Finding corresponding execution models For each trial unit, find the corresponding execution model in the process model as follows. For both initializer and finalizer units, locate initializer and finalizer models, respectively. For each handler unit, find the handler model with the equivalent connection, where the equivalence of connections is defined in the same way as is done in the model-derivation phase. If no corresponding handler model is found, mark the unit as an anomalous unit. If found, compute its suspect score by comparing the unit with the found model.

For anomalous processes and units whose corresponding models cannot be found in the above steps, we highlight them by assigning a maximum suspect score of 1, since such unknown behaviors should be of interest for the problem analyst. We compute the other units' anomaly scores with the algorithm described below.

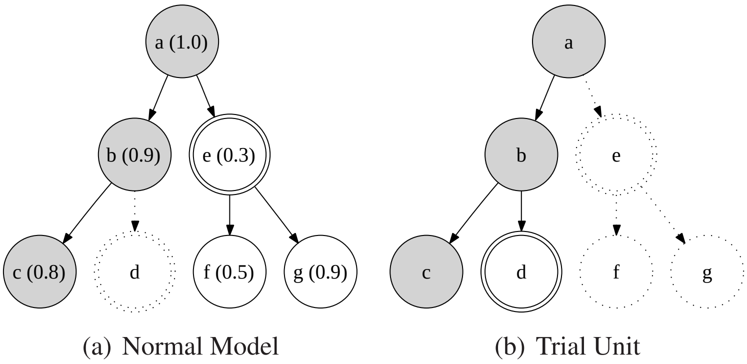
We consider the following functions more suspicious: 1) those with higher probabilities of occurrence, but not called when a failure occurs, and 2) those with lower probabilities of occurrence, but called when a failure occurs. Thus, our goal in designing the suspect-score calculation is that it gives higher values to those functions considered more suspicious.

First, we construct a call tree from the given trial unit, u, in the same manner as in the model derivation. Next, we compute *commonality* and *minimum difference* sets of the nodes in the model and trial tree. Let M and T be the sets of nodes that appear in the model and trial units, respectively. We define commonality, $M \cap T$, by the standard definition of the set commonality. Let us introduce minimum difference, $M \oplus T$, to filter out duplicate contributions to the suspect-score calculation. For example, suppose that there is a call stack $f \rightarrow g \rightarrow h$ in a model, where g is called by f and h is called by g. In this case, if g is not called in a trial unit, h must not be called either. Since the absence of h is a direct effect of the absence of g, the former absence should not be of interest in assessing the difference between normal and anomalous executions. Based on this observation, we define the minimum difference as a set of nodes that only appear in either of the two sets, but excluding nodes whose parent is also included in the minimum difference set. For example, in Figure 6.4, the commonality of the trees includes the nodes labeled a, b, and c; the minimum difference only includes nodes d and e.

Next, we define an *effective node set*, E, as the union of the commonality and minimum difference sets, i.e., $E = (M \cap T) \cup (M \oplus T)$, and call the nodes in E *effective nodes*. For every effective node $n \in E$, we compute suspect score $\Delta(n)$ as:

$$\Delta(n) = \begin{cases} 1 - p(n) & \text{if } n \in M \cap T \\ p(n) & \text{if } n \in M \wedge n \notin T \\ 1 & \text{if } n \notin M \wedge n \in T \end{cases} \tag{6.1}$$

For example, if a function was called 90% of the time when the system was operating normally and was also called when a failure occurred, we give a Δ of 0.1 to that node. Node b in Figure 6.4 illustrates such a case. In contrast, if that function



(a) Normal Model

(b) Trial Unit

Figure 6.4: Sample normal model and its trial unit. In the left tree, the value in the parentheses of each node shows its estimated probability. Dotted edges and nodes indicate that such nodes and edges do not exist in that tree. Gray nodes indicate commonality, while nodes with double lines indicate the minimum difference.

was not called in the given trial unit, we give Δ of 0.9. This scoring scheme meets the design goal for suspect scores.

Finally, we define suspect score $\Delta(u)$ for given unit u as:

$$\Delta(u) = \frac{\sum_{n \in E} \Delta(n)}{|E|} \tag{6.2}$$

In other words, we use the average of the scores of all nodes in the commonality and minimum difference sets as the suspect score of the unit. Since we only consider the effective nodes, we avoid having non-interesting calls affect the overall scoring. For example, we compute the suspect score of the trial unit in Figure 6.4(b) as:

$$\begin{aligned} \Delta(u) &= \frac{\sum_{n \in E} \Delta(n)}{|E|} \\ &= \frac{\Delta(a) + \Delta(b) + \Delta(c) + \Delta(d) + \Delta(e)}{5} \\ &= \frac{0.0 + 0.1 + 0.2 + 1.0 + 0.3}{5} \\ &= 0.32 \end{aligned} \tag{6.3}$$

6.3 Prototype Implementation

To collect function-call traces, we implemented a tracer for C and Python programs as well as a non-blocking concurrent trace-buffer pool. The buffer pool allows both traced processes and trace readers to access trace data in a concurrent, non-blocking fashion. Below, we describe their implementation details.

6.3.1 Trace Collection

Our current implementation supports tracing of function calls in C and Python programs as well as dynamic library calls. For tracing C programs, we currently use the compile-time function-call instrumentation available in the gcc compiler, which requires recompilation of the traced program by the gcc compiler. We are planning to use binary instrumentation tools, such as Dyninst [11], for greater flexibility.

For tracing Python programs, we use the debugging API, `sys.settrace`, which is available in the standard Python implementation. Almost no modifications to the traced program are required; we use our own Python code that sets our tracing functions to be invoked each time when the traced process makes calls and returns, and then calls the original starting function of the target program.

For tracing dynamic library calls, we use the library preloading mechanism available on standard Unix and Linux systems so that our library wraps the target library calls. This technique requires no modifications in the traced program, but needs to set the environment variable, `LD_PRELOAD`, to include our tracer library. The wrapper functions, upon being called by the target, generate trace entries before and after calling the real library functions.

6.3.2 Non-Blocking Concurrent Trace-Buffer Pool

To make a trace buffer accessible from both a traced process and trace readers, we inject a shared library into the process using the dynamic library preloading mechanism. The library, upon being loaded, allocates a shared memory region of specified size, and divides it into sub-buffers. Each sub-buffer is in either a *free* or *written* state, enqueued to either a free or written queue. The traced process records its traces into free buffers, while trace readers consume the traces from written buffers. Figure 6.5 illustrates an example where a buffer pool is used by a single traced process and three reader processes.

The trace library initially adds the sub-buffers to the free queue for write accesses from the target process. The target process, instrumented to generate function-call traces using the methods described above, obtains a free buffer from the free queue in the pool, and starts executing its functions, appending call traces to the buffer. Every time when the traced process finds that the current buffer becomes out of space, it obtains the next available free buffer from the pool. Simultaneously, multiple trace-reader processes can attach to the pool and obtain the written buffers for read access. After a reader process has finished reading

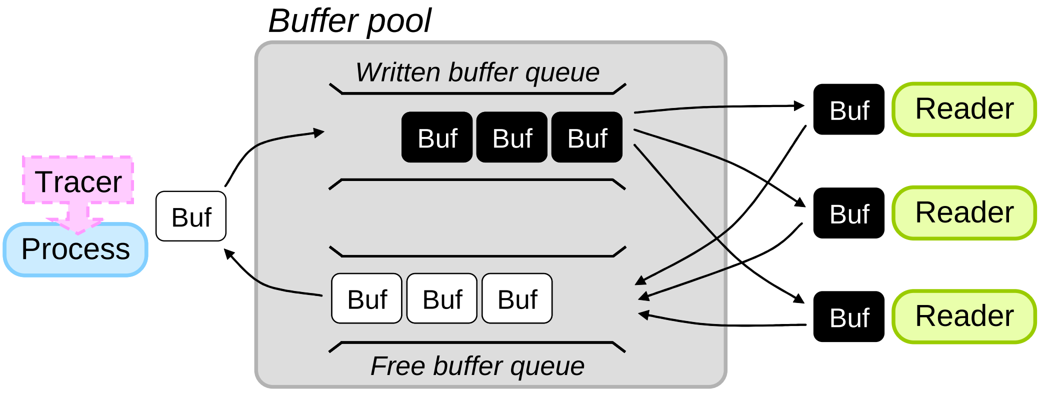


Figure 6.5: Example usage scenario of trace-buffer pool.

its buffer, it return the buffer to the pool, which then becomes available as a free buffer.

When a traced process attempts to obtain a free buffer, but no more such buffers are currently available, it gives up generating traces for a specified number of calls and returns, instead of blocking on a free buffer becoming available. It retries to obtain a free buffer after a specified count of calls and returns. This non-blocking scheme aims to minimize perturbation to the traced process, while sacrificing the completeness of the trace.

6.4 Evaluation

To evaluate the effectiveness of the proposed approach, we applied our prototype fault localizer to a known bug in a distributed job manager called MPD. MPD spawns a parallel job on a specified set of machines, monitors the job status, and returns the output of each process to the user. It is shipped with MPICH2, a standard implementation of the Message Passing Interface (MPI), and used by a wide variety of parallel programming users [35].

A user of the distributed computing platform called InTrigger reported a hang of MPD when he had tried to run a small MPI program. InTrigger is a large-scale computing platform consisting of six clusters distributed over Japanese universities and national laboratories. We applied our localization method to MPD running on the InTrigger platform, and successfully identified an anomalous event that was highly correlated with the bug. The rest of this section describes the experimental setup and the fault localization results of the bug.

6.4.1 Partial-Message Receive Bug in MPD

MPD manages all nodes by running daemon processes that are connected by a ring-topology network. When a new parallel job is submitted to the system, it spawns a specified number of processes by forwarding the job information over the ring network. The network is also used to coordinate the ready state of all processes to start execution.

A reported hang occurred when a user submitted a small MPI program using MPD version 1.0.5p4 running on multiple distributed clusters. Although the MPI program is a very small test program that runs flawlessly within a single cluster, using multiple clusters prevented the program from starting, and it apparently hanged up during its job-startup stage. Further examination by the user through printf-style message logging revealed that one of the MPD job manager daemons erroneously closed a connection in the ring-topology daemon network, causing all the daemons to infinitely wait for a particular message that should have been sent over the ring connection. The reason for the connection reset turned out that one of the calls to `recv` function in the socket API silently ignored its return value. The call expected to receive eight bytes every time when a message arrived at the connection, but in fact it sometimes received partial messages, which in turn caused the daemon to close the connection, because it erroneously determined that an error had occurred in the system. Of course, this assumption does not necessarily hold in distributed environments, yet, since MPI is mainly used in tightly connected single-site clusters, the bug had not been reported before.

This particular fault exhibits several common properties that make fault localization particularly difficult in such large-scale environments. First, it is nondeterministic: in some runs, the daemons always received completely formed messages, allowing jobs to be successfully started even on multiple clusters. In fact, the greater the number of nodes, the more often the fault occurred, requiring scalable localization techniques. Second, since it is a timing-related bug, use of an interactive debugger, if possible, would significantly reduce the chance of reproducing the bug, although debugging with as little overhead as printf-style message logging still allowed the bug to occur. Third, this is not a fail-stop, but a silent bug. Although the ring topology was not operating correctly after the bug occurred, each daemon was still in a normal state, waiting on the event-processing loop. While the examination of the call stack of a failed process would be an effective debugging technique in many cases, it would not help reveal the root cause in this case; the stack trace would look normal since waiting on the event loop is a legitimate operation.

Table 6.1: Process-model generation configurations and results.

Application	# nodes	Time (sec)	Trace Size (KB)	Model Size (KB)
CPI	58	0.95	949.2	43.8
CG Class C	4	127.75	448.8	43.4
IS Class A	8	3.29	483.7	43.7
CG Class C	16	58.31	566.9	43.5
MG Class B	32	2.16	682.9	43.7
LU Class D	32	1782.85	939.7	43.6
BT Class D	49	1223.16	1395.3	43.6

6.4.2 Model Derivation

We generated the global model from normal traces of MPD executions on a single cluster as follows. First, we traced MPD on seven different configurations using a simple MPI program called CPI and the NPB [94] as sample parallel jobs. CPI, shipped with the MPICH itself, calculates the value of π by using a Monte Carlo method. For the buffer pool on each node, we allocated ten sub-buffers of 1MB, or 10MB in total. We obtained per-process function-call traces using an online trace reader that saved the content of the written trace buffers to its local disk. Next, on each local node, we generated a process model for each trace using our prototype model generator. Next, we gathered the process models into a central repository and generated the global model. Below, we describe the detailed results for model derivation as well as the performance overhead caused by function tracing.

6.4.2.1 Process-Model Derivation

Table 6.1 lists the seven configurations and their results in process model generation. We executed traced MPD processes on different numbers of nodes, each of which hosted a single MPD process. The three columns for time, trace size, and model size list the averaged values for all the nodes. Because we implemented the model derivation as a Python program, and the derived process models were thus Python objects, we measured their sizes by serializing them to binary data. We can see that, while the execution times vary significantly, the sizes of the resulting process model are very similar, suggesting that the MPD processes exhibited repetitive function-call behaviors and our modeling efficiently encoded their behavior without substantially duplicating information.

To study the performance impact that MPD tracing had on the application programs, we compared their execution times with and without our tracing enabled.

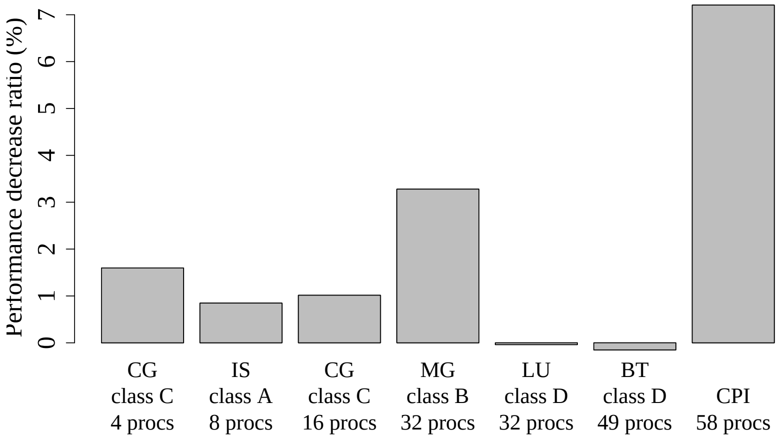


Figure 6.6: Performance overhead due to function-call tracing.

In Figure 6.6, the y-axis shows the relative performance when executed under our tracer compared to normal executions. The overhead ranges from approximately 0% to 7%. while such a small overhead does matter for extremely performance-critical systems, we expect that such small perturbation would be affordable in many cases, considering the effectiveness of our fault localization presented below.

6.4.2.2 Global-Model Derivation

We gathered all the 199 process models into a central location, and generated the global model including the overall behaviors of MPD processes. The resulting model was 171KB in size with only three different process groups. Although we aggregated such a large number of process models with different runs, this result again suggests that our model derivation was efficient in terms of disk space.

6.4.3 Localization through Suspect-Score Computation

We obtained a set of per-process trial traces by reproducing the bug on three geographically distributed clusters in InTrigger, called hongo, chiba, and okubo. We started traced MPD processes with the sample MPI program CPI on 78 nodes spanning the three distributed clusters, and determined that the system hung up after no output was observed for approximately 70 sec. We stopped the system by killing all the daemons. Out automated analysis decomposed the trial traces into execution units and computed the anomaly scores.

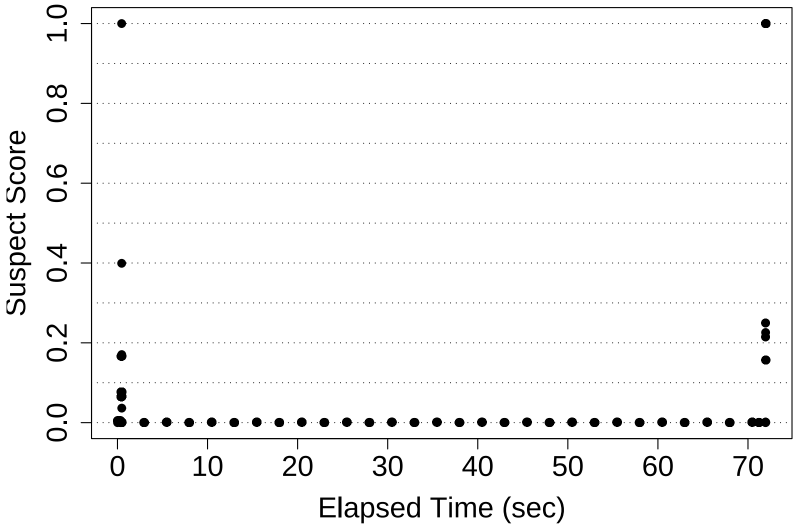


Figure 6.7: Suspect scores for execution units in trial traces.

Figure 6.7 shows the scores of all units plotted at their corresponding timestamps. We can see that the beginning and ending of the run had significantly deviant behaviors, while the rest did not exhibit any interesting behaviors. We can see that the most interesting unit is the one with the maximum score plotted near the beginning of the run, since the earlier a deviant behavior occurs, the more likely it would be correlated with a failure. We find that the particular unit is a handler unit that was notified of a connection-reset event. The reason for the high suspect score of the unit is that the reset event had never been observed in the normal traces. From the trace entries before the unit, we found that the other peer of the connection was running on node hongo102 in the hongo cluster, while the unit itself was on node chiba121 in the chiba cluster. Another high-score duration before the end of the run turned out to be caused by our abrupt killing of the daemons.

To identify why the connection was reset between chiba121 and hongo102, we manually examined the MPD source code and the trace entries in hongo102 around the time when the reset event occurred. We found that hongo102 had closed the connection in function `handle_lhs_input` of the `MPDMan` class, and that the function closed this because a call to `recv_dict_msg` of `MPDSock` failed. The trace entries for the calls from `recv_dict_msg` included a call to `recv` that actually received only three bytes of data from the connection, while `recv_dict_msg` expected eight bytes. As a result, because of the partial-message receive bug, the execution of `recv_dict_msg` failed, causing `handle_lhs_input` to close the connection.

This case study suggests the effectiveness of our support for fault localization. Although the buggy `recv` call was not able to be pinpointed, our suspect score ranking identified the connection reset event observed by the connection peer process as the most suspicious event. Without our automated trace analysis, the problem analyst would have needed to examine all the traces of the 78 nodes accounting for the complete 70-sec run. Our analysis narrowed the localization down to only the traces of two nodes accounting for less than a second, significantly reducing the manual burden imposed by determining the root cause of the failure.

6.5 Summary

We presented our model-based approach to fault localization that aims to help the human analyst narrow down the area of the manual localization process into a small fraction of the overall system. Our method consists of two parts: pre-failure model derivation and model-based anomaly detection. The first part collects function-call traces from all processes and derives an execution model that reflects the function-calling behaviors of the target system. When a failure occurs, we compute the suspect scores of all execution units in the failure traces. The suspect score, ranging from 0 to 1, quantifies how likely the execution unit is correlated with the failure. Our claim is that the analyst can substantially reduce the burden imposed by fault localization by prioritizing execution units with higher suspect scores. Our preliminary experiment with a distributed job manager supported this claim: Our method narrowed down the bug-finding process of a 70-sec faulty run on a 78-node distributed platform into just sub-second behaviors involving only two nodes.

There are several issues that remain to be resolved. First, we will explore online approaches to model derivation and suspect-score calculation. Our current method dumps function-call traces into local disks by running an online trace dumper in the background of the target process. This scheme was effective for the presented experiment with MPD; indeed, we observed that a 24-hour run of MPD with continuous execution of the NPB benchmark programs generated only 5.5GB of traces per node, which is far smaller then the current standard capacity of HDDs. Yet, retaining traces of long-running applications inevitably exceeds disk space capacity, no matter how large it is, at some time in the course of execution. To keep the space overhead at a minimum, we will explore online modeling and suspect-score calculation (see Section 7.3.1 for a more detailed discussion).

Second, we will study how long it takes to collect sufficient traces to learn a model with acceptable accuracy. In the case study, we collected the traces of seven runs, whose execution times ranged from approximately a second to 30 min. While these seven runs were sufficient for the particular failure, larger, more complex systems could need more normal trace data before their models were stabilized. We will study the speed of the model stabilization speed by varying the normal-data collection time. Third, we intend to apply our technique to localizing faults in different distributed systems. Specifically, for each different target system, we are planning to investigate 1) whether the assumed event-driven architecture holds or not, 2) how concise our modeling can encode its behaviors, and 3) how effective the model can be in localizing faults on the system. Different systems could employ different event-handling mechanisms, such as polling and signals. We need to extend our current modeling to support behaviors involving such event-handling mechanisms. Also, more-complicated control flow might be observed in different systems because of non-serial program executions using various mechanisms, such as multithreading, exceptions, and non-local jumps using the `setjmp` and `longjmp` functions. We intend to develop methods of capturing and modeling of programs involving such control flow as well.

Chapter 7

Discussion

For the problem stated in the beginning of this dissertation, i.e., difficulties in localizing faults in large-scale computing systems (see Section 1.1), this dissertation presented an automated approach to with two different techniques, each of which exploits similarities across processes or time. We designed both techniques to reduce the burden on fault-localization processes in large-scale computing systems, particularly HPC clusters and Grids. We demonstrated the effectiveness of both techniques in localizing faults in these environments through several case studies.

The two main reasons for these difficulties are the scale of modern computing systems and rare nondeterministic faults. The model-based technique particularly focuses on the first reason; by deriving an execution model reflecting locally observable normal system behaviors, it detects anomalous behaviors from fault traces with no remote communications required. While it does not learn distributed behaviors across multiple nodes, our claim is that many faults would also exhibit locally observable anomalies; thus, detecting these anomalies might not be able to pin-point the exact root cause of a fault, yet they could narrow down further fault localization to a smaller part of the entire system. Our case study with the MPD partial-message receive bug supports this claim; although our automated analysis was not able to find the exact location of the bug, the analysis results did detect locally observable anomalies that were effective in finding the root cause.

Our always-on tracing approach aims at solving the second reason for the difficulties in localizing faults—rare nondeterministic faults. Function-call tracing allows us to examine the control flow of a target process even when such faults occur. While the performance overhead due to tracing calls and returns could be too large for some performance-critical applications to afford, we believe that a decrease in performance by a few percent, as was the case with the experimental case studies, would be sufficiently small for a wider class of applications.

The trace-based approach, however, does have limitations due to possible trace loss. The tracing technique presented in Chapter 5 used a fixed-length wraparound trace buffer where old traces were overwritten by new ones. Thus, while it kept the trace size constant, the behaviors that could be identified from the trace were limited by the buffer size. Since having a sufficiently large buffer for arbitrary long-running systems is impractical due to its disk space cost, it cannot help but lose information on earlier system behaviors. Thus, if a fault is a slow-developing problem that makes itself visible as a failure far later than the initial occurrence, we may miss the original anomalous event of the problem due to buffer overwrite. Thus, once the system starts misbehaving, the key to successful localization depends on how early the buffer content can be dumped to a file. This problem is less critical for fail-stop faults, such as the network-stability problem in SCore, since no new traces are added to the buffer, preserving the precise behavior of the process when the fault occurs; however, this does matter for non-fail-stop problems, such as the scbcast problem. In the scbcast case, we relied on the native timeout detector, `sc_watch`, which was shipped with the original SCore system, to detect failures and initiate trace dumping.

In the model-based technique, we introduced a concurrent trace buffer to save traces while they were being collected from the target system. This scheme alleviates the problem by exploiting local HDDs, whose data capacity should be much larger than the main memory. As mentioned in Section 6.5, the trace size for a 24-hour run of MPDs was approximately 5.5 GB, while the size of a single platter of today's HDDs can be as large as 334 GB, expected to hit 500 GB in the near future [5]. Thus, keeping traces for several days of a run would be feasible, allowing those faults that occur within such a time frame to be diagnosable with our technique.

A more fundamental approach to trace-size growth is to process the data on-the-fly and to only retain information of interest, instead of simply dumping all traces to disks. Our model-based technique would be particularly suitable for such an online scheme because of its decentralized localization and concurrent trace buffers. We will further discuss an online approach in Section 7.3.1.

The rest of this chapter discusses the similarities and differences between the two proposed techniques of localization and their limitations when applied to a broader class of faults. Finally, we present a possible approach to addressing the current limitations.

7.1 Comparison of Outlier and Model-Based Techniques

The outlier-detection and model-based techniques are two instantiations of detecting anomalies exploiting similarities observed in distributed computing systems.

The advantages of the former over the latter include: 1) no *a priori* processing, such as the model training, and 2) the ability to detect performance anomalies through time profiles. Unlike the former, the latter derives an execution model only taking function-call coverage into account; since a performance bug that manifests itself as an anomalously longer execution time of a particular operation would not change its function-call coverage, the latter could not detect such behaviors as anomalies.

In contrast, the advantages of the latter over the former include: 1) more scalable localization through the completely decentralized algorithm and 2) no assumptions regarding spatial similarities. Outlier-based analysis requires pair-wise comparison of all per-process traces, which would be even more problematic as system size grows. While the latter also requires gathering per-process data (i.e., process models) to a central repository, this is only required for the pre-fault model derivation phase; localization of faults does not require such global operations at all. Furthermore, a model derived in a smaller-scale system can be applied to localize faults in larger-scale systems; the MPD case study presented in Section 6.4 actually exploited this advantage: we derived a model using a single cluster, and localized the bug that was only observed on multiple-cluster environments.

The second advantage—no assumption regarding spatial similarities—simplifies the false-positive problem that the unsupervised ranking algorithm suffered when localizing the scbcast problem (see Section 5.3.4). Although the one-class algorithm alleviates the problem using previously collected traces, its effectiveness is limited by how thoroughly the previous traces cover anomalous behaviors, which is likely to require a very large collection of traces. Unlike the compact model derivation in the latter technique, we did not explore this problem in the study; indeed, this is one of the problems that motivated the model-based technique.

While both techniques have advantages and disadvantages, we could devise yet another technique that exploits both of them. That is to say, to find anomalous behaviors, we could compare fault traces with both those of other processes and their execution models. For example, assume that a majority of distributed processes exhibited an unknown but legitimate behavior. The model-based localization would find the behavior in a decentralized way but would determine it as anomalous since it was unknown. However, we could use the outlier-detection-based localization to further improve the accuracy of localization; we could gather only the traces for the behavior to a central location, and further localize the problem by the outlier-based technique. In this case, second-phase localization could determine the behavior as normal since it was observed in a majority of processes. As such, by the combination of the two techniques, we expect that we could further reduce the number of false positives. Further studies toward this direction remain a subject for future research.

7.2 Limitations

7.2.1 Limitations Derived from Assumptions

Our localization approach assumes that the target system mostly works as intended, but suffers from rare occurrences of faults. More specifically, in the outlier-detection based technique, we assume that the majority of involved processes operates properly, but a small number exhibit anomalous behaviors. Instead, if the majority exhibited anomalous behavior, we would end up giving "normal" annotations to their traces since they were not outliers. Although our technique could not localize the cause of problems with such symptoms, it could still help the problem to be understood to some extent. If no outliers were found, all the processes should have exhibited a small number of behavior patters. Such patterns could be identified by using data clustering methods such as hierarchical cluster analysis [40] as done by Dickinson et al. [22]. The problem analysis would only need to examine a small number of traces for each behavior pattern.

The model-based technique assumes that each process operates properly most of the time so that we can derive the execution model reflecting the normal behaviors of the system. Therefore, it is not particularly effective in early phases of software development where prototypes still fail most of the time. Yet, it could be useful to diagnose common situations with software problems that occur due to differences in underlying environments, as in the localization of the MPD partial-message receive bug.

Another assumption that can limit the application of our proposal is the manifestation of faults as control-flow changes. If no changes occur in function-call traces, but only exhibit more subtle misbehaviors, such as silent data corruption, we will not be able to detect anomalies. Although this assumption has been valid in our case studies, more thorough evaluations with a wider variety of systems are necessary for assessing how applicable this assumption is.

7.2.2 Fault Localization in Broader Problem Domains

This dissertation addressed localization of a particular class of faults, i.e., anomalies in distributed middleware-type software. While that is an important class of problems, anomalies in other domains, such as hardware, more performance-oriented scientific programs, and operating system and kernel code, can be sources of reliability and dependability issues as well. Although we believe that the general idea underlying anomaly-detection-based localization could be effective in other domains of faults, we would face different challenges in particular fault domains. For example, incorrect program results could be caused by not only logic software bugs, but also extremely high CPU temperature through bit-flip faults, requiring monitoring of various system properties. Excessively slow responses from a remote peer could be caused by various misbehaviors, such as intermediate overloaded routers, excessive CPU load of the peer, and misconfigured routing tables. How system behaviors should be modeled to localizing such diverse faults is still unclear and remains a subject of active research [61, 82].

The recent trend towards virtualization can further complicate problem diagnosis due to the added abstraction layer between bare metal hardware and operating systems. For example, virtual-machine-based cluster-computing environments, i.e., virtual clusters [65, 86], are a promising approach to future HPC computing platforms on Grids, since they could achieve more efficient, flexible use of distributed computing resources. However, problems occurring in such environments could be caused by virtualization layers as well as original system components, or even worse, mixtures of these. Localizing faults in such broader domains also requires further research.

7.3 Possible Approaches to Overcoming the Limitations

7.3.1 Online Approach to Modeling and Anomaly Detection

As previously discussed, one possible approach to fault localization in long-running systems is online system modeling and proactive, autonomous detection of anomalies. By processing raw traces on-the-fly and only retaining derived models for later localization of faults, the trace-size-growth problem inevitable in any post-mortem analysis techniques, including ours, could be avoided. Furthermore, by proactively finding anomalous behaviors, non-fail-stop system faults could be detected and analyzed without manual assistance; our current techniques require initial notification of failures from other sources (i.e., the default timeout detector in the SCore case or manual termination of the non-responsive system in the MPD case). Such proactive detection of faults would play an important role in autonomic system management [41].

Of our two techniques, the model-based one would be more suitable for such online schemes than the outlier-detection-based one thanks to its decentralized nature of fault localization. A model-based technique adapted for online processing would derive models while simultaneously finding anomalies in a running process. We could flag as an anomaly an execution unit with a suspect score larger than a given threshold. It would initially generate many false alarms due to lack of learned behaviors; however, over time, it would eventually observe most normal behaviors, and then generate fewer and fewer false alarms. How long it would take to derive false-alarm-free models would depend on the complexity of the target program and the sensitivity of fault detection defined by the anomaly threshold. As a program becomes larger, it tends to exhibit more types of behaviors, requiring a great deal of training data. Also, the lower the suspect-score threshold, the more alarms it would generate, whether true or false, resulting in a smaller precision ratio. The higher the threshold, on the other hand, the more likely it would miss true anomalies, resulting in a smaller recall ratio. Finding the right balance of recall and precision is a common problem in any kind of information retrieval from a large volume of data, and generally depends on the particular system being observed.

Implementation issues in accomplishing such an online approach include the computation and memory cost in analyzing continuously generated data. This should be kept as low as possible to minimize perturbation to the system, while still presenting rich diagnostic information when it misbehaves. Given the recent trends toward higher parallelism with multi-core CPUs and Chip Multithreading Technology (CMT) [44, 62], however, we can envision that a helper thread running on an idle core could monitor and learn the behaviors of other threads for diagnosing problems. The availability of idle resources generally depends on the parallelism of applications running on the CPU; however, we expect that it would be acceptable in many situations to allocate part of the resources for such purposes, particularly because of the usefulness of being able to localize faults quickly and automatically.

Chapter 8

Conclusions

This dissertation discussed our studies on fault localization in large-scale computing systems. This chapter concludes the dissertation with a summary of contributions and a discussion on directions for future research.

8.1 Summary of Contributions

This dissertation presented two trace-based fault-localization approaches for large-scale computing systems. Both approaches assume similarities in process execution behaviors and localize faults by finding anomalous behaviors that violate these similarities.

The first approach finds outliers in a distributed collection of identical processes by assuming spatial similarities, i.e., that normal processes should exhibit similar behaviors to one another. Specifically, it first performs process-level localization to find faulty processes in the target system, for which we have presented two techniques, the first for fail-stop faults and the second for non-fail-stop faults. For fail-stop faults, we find an anomalous process by finding the process with the earliest timestamp, since fail-stop faults are often caused by a misbehaving process causing the other processes to stop functioning. For non-fail-stop faults, we find outliers by examining behavioral differences in more detail. That is, we first compute the time profile from each per-process trace that quantifies how long the process spent in each function, and then compute the suspect score that reflects how its time profile is different from the rest of the processes. Finally, we find outliers by locating the processes with the largest scores. Having found suspicious processes, we locate the most suspicious functions by finding either the function of the last trace entry or the functions that made the largest contributions to the suspect score.

This approach is especially effective for such faults as performance bugs, livelocks, and deadlocks. In other words, any misbehavior that affects execution times abnormally would be detectable since it should manifest itself as an outlier.

We evaluated the proposed technique by localizing faults in a production-cluster environment consisting of 129 nodes. The cluster used the SCore distributed-computing middleware for managing parallel-job executions. The cluster had experienced occasional crashes and hangups in SCore without the reasons being evident. We collected the function-call traces of the SCore processes and analyzed their fault traces with our automated two-step localization. The results successfully identified the causes of fail-stop faults and non-fail-stop ones with little manual assistance, suggesting the effectiveness of the proposed technique.

The second approach finds anomalous behaviors by assuming historical similarities, i.e., that each process exhibits repetitive, self-similar, behaviors. The first phase in the process derives a concise execution model reflecting the normal behaviors of the target system. To do so, we collect per-process function-call traces while the target operated as intended, and construct a process model for each process, which describes how the process behaved on receiving network messages. Specifically, we automatically identify which functions were called when a message arrived at each network connection, and construct a call tree for each connection. We annotate each tree node with an estimated call probability by counting the number of occurrences in the collected traces. The final output of the model-derivation phase is a global model that aggregates all the process models by merging them for the processes with the same estimated role. We consider two processes to have performed the same role if they operated with the same set of network connections. To localize a fault, we deploy the derived global model in all participating processes, and collect function-call traces as in the model-derivation phase. We next compute how differently each process responded to incoming messages by comparing the collected traces with the derived model. We consider a function suspicious in the two cases when: 1) its estimated probability was low, yet it was called in a failure run, and 2) its probability was high, yet it was not called. Finally, as in the outlier-detection based technique, we compute a suspect score for all call trees to quantify its likelihood being the root cause of the failure, and present the scores to assist the problem analyst in diagnosing the fault.

This approach is especially effective in localizing program logic bugs For example, assume that an application performs an operation when receiving user requests. Typical request handling involves executing certain functions depending the types of the requests: every time a user request arrives at the application, some functions for the particular request type should be called. However, if the application failed to correctly serve a request, some of the functions for the request type might not be called. The model-based localization could detect such behaviors as anomalies; the problem analyst would be able to determine the source of the failure automatically.

To evaluate the effectiveness of the proposed technique, we applied our prototype model-based localizer to a known bug in a distributed job manager called MPD. The job manager operated as intended on a single-cluster environment, but hanged up nondeterministically if executed on geographically distributed multiple clusters. We collected its normal by running it on a single cluster and fault traces by running it on a distributed environment with three clusters. We derived its execution model from the normal traces and found an anomalous execution unit of a single MPD process in the fault traces by computing their suspect scores. Further manual examination with source code revealed that the unit erroneously closed a connection with other processes, causing the entire system to block infinitely. These results demonstrate how effective the model-based technique is in localizing faults in large-scale computing systems.

8.2 Directions for Future

Our work is by no means complete. Our ultimate goal is to accomplish fully automated fault localization for a wider variety of systems and, moreover, autonomic fault recovery for certain classes of problems. Specifically, we intend to continue our research in the following directions.

- *Autonomic fault detection.* Online system modeling and anomaly detection as discussed in Section 7.3.1 would allow us to detect faults in autonomically. Our ultimate goal in this direction is to eliminate human interactions as much as possible, while keeping false positive and negative ratios low.

- *Autonomic fault recovery.* Once a fault is detected in a system, simple fault recovery would terminate and restart it from the very beginning. Another technique could use checkpoint and restart techniques for retrying faulty operations. Specifically, our model-based localization would identify when the system starts misbehaving; thus, by restarting the system from a checkpoint saved before the identified fault occurred, it could be re-executed from a normal state and the anomalous operation reattempted. Although this scheme could not avoid deterministic faults, it might be effective for nondeterministic faults, which do not necessarily occur.

- *Whole-system fault localization.* Thus far, our study has focused on localizing faults in user-level distributed software; we have not yet explored faults in operating systems or in interactions of operating systems and user-level components. Another direction would be to construct a model describing whole-system behaviors and to capture similarities across a variety of such components. To achieve that end, system virtualization, such as Xen by

Barham et al. [7], as well as kernel-level system instrumentation, such as Kerninst by Tamches and Miller [84], might come into play. For example, a hypervisor could monitor the behaviors of a virtual machine by instrumenting the guest OS as well as the applications running atop it. The real challenge would be how to correlate various events across multiple software layers on a single machine as well as among multiple distributed machines.

In conclusion, we stress that even in large-scale computing systems automated fault localization is indeed effective, although it might not completely eliminate manual effort. Our sincere hope is that such automated localization will emerge to play a more important role toward achieving autonomic computing systems.

Bibliography

[1] Advanced Micro Devices. AMD64 Architecture Programmer's Manual Volume 2: System Programming, September 2007.

[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 74–89, 2003.

[3] Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC'02)*, pages 1–16, Los Alamitos, CA, 2002.

[4] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory Lee, Barton P. Miller, and Martin Schulz. Stack Trace Analysis for Debugging Large Scale Applications. In *International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–10, 2007.

[5] Nikkei Electronics Asia. 3.5-Inch HDD Capacity to Hit 500GB/Platter in 2008. Available at `http://techon.nikkeibp.co.jp/article/HONSHI/20070926/139720/`, Oct 2007.

[6] A. Avizienis, J.-C. Laprie, and B. Randell. Fundamental Concepts of Dependability. Technical report, Department of Computing Science, University of Newcastle, 2001.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP'03)*, pages 164–177, 2003.

[8] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *6th Symposium on*

Operating Systems Design and Implementation (OSDI'04), pages 259–272, 2004.

[9] Stephen D. Bay and Mark Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 29–38, 2003.

[10] Daniel Becker, Felix Wolf, Wolfgang Frings, Markus Geimer, Brian J. N. Wylie, and Bernd Mohr. Automatic Trace-Based Performance Analysis of Metacomputing Applications. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–10, Long Beach, CA, March 2007.

[11] Bryan R. Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[12] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[13] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[14] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *Grid'2005 Workshop*, Seattle, USA, November 13-14 2005. IEEE/ACM.

[15] Anthony Chan, David Ashton, Rusty Lusk, and William Gropp. Jumpshot-4 Users Guide. `http://www.mcs.anl.gov/perfvis/software/viewers/jumpshot-4/usersguide.html`, 2008.

[16] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 595–604, June 2002.

[17] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-Based Failure and Evolution Management. In *Proceedings of the 1st USENIX/ACM Symposium on Networked*

Systems Design and Implementation (NSDI '04), San Francisco, CA, March 2004.

[18] Cluster File Systems, Inc. Lustre file system. `http://www.clusterfs.com`, 2008.

[19] Cluster Resources, Inc. Torque resource manager. `http://www.clusterresources.com/pages/products/torque-resource-manager.php`, 2008.

[20] Ira Cohen, Moises Goldszmidt, Terence Kelly, Symons Julie, and Chase Jeff. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Operating System Design and Implementation (OSDI'04)*, pages 231–244, Dec 2004.

[21] Charng da Lu and Daniel A. Reed. Assessing Fault Sensitivity in MPI Applications. In *ACM/IEEE SC 2004 Conference (SC'04)*, 2004.

[22] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 339–348, Toronto, Ontario, Canada, May 2001.

[23] Catalin L. Dumitrescu, Ioan Raicu, and Ian Foster. Experiences in Running Workloads over Grid3. In Hai Zhuge and Geoffrey Fox, editors, *4th International Conference on Grid and Cooperative Computing (GCC'05)*, number 3795 in Lecture Notes in Computer Science, pages 274–286. Springer, 2005.

[24] Toshio Endo and Satoshi Matsuoka. Massive supercomputing coping with heterogeneity of modern accelerators. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, USA, April 2008. To appear.

[25] NR Adiga et al. An Overview of the Blue Gene/L Supercomputer. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC'02)*, pages 1–22, Nov 2002.

[26] *Using the GNU Compiler Collection*. Free Software Foundation, Inc, 2007. `http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/`.

[27] *Debugging with GDB*. Free Software Foundation, Inc, 2007. `http://sourceware.org/gdb/current/onlinedocs/gdb.html`.

[28] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3):131–163, 1997.

[29] Fabrizio Gagliardi, Bob Jones, François Grey, Marc-Elian Bégin, and Matti Heikkurinen. Building an infrastructure for scientific Grid computing: status and goals of the EGEE project. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 363(1833):1729–1742, August 2005.

[30] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 289–300, 2006.

[31] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation*, pages 285–298, 2007.

[32] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable Parallel Trace-Based Performance Analysis. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI User' s Group Meeting*, volume 4192, pages 303–312, 2006.

[33] Jim Gray. Why Do Computers Stop and What Can We Do About It. In *6th International Conference on Reliability and Distributed Databases*, June 1987.

[34] Steven D. Gribble. Robustness in Complex Systems. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 21–26, 2001.

[35] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sep 1996.

[36] Kevin A. Huck and Allen D. Malony. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In *Proceedings of Supercomputing 2005*, Nov 2005.

[37] Intel Corporation. Intel 64 and IA-32 Arhictectures Software Devleoper's Manual Volume 1: Basic Architecture, August 2007.

[38] Yutaka Ishikawa, Atsushi Hori, Hiroshi Tezuka, Motohiko Matsuda, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, Jorg Nolte, and Mitsuhisa Sato. MPC++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, pages 427–466. The MIT Press, 1996.

[39] Yutaka Ishikawa, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Toshiyuki Takahashi, Francis O'Carroll, and Hiroshi Harada. RWC PC Cluster II and SCore Cluster System Software—High Performance Linux Cluster. In *Proceedings of the 5th Annual Linux Expo*, pages 55–62, Raleigh, NC, May 1999.

[40] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[41] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan 2003.

[42] Emre Kiciman and Armando Fox. Detecting application-level failures in component-based Internet services. *IEEE Transactions on Neural Networks*, 16:1027–1041, Sept 2005.

[43] George Kola, Tevfik Kosar, and Miron Livny. Faults in Large Distributed Systems and What We Can Do about Them. In *Proceedings of 11th European Conference on Parallel Processing (Euro-Par 2005)*, Lisbon, Portugal, August 2005.

[44] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: a 32-way multithreaded Sparac processor. *IEEE Micro*, 25(2):21–29, 2005.

[45] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic Replay of Distributed Java Applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS'00)*, page 219, 2000.

[46] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra Sahoo. BlueGene/L Failure Analysis and Prediction Models. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 425–434, June 2006.

[47] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03)*, pages 141–154, 2003.

[48] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 15–26, Chicago, IL, USA, 2005.

[49] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[50] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. Wids checker: Combating bugs in distributed systems. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07)*, pages 257–270, April 2007.

[51] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200, Chicago, IL, USA, 2005.

[52] Naoya Maruyama and Satoshi Matsuoka. Model-based fault localization in large-scale computing systems. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, USA, April 2008. To appear.

[53] Raissa Medeiros, Walfredo Cirne, Francisco Brasileiro, and Jacques Sauve. Faults in Grids: Why are they so bad and What can be done about it? In *Fourth International Workshop on Grid Computing*, page 18, 2003.

[54] Microsoft Corporation. Parallel Debugging Using Visual Studio 2005. `http://www.microsoft.com/downloads/details.aspx?FamilyID=798b01f3-2bad-41c7-a34a-c2a6e6f9d535`, Nov 2005.

[55] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[56] David L. Millis. The network computer as precision timekeeper. In *Precision Time and Time Interval (PTTI) Applications and Planning Meeting*, pages 96–108, Reston, VA, Dec 1996.

[57] Nelson Minar. A survey of the ntp network, December 1999.

[58] Alexandar V. Mirgorodskiy, Naoya Maruyama, and Barton P. Miller. Problem Diagnosis in Large-Scale Computing Environments. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'06)*, Tampa, Florida, November 2006.

[59] Alexander Mirgorodskiy and Barton Miller. Autonomous analysis of interactive systems with self-propelled instrumentation. In *MMCN 2005: 12th Multimedia Computing and Networking*, San Jose, CA, January 2005.

[60] Alexander V. Mirgorodskiy. *Automated Problem Diagnosis in Distributed Systems*. PhD thesis, University of Wisconsin-Madison, 2006.

[61] Jeffrey C. Mogul. Emergent (mis)behavior vs. complex software systems. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (EuroSys'06)*, pages 293–304, Leuven, Belgium, 2006.

[62] Robert Mullins. Introducing the next hot multicore chip design. available at `http://www.pcworld.com/article/id,136214-pg,1/article.html`, Aug 2007.

[63] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'07)*, pages 89–100, San Diego, California, USA, 2007.

[64] O. Y. Nickolayev, P. C. Roth, and D. A. Reed. Real-Time Statistical Clustering for Event Trace Reduction. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):144–159, 1997.

[65] Hideo Nishimura, Naoya Maruyama, and Satoshi Matsuoka. Virtual Clusters on the Fly — Fast, Scalable, and Flexible Installation. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, Rio de Janeiro, Brazil, 2007.

[66] Adam Oliner and Jon Stearley. What Supercomputers Say: A Study of Five System Logs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'07)*, pages 575–584, Edinburgh, UK, 2007.

[67] DAS-3 project page. The distributed asci supercomputer 3 (DAS-3). `http://www.cs.vu.nl/das3/`, 2007.

[68] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data (SIGMOD'00)*, pages 427–438, Dallas, Texas, 2000.

[69] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 221–230, Toronto, Ontario, Canada, 2001.

[70] Manos Renieris and Steven P. Reiss. Fault Localization With Nearest Neighbor Queries. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 30–39, Oct 2003.

[71] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 115–128, San Jose, CA, May 2006.

[72] Philip C. Roth and Barton P. Miller. On-line Automated Performance Diagnosis on Thousands of Processes. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, pages 69–80, New York, NY, USA, March 2006.

[73] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of ACM/IEEE Conference Supercomputing (SC'03)*, page 21, 2003.

[74] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. Benchmarking the MRNet distributed tool infrastructure: lessons learned. In *Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, page 272, Santa Fe, New Mexico, April 2004.

[75] Hideo Saito, Yoshikazu Kamoshida, Shogo Sawai, Ken Hironaka, Kei Takahashi, Takeshi Sekiya, Nan Dun, Takeshi Shibata, Daisaku Yokoyama, and Kenjiro Taura. InTrigger: A Multi-Site Distributed Computing Environment Supporting Flexible Configuration Changes. In *IPSJ Sig Technical Report HPC-111*, pages 237–242, Aug 2007. In Japanese.

[76] Yasushi Saito. Jockey: user-space library for record/replay debugging. In *Sixth International symposium on automated and analysis-driven debugging (AADEBUG 2005)*, pages 69–76, Monterey, CA, Sep 2005.

[77] Bianca Schroeder and Garth A. Gibson. A Large-Scale Study of Failures in High-Performance-Computing Systems. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 249–258, June 2006.

[78] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. Data Mining Methods for Detection of New Malicious Executables. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[79] S. Shende and A. D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2): 287–331, 2006.

[80] Satoru Shingu, Hiroshi Takahara, Hiromitsu Fuchigami, Masayuki Yamada, Yoshinori Tsuda, Wataru Ohfuchi, Yuji Sasaki, Kazuo Kobayashi, Takashi Hagiwara, Shin ichi Habata, Mitsuo Yokokawa, Hiroyuki Itoh, and Kiyoshi Otsuka. A 26.58 tflops global atmospheric simulation with the spectral transform method on the earth simulator. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing (SC'02)*, pages 1–19, Baltimore, Maryland, 2002.

[81] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pages 29–44, 2004.

[82] M. Steinder and A.S. Sethi. A Survey of Fault Localization Techniques in Computer Networks. *Science of Computer Programming*, 53(2):165–194, November 2004.

[83] W. Richard Stevens. *UNIX Network Programming: Neworking APIs: Sockets and XTI*, volume 1. Prentice Hall, 2nd edition, January 1998.

[84] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 117–130, New Orleans, Louisiana, Feb 1999.

[85] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

[86] Masaki Tatezono, Naoya Maruyama, and Satoshi Matsuoka. Making Wide-Area, Multi-Site MPI Feasible Using Xen VM. In *Workshop on XEN in HPC Cluster and Grid Computing Environments (XHPC'06)*, Sorrento, Italy, Dec 2006.

[87] David M.J. Tax. *One-class classification; Concept-learning in the absence of counter-examples*. PhD thesis, Delft University, June 2001.

[88] Rajeev Thaku and William D. Gropp. Improving the Performance of Collective Operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (10th European PVM/MPI User's Group*

Meeting), volume 2840/2003 of *Lecture Notes in Computer Science*, pages 257–267, Venice, Italy, September 2003. Springer.

[89] Tokyo Stock Exchange, Inc. Measures Taken for Preventing Recurrence, etc. Concerning the Derivatives Trading System Malfunction. Available at `http://www.tse.or.jp/english/news/200802/080222_a.html`, February 2008.

[90] TotalView Technologies, LLC. TotalView Debugger Users Guide. `http://www.totalviewtech.com/`, 2007.

[91] Chad Verbowski, Emre Kcman, Arunvijay Kumar, Brad Daniels, Shan Lu, , Juhan Lee, Yi-Min Wang, and Roussi Roussev. Flight Data Recorder: Monitoring Persistent-State Interactions to Improve Systems Management. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[92] Jeffrey Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 245–254, Santa Fe, New Mexico, USA, 2000. ACM Press.

[93] David Wagner and Drew Dean. Intrusion detection via static analysis. In R. Dean, editor, *Proceedings of IEEE Symposium on Security and Privacy*, pages 156–168, 2001.

[94] Parkson Wong and Rob F. Van der Wijngaart. Nas parallel benchmarks version 2.4. Technical Report NAS-02-007, NASA Ames Research Center, 2002.

[95] Lingyun Yang, Chuang Liu, Jennifer M. Schopf, and Ian Foster. Anomaly detection and diagnosis in grid environments. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC'07)*, Reno, NV, November 2007.

[96] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the 2006 EuroSys conference*, pages 375–388, 2006.

[97] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, page 32, 1999.

[98] Ziming Zheng, Yawai Li, and Stephane Zhiling Lanteri. Anomaly Localization in Large-Scale Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'07)*, pages 322–330, Austin, Texas, Sep 2007.