AUTOMATED PROBLEM DIAGNOSIS IN DISTRIBUTED SYSTEMS

by

Alexander Vladimirovich Mirgorodskiy

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin–Madison

2006

AUTOMATED PROBLEM DIAGNOSIS IN DISTRIBUTED SYSTEMS

Alexander V. Mirgorodskiy

Under the supervision of Professor Barton P. Miller

At the University of Wisconsin–Madison

Quickly finding the cause of software bugs and performance problems in production environments is a crucial task that currently requires substantial effort of skilled analysts. Our thesis is that many problems can be accurately located with automated techniques that work on unmodified systems and use little application knowledge for diagnosis. This dissertation identifies main obstacles for such diagnosis and presents a three-step approach for addressing them.

First, we introduce *self-propelled instrumentation*, an execution monitoring approach that can be rapidly deployed on demand in a distributed system. We dynamically inject a fragment of code referred to as the *agent* into one of the application processes. The agent starts propagating through the system, inserting instrumentation ahead of the control flow in the traced process and across process and kernel boundaries. As a result, it obtains the distributed control flow trace of the execution.

Second, we introduce a *flow-separation algorithm*, an approach for identifying concurrent activities in the system. In parallel and distributed environments, applications often perform multiple concurrent activities, possibly on behalf of different users. Traces collected by our agent would interleave events from such activities thus complicating manual examination and automated analysis. Our flow-separation algorithm is able to distinguish events from different activities using little user help.

Finally, we introduce an automated *root cause analysis* approach. We focus on identification of anomalies rather than massive failures, as anomalies are often harder to investigate. Our techniques help the analyst to locate an anomalous flow (e.g., an abnormal request in an e-commerce system or a node of a parallel application) and to identify a function call in that flow that is a likely cause of the anomaly.

We evaluated these three steps on a variety of sequential and distributed applications. Our tracing approach proved effective for low-overhead on-demand data collection across the process and kernel boundaries. Manual trace examination enabled us to locate the causes of two performance problems in a multimedia and a GUI application, and a crash in the Linux kernel. Our automated analyses enabled us to find the causes of four problems in the SCore and Condor batch scheduling systems.

Acknowledgements

I would never have completed this work without many people who helped me at different stages of the process. First, I thank my adviser, Bart Miller, for being a perfect mentor, teaching me how to develop, evaluate, express, and defend my ideas. These skills are among my most valuable experiences from the graduate school. Furthermore, Bart's wise advice and guidance proved immensely helpful on thousands of occasions, during research and life in general.

I thank other members of my committee (Somesh Jha, Ben Liblit, Miron Livny, and Paul Wilson) for their time and help. Somesh Jha was always willing to offer useful feedback and suggestions on root cause analysis techniques. Ben Liblit taught an excellent seminar on software quality that I really enjoyed. Ben also provided especially detailed and useful comments on this document. Miron Livny was a source of excellent questions and suggestions that helped me refine and improve this work. Paul Wilson offered thoughtful comments and questions during the defense.

I thank Vic Zandy whose suggestion to look into diagnosing sporadic performance problems in everyday systems led me to the idea of self-propelled instrumentation. His feedback also was a great help at the early stages of the process.

I was fortunate to collaborate with Naoya Maruyama on automated problem diagnosis approaches. The foundation of our anomaly detection and root cause analysis techniques was laid via numerous discussions with Naoya and Bart. Naoya also actively participated in developing the trace processing and analysis tools, and in the presentation of those results. Finally, Naoya made the experiments with SCore possible.

Xiaojin (Jerry) Zhu offered his insight into machine learning approaches. Discussions with him enabled us to put our anomaly detection algorithms into the context of previous work on machine learning.

Remzi Arpaci-Dusseau was a member of my preliminary examination committee, provided invaluable feedback throughout the process, and offered career guidance.

Past and present members of the Paradyn project provided much needed support, critique of ideas, and served as the source of amazing technical knowledge. The work of Ari Tamches on kernel instrumentation was the model of exciting technical work that stimulated my interest in the instrumentation technology. The Dyninst expertise of Laune Harris was invaluable in debugging my binary parsing tool, a building block for self-propelled instrumentation. Eli Collins offered thoughtful comments on papers and ideas and provided encouragement throughout the process. Dorian Arnold was the excellent source of information on large-scale HPC systems and tools. Drew Bernat and Matt Legendre were always willing to help and provided useful feedback on presented ideas.

Jaime Frey and Jim Kupsch were a great help in setting up experiments with the Condor system and answering my numerous questions. They also offered their comments on the prose.

My friends, Denis Gopan and Julia Velikina, Sasho Donkov, Taras Vovkivsky and Tamara Tsurkan, Alexey Loginov, Laune Harris, and Eli Collins made my stay in Madison pleasant and enjoyable. Their support was always there when I needed it.

My parents, Elena and Vladimir Mirgorodskiy have been the immense source of love and wise advice. This journey has probably been more difficult for them than it has been for me, yet they were a constant source of encouragement.

My wife Nata has been a great companion for all these years. She was willing to listen and offer excellent judgement on many topics in life, including tricky computer science concepts. Above all, her love and emotional support were the key factors that enabled me to complete this work.

Contents

List of Figures

Chapter 1

Introduction

Quickly finding the cause of software bugs and performance problems in production environments is a crucial capability. Any downtime of a modern system often translates into lost productivity for the users and lost revenue for service providers, costing the U.S. economy an estimated $59.5 billion annually [103]. Despite its importance, the task of problem diagnosis is still poorly automated, requiring substantial time and effort of highly-skilled analysts. Our thesis is that problem diagnosis in production environments can be substantially simplified with automated techniques that work on unmodified systems and use limited application-specific knowledge. This dissertation presents a framework for execution monitoring and analysis that simplifies finding problem causes via automated examination of distributed flow of control in the system.

The following four challenges significantly complicate problem investigation in production systems. Along with each challenge, we list corresponding constraints on the design of a diagnostic tool for such environments.

- **Irreproducibility.** A problem in a production system is often difficult to reproduce in another environment. Most easy-to-locate bugs are fixed at the development stage. Bugs that are left in the system are either intermittent (happen occasionally) or environment-specific (happen only in a particular system configuration). Replicating the exact configuration of a production system in a controlled development environment is not always possible because production systems are often large-scale and distributed. Furthermore, they may use hardware components that are not available to the developer. A tool to diagnose problems in such environments must be able to work on a given system in the field and collect enough information to analyze the problem in a single run.

- **Sensitivity to perturbation.** Since the analysis has to be performed in a real production environment, it should not noticeably disturb the performance of the system. A diagnostic tool for such systems needs to be as low-overhead as possible.

- **Insufficient level of detail.** Modern systems are built of interacting black-box components that often come from different vendors. Finding the causes of most bugs requires detailed understanding of program execution. However, the black-box components provide little support for monitoring their activities. A diagnostic tool for such environments must be able to collect run-time information inside the black-box components and across the component boundaries.

- **Excessive volume of information.** When a system does support detailed monitoring of the execution, the volume of collected data is often exceptionally high. Even if such data could be collected with little overhead, it may be impossible to analyze by hand. Ideally, a diagnostic tool should analyze the data and identify the cause of the problem automatically. Alternatively, it could simplify manual examination by filtering out activities that are not related to the cause of the problem.

1.1 Overview of the Approach

To address these challenges, we have developed a set of techniques for data collection and analysis. Our data collection approach is detailed, low-overhead, and can be dynamically deployed on a running production system. Our data analyses are fully automated for environments characterized by cross-host repetitiveness (e.g., large-scale parallel systems) or cross-time repetitiveness (e.g., client-server systems). As part of our automated approach, we identify causal dependences between events in the system. Such information is also useful for simplifying manual data examination by filtering out activities not related to the cause of the problem.

For detailed and low-overhead data collection, we developed a new technique that we called *self-propelled instrumentation* [86]. The corner stone of self-propelled instrumentation is an autonomous agent that is injected in the system upon a user-provided external event and starts propagating through the code carried by the flow of execution. Here, propagation is the process of inserting monitoring or controlling statements ahead of the flow of execution within a component and across boundaries between communicating components — the boundary between processes, between a process and the kernel, and between hosts in the system. As the system runs, the agent collects *control-flow traces*, i.e., records of statements executed by each component. Our current prototype supports collection of function-level traces.

Our automated diagnostic approach focuses on identification of *anomalies*: intermittent or environment-specific problems. An example anomaly is a failure in an e-commerce system that correctly services most users and returns an error response for a request submitted with a particular version of a Web browser. Another example is a hardware error that causes a host in a large cluster to fail, while the rest of the hosts continue to function correctly. Since anomalies are difficult to reproduce, they are often harder to investigate than *massive failures* that happen on all nodes in the system or for all requests in the system.

The fact that anomalies represent unusual and infrequent system behaviors allows us to identify them with a new outlier-detection algorithm [85]. In large-scale parallel systems built as collections of identical nodes, our approach compares *per-node* control-flow traces to each other to identify nodes with unusual behavior. Such nodes probably correspond to anomalies. In client-server environments, our approach compares *per-request* distributed control-flow traces to each other to identify anomalous requests. As the last step in both environments, we examine the differences between the anomalous trace and the normal ones to identify the cause of the anomaly.

Obtaining per-request traces in client-server systems is a difficult task. Such systems typically service more than one request at the same time. For example, a new HTTP connection request can arrive while the Web server is servicing a previous request from another user. As a result, our agent will follow both requests in parallel, producing a single trace where events of the two requests are interleaved in an arbitrary order. To attribute each event to the corresponding request, our automated approach arranges observed events in causal chains that originate at user-provided start-request events. We build such chains by introducing a set of application-independent rules that dictate when two events are related to each other. We also allow the user to supply additional application-specific knowledge to analysis where necessary.

To illustrate how our techniques fit together, consider a common problem in a Web environment: the user clicks on a link in the Web browser and receives a report of an internal Web server error in return. To find the cause of such a problem, the user can inject the agent into the browser and instruct it to start the analysis upon the next click on the link. The agent starts following the execution of the browser and collects the trace of function calls that the browser makes. When the browser sends an HTTP request to the Web server, the agent crosses the process and host boundaries to start tracing the Web server. If the Web server later interacts with a database, the agent similarly propagates into the database server.

When the Web server internal error happens and the error message reaches the browser, the agent stops tracing and gathers per-host fragments of the trace at a central location. Then, an automated trace analysis engine establishes correspondence between events in different processes to produce a complete trace of events between the click on the link and the manifestation of the internal error. Next, the analysis engine performs trace post-processing to remove activities of other users that were also submitting their requests to the system at the same time, but that are deemed unrelated to the observed error. Finally, the engine processes the trace with anomaly-identification techniques to identify the root cause of the problem or sends the filtered trace to the analyst for manual examination.

Three main phases of this approach are event trace collection, separation of events into per-request traces, and trace analysis. The contributions of this work at each phase are as follows.

1.2 Trace Collection

By following the flow of control and intercepting communication between components, our agent is able to obtain a causal trace of events — record events in the system that occurred in response to a given starting event. Our approach for following the flow of control within a process is similar to techniques for dynamic binary translation [17, 28, 73, 75, 89]. We intercept the execution of an application or the kernel at control-transfer instructions, identify where the control flow will be transferred next, and instrument the next point of interest. Our current prototype intercepts control at function call instructions though finer-grained branch-level instrumentation also can be implemented.

Our approach satisfies three important properties: zero overhead without tracing (not perturbing the system that works normally), rapid on-demand deployment (enabling the analyst to start the analysis on an already-running system), and low overhead with tracing (minimizing perturbation after deployment). We use in-situ instrumentation and offline code analysis to satisfy these properties. Previous approaches for dynamic binary translation and dynamic instrumentation [18, 19, 124] violate at least one of the properties.

Furthermore, the property of on-demand deployment in a distributed environment is not supported by any of the previous instrumentation techniques. As a result, previous analysis approaches typically use static built-in mechanisms for data collection [11, 24, 32, 136]. In contrast, our agent can propagate across communication channels from one process to another, discovering new processes as the system executes. This property enables construction of a new family of on-demand manual and

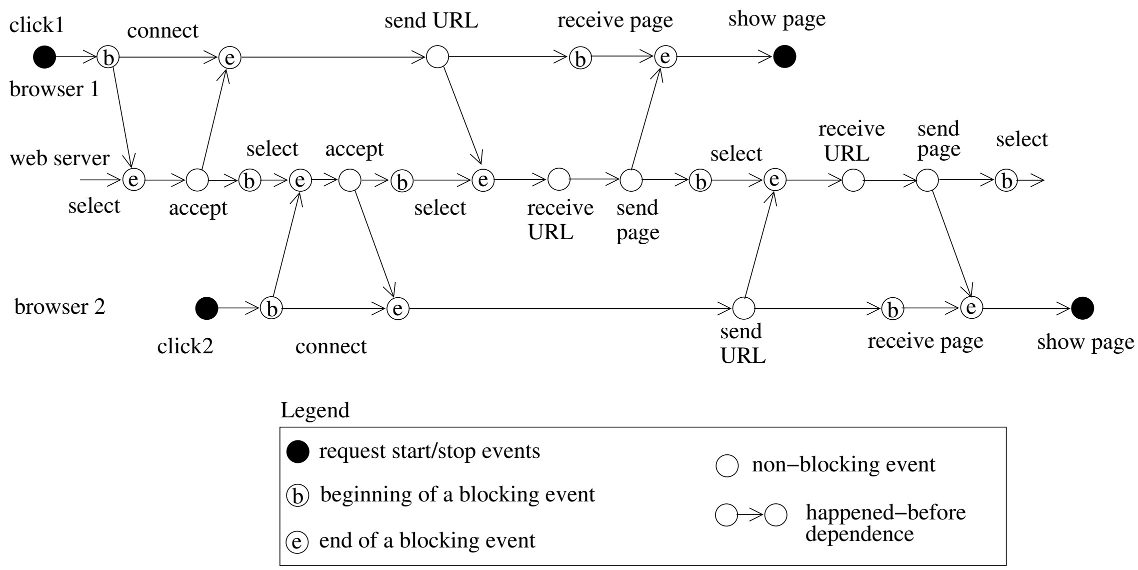


Figure 1.1: *A PDG for a Web server executing requests from two Web browsers*

automated analysis techniques.

To propagate across communication channels, we intercept standard communication operations. For example, when an application invokes a *send* operation on a TCP socket, we identify the address of the peer, put a mark into the channel using the TCP out-of-band mechanism [120], and let the *send* operation proceed. We then contact the remote site, asking it to inject our agent into the process that has that socket open. When injected, our agent instruments the receiving functions so that when the marked message arrives, the agent starts following the receiver's flow of control.

Chapter 3 provides an in-depth description of our techniques for intra-process and inter-process propagation of instrumentation.

1.3 Reconstruction of Causal Flows

A common method to represent and analyze event traces in a distributed system is to construct a Parallel Dynamic Program Dependence Graph, also referred to as a Parallel Dynamic Graph or PDG [29]. The graph is a DAG where nodes represent observed events and edges represent Lamport's happened-before dependences between the events [61]. Namely, nodes for events A and B are connected by an edge if B occurred immediately after A in the same process or if A is an event of one process sending a message and B is an event of another process receiving the same message. Figure 1.1 shows an example PDG for a Web server executing requests from two users who clicked on links in their Web browsers.

A PDG constructed for a real-world distributed system may contain events from several concurrent user requests. Attempts to use such a graph for manual or automated system diagnosis encounter two problems. First, for manual examination, the presence of unlabeled events from multiple unrelated activities is confusing. The analyst needs to know which event belongs to which user request. Second, events that belong to unrelated requests may occur in a different order in different runs. This behavior complicates automated diagnosis via anomaly-identification: a normal PDG being examined may appear substantially different from previous ones and will be marked as an anomaly.

To provide for effective manual and automated analysis, we decompose the PDG into components called *logical flows* or simply *flows*. Each flow is a subgraph of the PDG that represents a semantic activity, such as a set of events that correspond to processing a single user request. Flows are useful for manual trace examination as

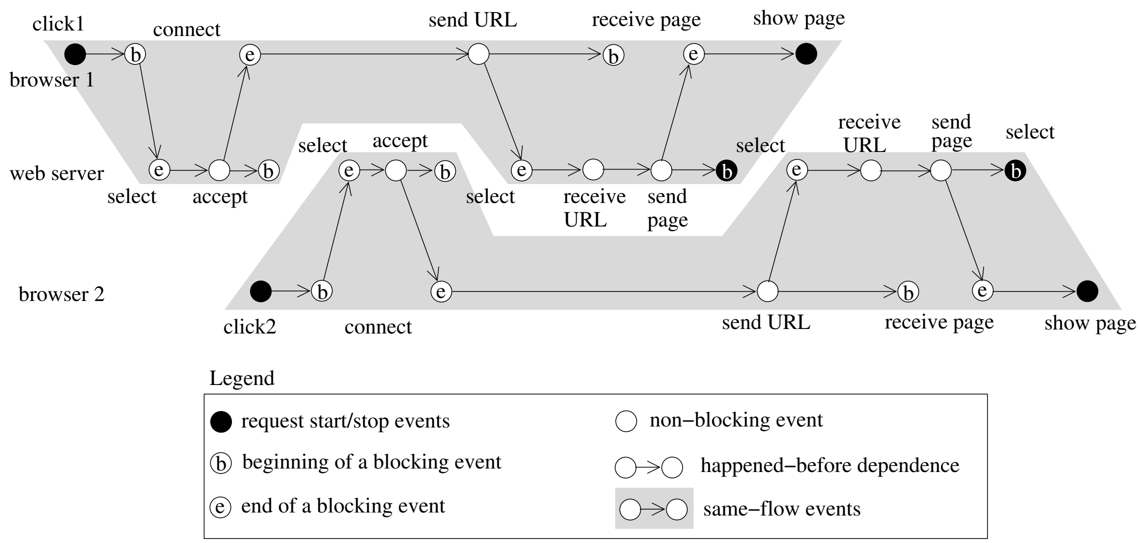


Figure 1.2: *Separated flows for the execution shown in Figure 1.1*

they allow the analyst to determine which event belongs to which request. Flows are also useful for automated analysis as they can be accurately compared to each other. They have less variability than a PDG that interleaves activities of multiple requests.

In most systems that we have analyzed, flows satisfy four application-independent properties. First, each flow has a single start event that happens before other events in the flow. In an e-commerce environment for example, the start event may correspond to a mouse-click event initiating a new HTTP connection. Second, each flow has one or more stop events that happen after other events in the flow. For example, the stop events may correspond to the Web server blocking in the *select* call after replying to the client and to the client showing the Web page. Third, flow graphs are complete, i.e., each event belongs to a flow. For any event in the execution, we must be able to identify the activity to which it corresponds.

Fourth, flow graphs are disjoint, i.e., no event belongs to more than one flow.

This property corresponds to the assumption that each thread of control in the system works on a single request at a time. For example, a thread may service multiple concurrent HTTP requests, but it does not work on them simultaneously. Instead, it switches from one request to another, so that each event belongs to a single request. While the first three properties held in all environments we studied, the fourth property may be violated. In Section 4.2, we provide examples of violations and outline potential techniques for addressing these violations.

Construction of flows can be viewed as a graph transformation problem. Namely, we need to find a set of transformations to convert the PDG into a new dependence graph composed of several disjoint subgraphs that satisfy the four properties of flows. We will refer to this dependence graph as *Flow PDG* or *FPDG*. Our basic graph transformations are removal of an edge and introduction of a transitive edge. We need to remove edges that connect unrelated flows. We need to introduce an edge to connect events of the same flow that were transitively dependent in the original PDG, but became disconnected after removing some edges. Figure 1.2 shows an FPDG corresponding to the PDG in Figure 1.1.

Identifying nodes in the PDG where edges need to be removed or added may require detailed knowledge of program semantics. In our experience however, such locations often can be identified with application-independent heuristics. For example, on receiving a new request, a process typically finishes or suspends handling of the current request and switches to servicing the new one. Therefore, the intra-process edge that connected the receive event to the immediately preceding event in the same process can be removed. To handle scenarios where our heuristics incorrectly remove edges, we allow the user to provide application-specific knowledge to the algorithm in the form of *mapping directives*. Such directives identify pairs of events that should belong to the same flow, so they allow us to insert additional edges into the PDG.

A technique most similar to our flow-construction approach is that of Magpie [11]. Magpie uses three steps to process event traces. First, it separates events into per-request traces using an application-specific set of rules for joining events. Second, for each per-request trace, it builds a dependence tree similar to a single-flow subgraph of our FPDG. Third, it deterministically serializes the dependence tree to produce a request string. Note that Magpie does not use the dependence tree to separate events from different requests. Instead, it relies on application-specific rules provided by the user. Applying such an approach to an unfamiliar system may not be feasible.

In Chapter 4, we describe the details of our flow-construction techniques.

1.4 Automated Trace Analysis

Tracing a distributed system may generate large amounts of data. Analyzing such traces without automated tools is not always possible. In large-scale systems, even finding the failed host is often difficult due to silent and non-fail-stop failures. To automate diagnosis in such environments, we developed a two-step approach. First, we use cross-node and cross-run repetitiveness inherent in distributed systems to locate an anomalous flow. Such a flow may correspond to a failed host in a cluster or a failed request in a client-server system. Second, we examine the differences between that flow and the normal ones to identify the cause of the anomaly.

To locate an anomalous flow, we compare flows to each other to identify outliers, i.e. flows that appear substantially different from the rest. Our algorithm can operate in the unsupervised mode, without requiring examples of known-correct or known-faulty flows. However, if such examples are available, we are also able to incorporate them into analysis for improved accuracy of diagnosis. As a result, our algorithm may have a better accuracy of anomaly identification than previous unsupervised approaches [11, 25, 36]. At the same time, unlike previous fully supervised approaches [32, 136], it is able to operate without any reference data or with examples of only known-correct executions.

Our outlier-detection algorithm supports location of both fail-stop and non-fail-stop problems. It begins by determining whether the problem has the fail-stop property. If one of the flows ends at a substantially earlier time than the others, we assume that the problem is fail-stop, and the flow corresponds to the faulty execution. This simple technique detects problems that cause a process to stop generating trace records prematurely, such as crashes and freezes. However, when all flows end at similar times, it fails to identify the faulty one. Such symptoms are common for infinite loops in the code, livelocks, and severe performance degradations.

For non-fail-stop problems, we locate the anomalous flow by looking at behavioral differences between flows. First, we define a pair-wise distance metric between flows. This metric estimates dissimilarity of two flows: large distance implies that flows are different, small distance suggests that the flows are similar. Next, we identify flows that have a large distance to the majority of other flows. Such flows correspond to unusual behavior as they are substantially different from everything else. They can be reported to the analyst as likely anomalies. To avoid identifying normal but infrequent flows as anomalies, our approach can make use of previous known-correct flow sets where available.

Once the anomalous flow is identified, we locate the cause of the problem with a two-step process. First, we determine the most visible symptom of the problem. For fail-stop problems such as crashes and freezes, the last function recorded in the flow is a likely reason why the flow ended prematurely. For non-fail-stop problems, we identify a function with the largest contribution to the distance between the anomalous flow and the majority of other flows. That function is a likely location of an infinite loop, livelock, or a performance degradation.

Second, if the most visible symptom does not allow us to find the problem cause, we perform coverage analysis. We identify differences in coverage between normal and anomalous flows, i.e., functions or call paths present in all anomalous flows and absent from all normal ones. The presence of such functions in the trace is correlated with the occurrence of the problem. By examining the source code to determine why these functions were called in the anomalous flows, we often can find the cause of the problem. We reduce the number of functions to examine by eliminating coverage differences that are effects of earlier differences. We rank the remaining differences by their time of occurrence to estimate their importance to the analyst.

Chapter 5 provides a detailed description of our data analysis approaches.

1.5 Dissertation Organization

To summarize, the rest of the dissertation is structured as follows. Chapter 2 surveys related work in the areas of dynamic program instrumentation, flow reconstruction, and automated problem diagnosis. Chapter 3 describes self-propelled instrumentation within a component and across component boundaries. Chapter 4 introduces techniques for separating different semantic activities into flows. Chapter 5 describes our automated approach for locating anomalies and their root causes. Chapter 6 presents examples of real-world problems that we were able to find with manual trace examination and our automated techniques. Chapter 7 brings all the techniques together and suggests directions for future research.

Chapter 2

Related Work

Debugging and performance analysis of distributed systems have been active areas of research for several decades and a variety of different approaches have been proposed. Here, we only survey dynamic analysis techniques, i.e., techniques that perform diagnosis using information collected at run time. We begin by summarizing complete approaches for finding bugs and performance problems. Then, we focus on individual mechanisms that could be used at different stages of the process: trace data collection, causal flow construction, and data analysis.

2.1 Overview of End-to-end Solutions

Existing approaches to debugging and performance analysis can be classified as trace-mining techniques, profilers, interactive debuggers, and problem-specific techniques. When comparing these classes, we use four key criteria that determine their applicability to analysis of production systems: autonomy (ability to find the cause of a problem with little human help), application-independence (ability to apply the technique to a wide range of applications), level of detail (identifying the cause of the problem as precisely as possible), and low overhead.

2.1.1 Trace-mining Techniques

There are several approaches that collect traces at run-time and use data mining techniques to locate anomalies and their causes [11, 24, 32, 36, 68, 136]. The key difference between them lies in their data analysis techniques, which we discuss in more detail in Section 2.4. Here, we only survey two projects with the goals most similar to our work: Magpie [11, 12] and Pinpoint [24, 25, 26].

Magpie builds workload models in e-commerce request-based environments. Its techniques can be used for problem diagnosis as follows. First, Magpie collects traces of several types of events: context switches, disk I/O events, network packet transmission and reception events, inter-component communication events, and request multiplexing and demultiplexing events. To obtain such traces, Magpie uses pre-existing ETW (Event Tracing for Windows) probes [78] built into standard Windows kernels and also modifies applications and middleware components to emit additional trace statements at several points of interest.

Second, Magpie separates the obtained event trace into per-request traces. To assign each event to the correct request, Magpie uses an application-specific schema, that is, a set of rules that dictate when two events should belong to the same request. Each request starts with a user-provided start event and additional events are joined to it if their attributes satisfy the schema rules. Ultimately, each request is represented as a string of characters, where each character corresponds to an event.

Third, the obtained per-request strings are analyzed to identify anomalies, that is infrequent requests that appear different from the others. Magpie uses behavioral clustering to group similar request strings together according to the Levenshtein string-edit distance metric [34]. In the end, strings that do not belong to any sufficiently large cluster are considered anomalies.

Finally, to identify the root cause (the event that is responsible for the anomaly), Magpie builds a probabilistic state machine that accepts the collection of request strings [12]. Magpie processes each anomalous request string with the machine and identifies all transitions with sufficiently low probability. Events that correspond to such transitions are marked as the root causes of the anomaly.

The approach of Magpie is autonomous and low-overhead. However, it requires application modifications for trace collection and relies on detailed knowledge of application internals for request construction. Our approach works on unmodified systems and uses little application-specific knowledge. Furthermore, the analysis techniques of Magpie may have lower diagnostic precision and accuracy than ours. Magpie only identifies a high-level event that may have caused the problem, while we identify the anomalous function. Although precision of Magpie's diagnosis could be improved by applying its techniques to our function-level traces, this approach is likely to be less accurate than ours. Function-level traces are highly variable and not all differences between them correspond to real problems. Our approach eliminates some of this variability by summarizing traces and comparing the summaries rather than the original traces.

Pinpoint uses different mechanisms for trace collection, building requests strings, and analyzing the strings. However, its limitations are similar to those of Magpie. Pinpoint requires application and middleware modifications to collect traces and attribute events to the corresponding requests. Its analysis techniques have only been applied to coarse-grained diagnosis, identifying a faulty host or a software module. For the first step of diagnosis, identification of an anomalous request, Pinpoint uses original traces. This approach may not be accurate for function-level traces due to their variability.

For the second step of diagnosis, locating the cause of the anomaly, Pinpoint uses trace summaries (vectors of component coverage for each request) rather than original traces. It looks for correlation between the presence of a component in a request and the request failure. One of our techniques applies a similar idea to function-level coverage data. In our experience, this technique is often capable of locating the set of suspect functions, but requires manual effort to find the actual cause in the set. We simplify manual examination by eliminating differences in coverage that are effects of an earlier difference. We also rank the remaining differences so that more likely problem causes are ranked higher and can be examined first.

2.1.2 Delta Debugging

To locate bugs automatically, the Delta Debugging technique narrows down the cause of a problem iteratively over multiple runs of an application [138]. Delta Debugging requires the user to supply two sets of input such that the application works correctly on the first set and produces a detectable failure on the second. The user also supplies a mechanism to decompose the difference between the two input sets into smaller differences. By applying some of these differences to the original correct input, Delta Debugging can produce a variety of test input sets. By generating test sets iteratively, the Delta Debugging algorithm tries to identify the minimum set of differences that change the behavior of the application from correct to failed.

This minimum difference in the input introduces a set of differences in values of program variables. In the second step, Delta Debugging compares the values of all variables between the correct and the incorrect run and identifies such differences. If we set some of these variables in the incorrect run to the values obtained in the correct run, the application may start working correctly. Delta Debugging uses an iterative process to identify a minimum set of variables that cause the incorrect run to complete successfully. Code locations where these variables are modified often point to the problem cause.

The approach of Delta Debugging has three limitations. First, it requires extensive user intervention. Decomposing the difference between two input sets into smaller differences often requires understanding of the semantics of the input. Second, Delta Debugging requires the problem to be reproducible: the behavior on the same set of inputs should be the same over multiple runs. This limitation can be addressed by using a replay debugger to record and replay the executions; the combination of replay debugging and Delta Debugging has been used to narrow the cause of an intermittent problem to a particular thread scheduling decision [31].

Finally, Delta Debugging is invasive. It assumes that the application state can be disturbed and the application can be repeatedly crashed without producing catastrophic results. This assumption is unlikely to hold in real-world production systems. In contrast, our approach uses a non-invasive monitoring approach. It does not modify the application state and operates on traces of observed executions.

2.1.3 Performance Profiling

A popular approach for performance analysis is to collect aggregate performance metrics, called a *profile* of the execution. Typically, the profile contains function-level performance metrics aggregated over the run time of an application (e.g., the total number of CPU cycles spent in each function). The profile can be collected for a chosen application [9, 15, 43, 82, 121], the kernel [58, 77], an application and the kernel [84], or even the entire host, including all running processes and the kernel [5, 33]. Two common mechanisms for collecting such data are statistical sampling [5, 33, 43, 58, 77, 121] and instrumentation [9, 15, 82, 84, 87].

This approach is highly autonomous, it identifies the bottleneck functions without human guidance. It has low overhead that can also be controlled by changing sampling frequency or removing instrumentation from frequently executed functions if it proves costly [82]. Profiling does not need application-specific knowledge and many sampling and instrumentation techniques do not require application recompilation. Function-level data often provides sufficient detail for the developer to locate the cause of a problem.

Despite these advantages, the focus on aggregate metrics makes profiling unable to diagnose some performance problems. Finding the root causes of intermittent problems often requires raw, non-aggregated performance data. For example, interactive applications are often sensitive to sporadic short delays in the execution. Such delays are visible to the user, but they are ignored by profilers as statistically insignificant. Similarly, finding a problem in processing a particular HTTP request may not be possible with a profiler if the Web server workload is dominated by normal requests. Most anomalies in that environment would be masked by normal executions.

2.1.4 Interactive Debugging

A common approach for finding bugs in the code is to use an interactive debugger [70, 118]. In a typical debugging session, the programmer inserts a breakpoint in the code, runs the application until the breakpoint is hit, and then steps through individual program statements examining program variables and other run-time state. This debugging model is powerful, but it is not autonomous, requiring user help at all stages of the process.

Furthermore, interactive debugging introduces significant perturbation to the execution. There are several common scenarios when this perturbation may be unacceptable. If the application is parallel (e.g., multithreaded), single-stepping through its code may change the order of events and hide the problem that the analyst is trying to debug. If the application is latency-sensitive, interaction with a human analyst may significantly affect its execution (e.g., a video playback program will discard several frames and introduce a visible skip in the playback).

To support such scenarios, some debuggers use the concept of autonomous breakpoints [115, 128]. When such a breakpoint is hit, the debugger will execute actions that are pre-defined by the user and then resume the application immediately, avoiding the perturbation introduced by interactive techniques. The actions are flexible and can be used to examine and manipulate the application state. Typically, they are used to examine the state (e.g., print the value of a variable to the log), which makes them similar to tracing techniques that we describe in Section 2.2.

2.1.5 Problem-specific Diagnostic Approaches

Aside from the mentioned projects, there exists a variety of more specialized approaches that aim at locating a particular kind of problem. Since they do not attain the level of generality of approaches discussed above, we only survey them briefly. Several tools aim at detecting memory-related errors in applications [48, 89, 95]. Such tools can detect buffer overruns, memory leaks, attempts to use uninitialized memory, *free* operations on an already-freed memory region, and similar errors. There is also an extensive body of work on finding race conditions in multithreaded programs, e.g., [89, 91, 107, 111]. These techniques monitor memory accesses as well as *lock* and *unlock* operations performed by different threads to make sure that all shared memory locations are guarded by locks.

2.2 Mechanisms for Trace Collection

To detect bugs and performance problems, our approach collects function-level traces via self-propelled instrumentation. We then look for anomalous traces and identify the root cause of the anomalies. In this section, we survey previous trace collection mechanisms and compare them to self-propelled instrumentation.

Our main focus is on *control-flow tracing* techniques, i.e., techniques that record the sequence of executed instructions, basic blocks, or functions. Control-flow traces are often used for performance analysis as they allow the analyst to find where an application spends the majority of its time [23, 51, 80, 87, 101]. Control-flow traces are also useful for debugging as they allow the analyst to explain how an application reached the location of a failure [7]. Although not always sufficient to explain why the failure happened, this information is often helpful in diagnosis: many anomalies are manifested by deviations of the control flow from common behavior. Using techniques similar to Magpie [12] and Pinpoint [25], the analyst can identify an early symptom of the problem by examining such deviations.

An alternative debugging approach is to trace memory accesses of an application, recording all values loaded or stored to memory. We will refer to this approach as *data-flow tracing*. It allows the analyst to explain why a variable has a particular value at the location of a failure by examining the variables used to compute the value. This approach mimics a manual technique commonly used by programmers [130]. Although powerful, tracing all memory accesses introduces significant performance overhead and may be prohibitively expensive for use in production environments.

A common approach to reduce this overhead is to record only events necessary to replay the execution [29, 30, 64, 92, 106, 116]. Further details can then be obtained during replay, when the overhead is no longer the limiting factor. Although this technique has been primarily used in interactive debuggers, our anomaly-identification approach might be able to analyze data-flow traces automatically. Since support for deterministic replay is yet to appear in production systems, the feasibility of this approach is subject for future work. To date, we have shown effectiveness of our analyses on control-flow traces and we focus on mechanisms for control-flow tracing below.

Control-flow traces can be collected via static instrumentation [53, 63, 69, 80, 105, 117], dynamic instrumentation [18, 21, 123], dynamic binary translation [17, 73, 75, 89], single-stepping [104], and simulation [4]. We begin by describing the dynamic binary translation approaches as they are most similar to our work on self-propelled instrumentation. Then, we overview the other tracing techniques.

2.2.1 Tracing via Dynamic Binary Translation

Binary translation is a technique for converting programs compiled for one instruction set (guest ISA) to run on hardware with another instruction set (native ISA). A dynamic binary translator performs this conversion on the fly, by translating and executing the application a fragment (one or more basic blocks) at a time. Of special interest are the dynamic binary translators that perform translation from an instruction set to the same instruction set. This technique has been used for run-time optimization [8] and instrumentation purposes [17, 73, 75, 89].

Collecting control-flow traces via dynamic binary translation has two advantages. First, it does not require application-specific knowledge and can be used to obtain detailed function-level or basic block-level traces from any application. Second, this technique can support on-demand activation, letting the user start tracing only when necessary. This capability is crucial for analysis of production systems. It allows the user to avoid introducing any overhead when a system works correctly and analysis is not required. DynamoRIO [17] already provides this functionality and it can also be implemented by DIOTA [75], PIN [73], and Valgrind [89].

To perform code instrumentation, these frameworks use similar mechanisms. First, a translator copies a fragment of an application's code to a heap location in the translator's address space, referred to as the *fragment cache*. Next, it allows the generated fragment to execute. To regain control when the fragment finishes executing, the translator modifies all branches that lead out of the fragment to point back to the translator's code. When one of such branches executes, the translator receives control and starts building the next fragment from the destination of the branch.

To obtain an execution trace, this technique can be used to insert calls to tracing routines into new fragments as they are being generated. For example, by inserting trace calls before call and return instructions in each fragment, a binary translator can obtain a function-level trace of an execution. These capabilities are similar to those of self-propelled instrumentation.

There are three key differences between our approaches. First, our technique has low warm-up overhead, allowing us to start tracing with less perturbation to the execution. To provide such capability, we use *in-place* instrumentation, that is, we modify the original code of the application and avoid generating its copy in the fragment cache. Building the copy of the code is an expensive operation as all position-dependent instructions need to be properly adjusted. We lower overhead even further by performing code analysis in advance with an offline tool. When an application executes a function for the first time, we can instrument it without prior analysis. As a result, we introduce little perturbation even to code that is not executed repeatedly.

The disadvantage of our approach is that it may not trace parts of code that are not discovered statically [46, 86]. However, this limitation can be addressed with a hybrid technique, analyzing as much code offline as possible and relying on dynamic code analysis for the rest. A similar approach has proved viable for dynamic x86-to-Alpha binary translation in FX!32, where a persistent repository was used for keeping translated code [28].

Second, our technique can be used for both application and kernel instrumentation. Control-flow traces of the kernel proved useful for diagnosis of two non-trivial problems discussed in Chapter 3. We use the same mechanisms and the same code base for compiling our user- and kernel-side agents. Note that similar capabilities can also be implemented by binary translation approaches.

Third, our agent can propagate across component boundaries. It can cross the user-kernel boundary when an application performs a system call and start tracing both the application and the kernel. It can cross the process and host boundaries when one application communicates to another. Previous dynamic binary translation techniques do not provide such capabilities and are less suitable for system-wide analysis in distributed environments.

2.2.2 Other Control-flow Tracing Techniques

There is a variety of other techniques for detailed application-independent tracing. To obtain function-level traces, tools use single-stepping [104], simulation [4], source-level instrumentation [53, 69, 80], binary rewriting [63, 105, 117], trap-based dynamic instrumentation [21, 123], and jump-based dynamic instrumentation [18, 124].

Single-stepping is an old technique that allows a tool to monitor another process by pausing it after every instruction that it executes. To obtain a function-level trace, the itrace tool [104] single-steps through instructions in an application and generates a trace record whenever a call or a return instruction is encountered. Similarly, we could catch entries/exits from individual basic blocks by generating trace records for every control-transfer instruction (such as jumps, branches, calls and returns). This approach can be enabled at run time and therefore introduces zero overhead when tracing is not required. However, it is highly intrusive when tracing is being performed, because it incurs a context switch after every instruction in a traced process. Although it may be tolerable for debugging some time-insensitive applications, this approach is not suitable for performance analysis and finding time-sensitive bugs.

To shield applications from any tracing perturbation, developers of embedded systems often use simulators. With recent advances in simulation technology [76, 108], similar techniques have been applied to general-purpose systems [4]. This approach is able to collect detailed function-level or even instruction-level performance data with no visible perturbation to the system's execution. Furthermore, it provides deterministic replay capabilities, useful for finding intermittent bugs.

Despite these advantages, simulators have three critical drawbacks. First, they introduce the slowdown of 30–75 times relative to native execution [76]. This fact complicates collection of data for long-running or interactive applications. Second, to diagnose a real-world problem with such techniques, the analyst would need to reproduce the problem in a simulated environment. For subtle and intermittent bugs that are highly sensitive to changes in system configuration, it may be difficult to achieve. Finally, it is necessary to run all components of the system under a simulator, which is not always possible. Simulating a multi-host system and subjecting it to a real stream of requests coming from users through the Internet is probably not feasible.

To diagnose problems in real-world environments with little perturbation to the execution, analysts often employ code instrumentation techniques. For example, by using a tool for source-code analysis, the analyst could identify function calls in an application, modify the source code to insert trace statements around every call and recompile the application [87, 101]. This approach collects a function-level trace of the execution.

When tracing is being performed, source-level instrumentation introduces low run-time overhead, often lower than that of other instrumentation mechanisms. By using a *code-cloning* technique similar to [3], it can also have zero overhead when tracing is not required. Namely, a compiler can produce two copies of each function in the same application: the instrumented one and the original one. Normally, the system executes the uninstrumented copy. To initiate tracing, the analyst sends a signal to the running application, the signal handler transfers control to the instrumented copy, and the system starts generating traces.

A disadvantage of code cloning is that it doubles the code size. While not a problem for small applications, it may have adverse performance effects on larger systems. Keeping two copies of the code may increase the number of I-cache misses, TLB misses, and page faults. Furthermore, the increased size of the code presents additional problems in software distribution and installation.

A disadvantage of source-level instrumentation in general is the necessity to have the source code for all software components. This requirement is hard to satisfy in complex environments, where different components are often produced by different vendors. To address this limitation, instrumentation tools can use binary-rewriting techniques [63, 105, 117]. To obtain a function-level trace, binaries and libraries in the system can be processed to insert trace statements around all call instructions. This approach works well on proprietary applications, since it does not require the source code to be available. It has low tracing overhead, comparable to that of source-level instrumentation. Although we are not aware of any binary rewriter that uses the code-cloning technique discussed above, code cloning could also be applied to binary rewriters to introduce no overhead when tracing is not necessary.

Despite these advantages, binary rewriting has three shortcomings that we address with self-propelled instrumentation. First, binary rewriting requires the analyst to anticipate potential problems and instrument binaries in the system before running them. Our approach can be applied to applications and the kernel on the fly, making it suitable for diagnosing problems in long-running systems without restarting them.

Second, some instrumentation decisions can only be made at run time. For example, we can avoid tracing a function if we discover that it is executed frequently and tracing it introduces high overhead. We could also trace infrequently executed code at a finer level of detail, thus increasing the diagnostic precision (infrequent code is typically less well tested) without introducing high overhead. Third, our approach can be augmented with run-time code discovery capabilities to trace code undetected by static analysis and code generated at run-time. Tracing such code may not be possible with binary rewriting.

To insert trace statements on the fly, a tool can use dynamic instrumentation techniques. Such techniques can be categorized into trap-based and jump-based ones. An example of a trap-based tool is LTrace [21]. This tool traces calls to functions in shared libraries by inserting breakpoint traps at entries of all exported functions. The Solaris Truss tool [123] uses the same technique, but is able to instrument all functions present in the symbol table, not only those exported by shared libraries. This approach is simple to implement, but incurs substantial run-time overhead. The cost of a breakpoint trap on every function call is prohibitively expensive for performance analysis and for finding many timing-sensitive bugs.

A more efficient approach for function-level tracing is to use jump-based instrumentation capabilities, such as those provided by Dyninst API [18]. This approach achieves low overhead with tracing and zero overhead without tracing. However, it takes significant time to transition between the two states. For time-sensitive applications (GUI, multimedia, telephony) and large concurrent multi-user systems (Web and database servers), this delay may be unacceptable.

Insertion of trace statements at all locations in the code is an expensive operation for two reasons. First, even if an application uses a small fraction of all functions in its binary and shared libraries, all functions need to be instrumented before this approach can start generating complete traces. Second, a Dyninst-based tool and the traced application have separate address spaces. The tool thus requires one or several time-consuming system calls to insert each instrumentation snippet into the application's code. In contrast, self-propelled instrumentation and dynamic binary translators instrument only the functions actually executed. Furthermore, they share the address space with the application and thus can perform instrumentation with fast local memory operations.

2.2.3 Summary of Trace Collection Approaches

Dynamic binary translators are superior to other previous techniques for obtaining control-flow traces as they can be deployed on an unmodified system without recompilation or restart, have zero overhead when tracing is not necessary, and have low overhead with tracing. Their start-up overhead is also lower than that of other dynamic instrumentation techniques since dynamic binary translators instrument only the code actually executed and perform instrumentation in the same address space via local memory operations. However, this start-up overhead is still substantial because of the need to decode each instruction on its first occurrence, copy it to the fragment cache, and adjust it, if the instruction is position-dependent.

Self-propelled instrumentation further lowers that part of the overhead by discovering code in advance with an offline tool and performing in-place instrumentation, thus eliminating the need to relocate the entire code of the application to the fragment cache. Unlike previous tools for dynamic binary translation, we also support instrumentation of a live operating system kernel. Finally, the unique feature of self-propelled instrumentation is the ability to cross component boundaries, propagating from one process into another or from a process into the kernel. This property is crucial for system-wide on-demand analysis of production environments.

2.3 Reconstruction of Causal Flows

A trace from a real-world distributed system may contain interleaved events from several concurrent user requests. Such a trace is difficult to analyze both manually and automatically. To simplify trace analysis, several previous approaches attempted to separate events from different requests into per-request traces that we call flows [1, 11, 24, 60, 66, 81]. We will use two criteria to compare such approaches: accuracy (correctly attributing events from the same request to the same flow) and application-independence (ability to apply the technique to a new unfamiliar application).

2.3.1 Aguilera et al.

Aguilera et al. studied automated techniques for building causal flows without application-specific knowledge [1]. They treat hosts as black boxes, monitor network packets on the wire, and use statistical methods to match packets arriving at each host with subsequent packets leaving the same host. Such packets are then attributed to the same causal flow. Their key matching criterion is the delay between receiving the first packet and sending the second packet. For example, if a statistically significant fraction of network packets from host B to host C are sent within a certain time interval after host B receives a packet from host A, then the packet received from A and the packet sent to C are attributed to the same causal flow.

This approach is application-independent and can be tested on any network-based system. It identifies frequent causal paths in the system and thus may be used for locating performance bottlenecks. However, its probabilistic techniques may not be able to infer causal paths of infrequent or abnormal requests as such requests would be considered statistically insignificant. Results presented by the authors indicate that the approach generates many false paths for rare requests. As a result, this approach may not be accurate for locating the causes of infrequent bugs and performance anomalies. In contrast, our rules are not probabilistic and can be used for reconstructing even anomalous flows.

2.3.2 DPM

DPM presented another application-independent approach for causal path reconstruction [81]. DPM is a framework for performance analysis of complex distributed systems. It assumes that the system is built as a collection of server processes that accept requests from user processes and communicate with each other by exchanging messages. To monitor the execution of the system, DPM used in-kernel probes to collect traces of message send and receive events for server processes. Next, it built a program history graph from the events, similar to the PDG described in Section 1.3.

Once the graph is constructed, DPM generates causal strings, where each character represents the name of a server process sending or receiving a message. Each string starts as a single character corresponding to a receive event from an external user process. Additional characters are appended to each string by traversing the graph from that event using two simple rules. First, a send event in one process and the corresponding receive event in another process are assigned to the same string. Second, a receive event in a process and the immediately following send event in the same process are also assigned to the same string.

Similar to DPM, our approach uses rules to build causal chains by traversing the graph from user-provided start events. The two rules proposed by DPM are also part of our set. However, there are two key differences between our approaches. First, we support applications that may violate one of the rules. For example, an application that receives requests as they arrive and then uses a FIFO queue to service them may violate the second rule. (The trace will contain two receive events followed by two sends. DPM will assign the second receive and the first send to the same string even though they correspond to different requests.) To address such scenarios, we allow the user to provide additional application-specific directives to the analysis.

Second, we support applications that perform intra-request concurrent processing. For example, a server process may accept a request, partition it into several fragments, handle the fragments concurrently in several helper processes, and recombine the results. Such a request is best represented as a dependence graph containing several causal paths. Our approach is able to construct such per-request graphs from a common PDG, identify causal paths within them, and compare the graphs across requests. In contrast, DPM assumed that events within each request are serialized and the system only contains inter-request concurrency.

2.3.3 Magpie

Magpie uses a different approach for attributing events to causal paths in e-commerce environments. It collects traces of system-level and application-level events using the Event Tracing for Windows framework [78]. To build a per-request trace, Magpie starts with a user-provided start event and joins additional events to it if their attributes satisfy certain application-specific rules, referred to as the *schema*. For example, the rules may specify that events with the same thread identifier belong to the same request if they occur after a thread unblocks and before it blocks again.

This approach would allow Magpie to avoid source code modifications, provided that all points of interest in the system already had tracing probes built-in. At present however, probes in existing systems are not sufficient for request reconstruction and Magpie requires additional middleware and application-specific instrumentation at all points where requests are multiplexed and demultiplexed.

Once a per-request trace is constructed, Magpie uses causal dependences between events to build a binary tree structure for each request. Each tree is similar to our concept of a flow, as described in Section 1.3. Magpie further flattens the tree using a deterministic depth-first traversal to produce a request string. Each character in the string represents an event in the original trace.

The key difference between our flow-construction approaches is that Magpie relies on the user to specify rules for building per-request traces while we provide an application-independent set of rules that can be used to build such traces (typically, without user's help). To apply the approach of Magpie to a new system, the user needs to identify all locations in the code where requests can be multiplexed and demultiplexed and provide tracing instrumentation for such locations. In contrast, our observation is that such locations often can be identified automatically. We resort to user's help and use an approach similar to that of Magpie only when our application-independent rules are insufficient for proper request construction.

2.3.4 Pinpoint

Pinpoint reconstructs causal paths of requests through J2EE e-commerce systems. First, it modifies the Jetty web server to assign a request identifier to each incoming HTTP connection. It then stores the identifier in thread-local storage so that the receiving thread can access it. When the thread issues a remote call, Pinpoint passes the identifier to the peer component via the modified Remote Method Invocation protocol. Pinpoint also modifies the J2EE middleware to record the names of communicating components in the trace along with the request identifier.

This approach is middleware-specific as it requires modifications to the inter-component communication protocol, which may not always be feasible. Furthermore, the identifier can be kept in thread-local storage only for systems where a thread does not switch from one request to another unless it receives a message that belongs to the second request. In our experience, this assumption is violated for event-driven applications that use queues to hold multiple incoming requests. When such an application completes servicing one request, it may dequeue another request and start servicing it without any intervening receives.

Pinpoint has also been applied to traces from two commercial systems, eBay and Tellme Networks [25]. Both systems provide custom logging frameworks that are used by the middleware and the applications. To assign identifiers to incoming requests and pass them through the system, both frameworks rely on middleware or application-specific instrumentation. Although such approaches can accurately track the paths of requests through components, they cannot be readily applied to other systems.

2.3.5 Li

The work by Li demonstrated the viability of causal chain construction for multi-component COM and CORBA systems without application-specific source code modifications [66]. This project aims at visualizing and analyzing remote call chains in component-based distributed systems. It traces cross-component function call and return events and builds system-wide chains of nested calls.

When a client issues an RPC call, it generates a unique identifier (UUID) that serves as the name of the call chain. The UUID is then propagated from the caller to the callee and stored in log records to attribute traced calls to this call chain. To communicate the UUID between the caller and the *remote* callee, Li uses a modified IDL (Interface Description Language) compiler that adds an extra argument to generated call stubs. To make the UUID available to a *local* callee (across a local function call), the UUID is also saved in thread-local memory, similar to the approach used by Pinpoint. If the callee makes a nested RPC call, it retrieves the UUID from thread-local memory and forwards it to its callee and so on.

There are two key differences between Li's approach to causal chain construction and ours. First, Li's approach requires IDL modifications and system recompilation. Our approach works on unmodified binaries and can be deployed at run time. Second, Li's approach only works for systems with RPC-based communications while our approach works for a more general class of message-based systems. Namely, Li's work uses the RPC-specific mechanism for passing the UUID across communications. It also assumes that all events in one thread between the start of a call and a reply belong to the same request. While true for RPC communications, this assumption is often violated for message-passing applications, as discussed in Section 2.3.4.

2.3.6 Krempel

The work by Krempel used causal flows for performance analysis and understanding of the behavior of MPI-IO applications [60]. In parallel I/O environments, a single MPI-IO call in an application can result in multiple disk accesses on multiple nodes of a parallel file server. For detailed performance analysis of an application it may be necessary to identify the mapping between high-level MPI-IO calls and the induced low-level activities on the server.

The approach of Krempel uses four steps to establish this mapping. First, for each observed MPI-IO call, this project creates a system-wide unique identifier, referred to as the *callid*. Second, the project makes the *callid* available throughout the application by placing it in a global variable (the application is assumed to be single-threaded). Third, the client-side library of the PVFS2 parallel file system [94] accesses the global variable and sends it to the server with each request via a modified protocol. Fourth, the *callid* is made available throughout the multithreaded PVFS2 server by adding it as an extra field to a request structure that is already being passed through system layers. Trace statements can then store the *callid* in all log records.

Although this approach can accurately track how a request propagates through the system, it requires substantial modifications to multiple components and protocols. Some of the techniques are also specific to a particular implementation and may not apply to a wider range of systems. For example, if the request structure was not accessible at all layers in the server, this tool would need a different technique to propagate the *callid* through the code.

2.3.7 Summary of Causal Flow Reconstruction Approaches

The causal flow reconstruction approaches that we presented can be categorized into two classes: application-independent (Aguilera et al. and DPM) and application-specific (Magpie, Pinpoint, the works of Li and Krempel). The application-independent approaches can be applied to a variety of systems, but they are not always accurate and can attribute events to wrong flows. The application-specific approaches accurately identify flows, but support only one system or a narrow class of systems. Applying such techniques to other environments requires substantial effort. This dissertation presents a hybrid approach that requires little or no application-specific knowledge while identifying flows with relatively high accuracy.

2.4 Trace Analysis

Once run-time data is collected and separated into different flows, anomalous flows can be identified to find the root cause of the anomalies. We begin by surveying previous techniques for anomaly identification and then discuss approaches for root cause analysis.

2.4.1 Finding Anomalies

We use two criteria to compare previous anomaly-identification approaches. First, an approach should be as accurate as possible, minimizing the number of false positives (normal executions reported as anomalies) and false negatives (anomalous executions reported as normal). False positives increase the burden on the analyst who is interpreting the results of automated analysis; false negatives prevent us from finding the true cause of a problem.

Second, an approach should use as little help from the user as possible. Some approaches rely on user-specified fault-detection mechanisms to classify runs into normal and anomalous. Others (*supervised approaches* [47]) detect anomalies from the collected data themselves, but require prior examples of known-correct and known-failed executions. Finally, some approaches are unsupervised, working without any prior examples.

To determine whether a request completed normally or failed, Pinpoint looks for known problem symptoms using auxiliary fault detectors. Some symptoms are externally-visible and can be observed by monitoring system output. Pinpoint locates such symptoms with external fault detectors. For example, it uses a network sniffer to detect TCP resets or time-outs that indicate node crashes or freezes. Similarly, it detects HTTP errors that indicate invalid requests or internal server errors. However, some errors are not externally-visible (e.g., detected by the system, but not reported to the user). Pinpoint detects some of them with an internal fault detector that looks for exceptions that cross component boundaries.

Fault detectors are effective at locating many problems with known symptoms. However, not all known symptoms can be easily detected. Especially challenging are non-fail-stop failures where the system continues running but stops making further progress. For example, reliably distinguishing a misbehaving server in an infinite loop from a normally-running server that executes a long complex request may be difficult. Furthermore, fault detectors may not be suitable for finding problems with symptoms never seen before. For example, if an application does not crash but produces incorrect results, a fault detector may not be able to determine that the results are incorrect.

In our experience however, problems with hard-to-detect or previously-unknown symptoms often can be distinguished from normal executions by looking at differences in internal program behavior. For example, an application in an infinite loop spends most time in a different part of the code than a normally-functioning one. An event trace captured when an application misbehaves may differ from the ones captured during correct executions. This property is used by several approaches to locate anomalous executions [11, 25, 32, 36, 136].

Magpie uses a unsupervised clustering algorithm for anomaly identification. Once it obtains per-request strings, it looks for strings that appear different from the others. Similar request strings are clustered together according to the Levenshtein string-edit distance metric [34]. In the end, strings that do not belong to any sufficiently large cluster are considered anomalies.

There are two key differences between Magpie and our approach. First, clustering in Magpie is unsupervised. It looks for unusual request strings and does not use a priori knowledge to categorize them. In practice however, we found that unsupervised techniques often generate false positives, reporting unusual but normal behaviors as anomalies. For example, the master node of a master-worker MPI application may behave substantially different from the worker nodes. Workers perform units of work, while the master distributes the units among the workers. Unsupervised approaches would flag the normal behavior of the master as an anomaly.

To address this limitation, we designed a one-class classification [126] algorithm. It can use prior known-normal behaviors, if available, to reduce the number of false positives and improve the accuracy of anomaly identification. For example, if a previous correct run of the master-worker MPI application contained a trace similar to that of the master node in the current run, we would consider the current behavior of the master normal.

Second, our function-level traces are substantially more detailed than event traces of Magpie. This property allows us to perform more precise diagnosis. However, it also presents challenges in analysis. Magpie groups raw request strings together based on the edit distance between them. The variability of our function-level traces makes this similarity metric ineffective: similar activities may produce substantially different raw traces. To combat variability, we first summarize traces by replacing all events between each pair of successive message-passing operations with a call profile. We then analyze similarity between computed summaries. Note that our summaries are at least as detailed as Magpie's strings since we retain all message-passing operations intact. Call profiles allow us to further improve diagnostic precision.

Pinpoint uses another unsupervised technique for finding unusual requests. This technique allows it to locate failures not identified by its fault detectors. First, Pinpoint builds a PCFG (probabilistic context-free grammar) that accepts a collection of previously-observed request strings. A derivation of the form $A \rightarrow BC$ in the grammar represents an observation that component A invoked components B and C in a sequence. Associated with each derivation is its observed probability of occurrence.

Second, Pinpoint determines whether a new request string is accepted by the constructed PCFG. If so, Pinpoint multiplies the probabilities of the corresponding derivations to compute the probability that the string will be generated by the PCFG. Strings with low probabilities are reported to the analyst as anomalies.

Similar to the approach of Magpie, this technique is unsupervised and works with unlabeled traces. It does not incorporate the fact that a trace is known-normal or known-faulty into the analysis. Furthermore, building an accurate PCFG requires a significant number of traces. Even for small grammars, the approach needs several hundred examples [20]. For our highly-variable function-level traces, the size of their grammar and the number of required traces are likely to be substantially larger. However, the number of traces may be limited by the number of hosts in the system (for approaches that detect anomalies using cross-host repetitiveness) or by the number of test runs available (for approaches that use cross-run repetitiveness). Our approach operates on less-variable trace profiles and can work even with as few as 5–10 traces.

Dickinson et al. use an unsupervised technique for classifying reports coming from beta testers [36]. Their work aims to reduce the number of reports an analyst has to inspect. First, the users run instrumented programs to collect call profiles. After each run, the user sends the profile to the analyst. Next, the analyst performs data clustering of all profiles using a distance metric to group similar profiles together. Finally, the analyst examines one or several profiles from each cluster to determine whether the entire cluster corresponds to correct or failed executions.

Unlike Magpie and our work, Dickinson et al. do not support analysis of distributed systems. For single-process applications however, their approach is similar to ours. We also construct function profiles and use a distance metric to estimate similarity between them. This technique ignores insignificant variations in raw traces. Similar to Magpie however, the approach of Dickinson et al. is unsupervised and may not be able to distinguish between true anomalies and unusual but normal behaviors.

Finally, there is an extensive body of work on anomaly detection for identifying host intrusions. Intrusion detection techniques most similar to our approach are dynamic model-based detectors [40, 41, 62, 113]. These techniques monitor the execution of the system obtaining sequences of events, typically system calls. The observed sequences are compared against the model of normal behavior and the alarm is raised if the observed behavior does not satisfy the model. To construct the model, these techniques introduce an earlier *training phase* where the system calls are recorded under known-normal conditions.

Although this approach analyzes coarser-grained system call traces, it could also be applied to our function-level data. Such an algorithm would be similar to our one-class ranking technique that uses examples of known-normal behavior for finding anomalies. However, we use known-normal examples only to reduce the number of false positives. We do not require the set of known-normal examples to be exhaustive and even can operate without any prior examples, using only trial traces to identify outliers. Obtaining an exhaustive set of known-normal behaviors is difficult for system call traces and even more challenging for finer-grained function-level data.

2.4.2 Finding the Cause of Anomalies

Once an anomalous trace has been identified, we need to find the cause of the anomaly. Our approach helps the analyst by identifying the most visible symptom of a problem. For example, if an application enters an infinite loop, we will point the analyst to the function containing that loop. The analyst would then use the collected trace or the application's source code to determine why that function was executed. Here, we compare this technique to approaches proposed in previous work [12, 24, 26, 32, 136].

Once Magpie locates an anomalous trace (request string), it allows the analyst to identify an event responsible for the anomaly as follows. First, it builds a probabilistic state machine that accepts the collection of request strings. Then, Magpie processes the anomalous request string with the machine and identifies all transitions with sufficiently low probability. Such transitions correspond to events where the processing of the anomalous request started deviating from the norm. These events are then marked as the root causes of the anomaly and reported to the analyst. Note that this approach is similar to the PCFG technique of Pinpoint, but Magpie applies it to root cause analysis rather than anomaly detection.

This approach could also be applied to our function-level traces. It may be able to identify the first function call that was present in the anomalous trace, but absent from the normal ones. This first function call may point the analyst to an early symptom of the problem. Such a symptom is a more direct indication of a problem than the most visible symptom identified with our approach.

In our experience however, function-level traces in distributed systems are highly variable and the first difference is often caused by noise in the trace or by minor variations in program input for different runs. Magpie could disregard such differences by collecting enough training data. However, if an approach aims to detect anomalies by using cross-run repetitiveness in behavior, it would need to collect training data for many workloads and program inputs, which is often impossible. Similarly, an approach that uses cross-host repetitiveness to detect anomalies in a single run of a distributed system can only operate with a fixed number of traces.

Pinpoint uses two different techniques for root-cause identification: clustering and decision trees. Both techniques look for correlations between the presence of a component in a request and the outcome of the request, as determined by their anomaly detection techniques. For example, if a component was present in traces for all failed requests and absent from traces for successful requests, it is marked as the cause of the problem and reported to the analyst.

When applied to function-level traces, the technique of Pinpoint can diagnose problems that are manifested by a difference in function coverage. For example, if function *foo* causes memory corruption whenever called and leads to a crash, this approach will notice that *foo* is only called in failed runs and report it to the analyst. A similar approach is used in *dynamic program dicing* where the dynamic backward slice for a variable with an incorrect value (a set of program statements affecting the variable) is compared to the slice for a variable with a correct value, e.g., the slice for the same variable in a successful run [27, 74]. The cause of the bug is likely to be among the program statements in the set-difference between these slices.

Identifying the actual cause of the problem in the set of coverage differences requires manual effort. Some differences correspond to noise in the trace or to normal variations in program input or workload. While statement-level analysis may provide for more precise problem localization, it requires more substantial manual effort than coarser-grained function-level analysis. In our experience, even the number of differences in function-level coverage is often prohibitively large for manual examination. To address this limitation, we have developed techniques for eliminating differences in function-level coverage that are effects of an earlier difference. We also rank the remaining differences to estimate their importance to the analyst. Whether these techniques can be applied at the statement level remains to be seen.

The coverage-based approach still requires some manual effort to analyze the remaining differences. Also, it may not detect some problem causes with no difference in coverage. As an alternative approach, we present the analyst with the most visible symptom of the problem (e.g., the last function in the trace for fail-stop problems or a function where an unusual amount of time is spent for non-fail-stop problems). This information is often sufficient for correct diagnosis.

The Cooperative Bug Isolation Project (CBI) uses another technique that correlates observed events with failures [67, 68]. CBI instruments an application to collect the values of various types of predicates for each run, such as whether a conditional branch was taken, whether a function returned a positive, negative, or zero value, and whether one scalar variable was greater than another. Further, each run is labeled as failed or successful, e.g., depending on whether the application crashed or not.

Finally, CBI uses statistical techniques to analyze the collected data from numerous runs and identify predicates that are highly correlated with failure. Such predicates allow the analyst to focus attention on specific parts of the code. CBI ranks predicates by their likelihood of predicting a bug. After locating the predicate with the highest rank (the best predictor for the first bug), CBI eliminates all runs where this predicate was true, and repeats the analysis, identifying the best predictor for the second bug, and so on.

There are three key differences between CBI and our approach. First, CBI requires all runs to be labeled as failed or successful. Similar to the fault detector-based technique of Pinpoint, this approach may not detect failures with previously-unknown symptoms. In contrast, we detect anomalies from the collected trace data.

Second, we collect the complete control-flow trace for an execution. CBI only collects sampled fragments of such a trace. Each CBI fragment contains more details than what we collect and thus can be used for more precise diagnosis. However, all fragments only cover a small fraction of an execution (typically, $^1/_{100}$ to $^1/_{1000000}$) to minimize the overhead of detailed tracing. As a result, the approach of CBI is effective at locating bugs in software with a large number of installations as it requires a few hundred reports. Our techniques proved to work well even with 5–10 reports.

Finally, by introducing a set of predicates that determine whether each function was present in the trace or not, the CBI analyses could be applied to our scenario. However, CBI's report ranking and elimination strategies may not be effective for our complete function-level coverage data. In our experiments, many coverage differences were perfectly correlated with the failure and would be given the same highest rank by the CBI's ranking approach. Most of these differences corresponded to normal variations in program input and workload and would not point to the problem cause. Furthermore, the CBI's run-elimination step would not be effective on our data as perfect correlations would eliminate all failed runs in the first step.

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

Yuan et al. operate on system call traces that have coarser granularity than our function traces. In our experience, once a problem happens, the behavior of an application changes significantly, so that even system call traces may appear anomalous. This property is often used in anomaly-based intrusion detection (e.g., see [41, 129]).

Although useful for detecting anomalies in system applications, it remains to be seen whether system call traces can accurately represent the behavior of other applications, such as parallel numeric codes. Moreover, system call traces may not be sufficient for finding many performance anomalies and their causes. In general, if the cause of a problem is in a function that does not make system calls, finding its location from system call traces may not be possible.

The approach of Yuan et al. would also work on our function-level traces. However, this technique targets the root causes of known problems and would be unable to diagnose new failures. Furthermore, it requires manual effort from the analyst to classify traces for previous failures.

Another supervised technique has been used by Cohen et al. [32] to correlate aggregate performance metrics, such as system CPU and disk utilization, with violations of SLOs (Service Level Objectives) in e-commerce environments. Metrics that had unusual values when the SLO was violated are reported to the analyst as the cause of the problem. Unlike the work of Yuan et al. and similar to Pinpoint, the violations are detected using an external fault detector, without analysts' help.

There are two key differences between our approach and that of Cohen et al. First, Cohen et al. report aggregate high-level metrics as problem causes. This information may be useful for system administrators to upgrade the system and thus alleviate the symptoms, however, we narrow problems to individual functions in the code, making it possible for developers to eliminate the root cause of the problem. Second, Cohen et al. only look for violations of specific performance goals. Our approach can also work for silent problems and problems with unknown symptoms.

2.4.3 Summary of Data Analysis Approaches

Magpie and Pinpoint are most similar to our approach to problem diagnostics. Both projects operate on traces collected in distributed systems and analyze such traces in two steps: they find an anomalous trace and then find an event in the trace responsible for the anomaly. Despite these similarities, Magpie and Pinpoint would not be accurate on our data. In our experience, analysis of highly-variable function-level traces requires techniques that disregard minor differences in the traces. A diagnostic approach should also be able to incorporate prior knowledge into analysis to avoid reporting normal but unusual behaviors as anomalies.

Our approach for finding the anomalous execution is most similar to that of Dickinson et al. We also summarize traces as profiles and then find the most unusual profile. Unlike Dickinson et al. however, our approach can make use of labeled examples of prior executions, if available. This technique allows us to reduce the number of false positives and identify anomalies with higher accuracy.

Once the anomalous execution is identified, several previous techniques can be used for finding the root cause of the anomaly. Pinpoint, CBI, and dicing may be most suitable for analyzing function-level traces. They operate on summary (coverage) data and are therefore less sensitive to noise in the trace. In our experience however, even the coverage-based approach generates a large number of differences when applied to our traces. Most of these differences do not correspond to real problem causes. Our techniques reduce the required manual effort by eliminating some differences in coverage that correspond to effects of an earlier difference.

The coverage-based approach still requires some manual effort to analyze the remaining differences. Also, it may not detect some problem causes that produce no difference in coverage. Therefore, we begin by presenting the analyst with the most visible symptom of the problem. For example, we can identify a function where an unusual amount of time is spent in the anomalous run. If the most visible symptom is not sufficient for correct diagnosis, we perform the coverage analysis.

Chapter 3

Propagation and Trace Data Collection

We locate bugs and performance problems by comparing control-flow traces for normal and anomalous activities in a system. This chapter describes self-propelled instrumentation, a new approach that we use for collecting such traces. Identification of the different activities within the traces and their automated comparison are described in Chapters 4 and 5 respectively.

The key concept in self-propelled instrumentation is run-time propagation of instrumentation ahead of the flow of control within a component (e.g., a process) and across component boundaries. This mechanism allows us to construct an autonomous agent for self-directed exploration of the system. The agent begins its analysis at a user-specified *activation event* that corresponds to the start of a semantic activity, such as an HTTP request and follows the effects of the event through the system, looking for anomalies in the observed control flow. The main properties of this mechanism are its ability to be deployed on demand in a distributed environment, collect detailed run-time information, and introduce little execution overhead. Previous instrumentation techniques do not satisfy at least one of these properties.

In a typical usage scenario, the agent attaches to a running application and stays dormant, waiting for the activation event. When the event happens, the agent starts propagating through the system, carried by the flow of execution. That is, the agent takes control at a point of interest in the application's code, executes user-specified *payload code* (e.g., collects trace data), decides when it wants to receive control next (e.g., on the next function call in the same process or on a message-arrival event in another process), and returns control to the application. When the user decides to stop the propagation procedure, the agent can be deactivated.

Below, we outline our techniques for activation, propagation, payload execution, and deactivation. The design and implementation of these techniques is discussed in Sections 3.1 through 3.3. We evaluate our techniques on a prototype agent called *spTracer*.

Activation and Deactivation. One of our key contributions is in developing a collection of techniques for rapid, on-demand analysis of production systems. The first step in deploying the agent on-demand is activation, i.e., injecting the agent code into the system and letting the user specify an event for the agent to start propagating. Standard OS mechanisms and techniques previously proposed by other researchers are sufficient for agent injection. For tracing applications, we compile the agent as a shared library and cause the application to load the library at startup using the *ld.so* dynamic loader [71] or at run time via Hijack [137]. For tracing the kernel, we compile the agent as a kernel module and load the module at run time via *insmod* [71].

After injection, the agent waits for an activation event and then starts propagating through the code. To debug the real-world problems described in Chapter 6, we developed several activation techniques. Our prototype agent can activate on process startup (to control the application from the beginning), on a function call (when a specified function in the application is executed, similar to a traditional breakpoint), on a timer event (at a certain time), on a key press (when a certain key combination is pressed), and on a message-passing event (when the process accepts a new network connection or receives data from an already-established connection). To capture such events, we used standard OS interfaces, developed OS kernel extensions, and instrumented select locations in applications' code, as described in Section 3.1. Designing a more versatile framework for specifying other types of activation events and developing OS capabilities necessary for capturing such events remains future work.

To start analyzing collected traces, we rely on the user to specify one or more *deactivation events*. On detecting a deactivation event, our helper tool saves collected traces to disk. To detect deactivation events, we use mechanisms similar to those used for activation. We can save the trace after the traced process terminates (e.g., for debugging application crashes) or on a signal from the user (e.g., for debugging application freezes and other non-fail-stop failures). For distributed applications, the host that detected a deactivation event notifies all other hosts in the system and they save their traces to local disks. Later, we collect all the traces at a central location for analysis.

Propagation. To propagate, the agent uses *in-place instrumentation*, a kind of dynamic instrumentation performed within the same address space. The agent examines the current location in the code where the application executes, finds the next point of interest ahead of the flow of control (we call it an *interception point*), modifies the code to insert a call to itself at that point, and allows the application to continue. When the application hits the instrumented point, the agent will receive control and propel itself to the next point.

A similar mechanism has been used by previous approaches for dynamic binary translation [17, 28, 73, 75, 89]. We improve upon this work in two main directions. First, our technique has substantially lower start-up overhead, as shown in Section 3.2.4. This property is crucial for on-demand deployment in production environments. Second, self-propelled instrumentation has the ability to cross component boundaries, propagating from one process into another or from a process into the kernel. This property is crucial for on-demand data collection in distributed systems.

To collect system-wide traces, our agent dynamically propagates from the process receiving the start-request event into other processes involved in handling the request when processes communicate. For example, consider a traced process that writes a message to a socket connected to another process (possibly, on another host) that is currently not traced. Our agent intercepts control of the first process before the *send* system call, identifies the address of the peer, requests our daemon on the remote end to inject a copy of the agent into the peer, and continues the *send* system call in the sender. When the message is received at the remote end, the newly-injected copy of the agent will start tracing the control flow of the receiver process from the *recv* system call.

To investigate problems that occur in the kernel code, we augment control-flow traces of an application with control-flow traces of the kernel. When a traced application performs a system call or is preempted by a hardware interrupt, the agent walks into the kernel and starts propagating through the kernel code. Note that since the agent shares the address space with the component it is instrumenting, we use the same mechanism and the same code base for propagation on both sides of the kernel boundary. The implementation of previous instrumentation techniques had to use different mechanisms for user and kernel spaces [18, 124].

Payload execution. For extensibility, the agent executes user-specified payload code at each interception point. In its simplest form, the payload can passively record and analyze execution information. This approach can be used for performance analysis and automated bug finding. The payload can also react to observed information, potentially altering the execution. Two common applications of such an active approach are security analysis (e.g., monitor a process and prevent it from executing malicious actions) and dynamic optimization.

Our current implementation uses the passive monitoring approach to record the time stamp of reaching each interception point along with the point identifier. We intercept the flow of control at functions' entries and exits, collecting an in-memory function-level trace of execution. We also record information about communication operations to construct the PDG for an execution, as defined in Section 1.3.

Our key contributions at the trace collection stage are two-fold. First, unlike previous approaches, information collected by our tracer is sufficient for constructing the PDG without the complete history of communications. This property is crucial for on-demand diagnosis where some message-passing operations may have occurred before tracing is deployed or are in-flight when the tracing is deployed. The unavailability of trace records for such operations complicates matching of send and receive operations on a channel. We address this problem with a technique that allows the tracers at the end points of a channel to synchronize their view of the channel.

Second, we retain the in-memory trace after a sudden application crash. This property achieves trace durability without the expense of writing trace records to disk.

To provide this property, we keep the trace data in a shared-memory segment and map the segment both into the application process and a helper tool. If the application process dies, the segment is left around and the helper tool can save its contents to disk. This technique does not protect the data from the crash of the operating system or a hardware failure, but it is useful for surviving application crashes, which are typically more frequent than whole-node crashes. Note that other tools, such as Paradyn [82] and OCM [134], have used a shared-memory segment for fast accesses to performance data from another process. However, to our knowledge, this technique has not been previously used for achieving trace durability.

3.1 Activation

Our diagnostic approach relies on the user to specify an *activation event*, i.e., an event that causes the agent to start collecting trace data. For our analyses to be effective, there should exist a failed execution where this event happens before the fault. By recording all events that are ordered after the activation event by the happened-before relation [61], we can then capture the occurrence of the fault in the trace. Subsequently, our analysis techniques will attempt to detect that a failure happened and locate the fault within the recorded trace.

We often can use an event that triggers a problem as an activation event. For example, many bugs in GUI applications occur in response to a user-interface event such as a mouse click or keyboard input. If we start tracing after such an event, we can capture the cause of the problem. Many bugs in servers occur in response to an external connection event. Some bugs occur when an application is run with a particular input, so the application startup can serve as an activation event. Note that usually, the closer is the activation event to the actual fault, the easier it is to identify the fault within the trace.

Here, we describe our techniques for injecting the agent into the system and capturing several types of activation events. The current mechanisms are motivated by our experimental studies discussed in Chapter 6, though we plan to support a wide variety of events in the future. We inject the spTracer library into the application's address space using two different mechanisms. To capture control flow from the moment the application starts, we can load the library at startup via the *LD_PRELOAD* mechanism of the *ld.so* dynamic loader [71]. This mechanism is also useful in multi-process tracing, when a traced process forks and the child issues the *exec* system call to load another application. We use *LD_PRELOAD* to inject the library into the newly-loaded application and start tracing both applications.

To support tracing of long-running applications that were started before the need for tracing arose, we use the third-party Hijack mechanism to cause an application to load our shared library at run time [137]. This mechanism is implemented in an external tool process that attaches to an application via the standard *ptrace* debugger interface [71]. The tool writes a small fragment of code for loading the library into the application's address space and causes the application to execute it. The tool then detaches from the application and the newly-loaded agent continues further instrumentation autonomously. Hijack is also useful in multi-process tracing, when a traced process sends a message to an untraced process. It allows the agent in the sender to inject itself into the receiver and start tracing both processes.

After the insertion, the agent briefly receives control to initialize its state and otherwise lays dormant waiting for one of the following activation events. First, it can initiate propagation when a particular function in the application is executed. This method assumes that the user is familiar with internals of the application to specify the name of such function. When spTracer is injected into the application, it instruments all call sites in the chosen function with jumps to the agent library and lets the application continue. When the instrumentation is reached, the agent receives control and starts propagating.

Second, spTracer can initiate propagation immediately after injection. This technique allows us to collect complete control-flow traces from the moment the application starts. To provide this activation method, we specify *main* as the starting function and apply the above technique.

Third, the function-based approach is also useful for starting the propagation on interaction with another process. For example, to obtain a trace of an HTTP request processing in a Web server, we can instrument the *accept* system call, thus catching the moment when a client connects to the server, and start following the application's flow of control after *accept* returns. Similarly, we can instrument the *recv* system call to start propagation when a message is received on an already-opened connection. In addition to activation upon communication, we also use this mechanism for system-wide propagation across communication channels.

Fourth, we can initiate tracing when a user-specified time interval elapses since the start of an application. This mechanism is useful for analyzing problems that happen at a fixed moment and do not depend on input from the user. To receive control on timer expiration, the tracer library uses the *alarm* system call to arrange for a *SIGALRM* signal to be delivered to the application in the specified number of seconds. The tracer library catches the signal, determines where the application was executing before the signal, instruments that function and the agent starts propagating from that point as usual.

Finally, we can activate spTracer when the user hits a particular key combination. To be notified of keyboard events, we implemented a Linux kernel module that intercepts keyboard interrupts (*IRQ1*). When the application starts, our library issues an *ioctl* system call to register the application with the kernel module and enable this snooping ability. When the module receives a keyboard interrupt, it checks if the key combination is pressed and sends a *SIGPROF* signal to the registered application to initiate tracing. The library will then catch the signal and process it similarly to *SIGALRM* events.

3.1.1 Stack Walking

The activation methods discussed above have a common shortcoming of tracing only a subgraph of the application's call graph. Indeed, they start tracing at a particular function and propagate the instrumentation into its callees and further down the call graph. However, the callers of the initial function will not be instrumented and the tracing may stop after the initial function returns. If tracing starts in a function which contains no calls, we will not see any trace records.

To address this problem, we augmented the starting mechanisms with a stack walking capability. When tracing starts, we walk the stack and instrument not only the current function, but also all functions on the stack up to *main*. This technique allows us to continue tracing after the current function returns and obtain complete traces. However, stack walking techniques are not fully robust. Optimizing compilers often emit functions with no stack frames or with stack frames that do not use the frame pointer. Walking the stack past these functions or even detecting such cases is unreliable and may crash the application.

We encountered this stack walk problem in the Wine study discussed in Chapter 6. For that study, our interim solution was to start the propagation process on application startup, but generate no trace records until an activation event. When the activation event happens, all functions currently on the call stack are already instrumented. Therefore, the agent is able to start tracing without walking the stack. A disadvantage of this strategy is that it requires preloading of the agent on application startup and may not work if the agent can only be loaded at run time.

To support on-demand agent injection even in optimized applications, we propose a new scheme that is yet to be implemented. When tracing starts, we will instrument not only call sites in the current function, but also all its return instructions. When the return-point instrumentation is reached, the stack pointer points to a location containing the return address. Therefore, the tracer will be able to find the caller of the current function and propagate into it. In turn, it will also instrument the caller's return points to advance to its caller when necessary. Notice that the return-point instrumentation is needed only to walk out of functions that were on the stack when we started tracing. It can be removed when reached.

3.2 Intra-process Propagation

Once active, our agent starts propagating through the code of the system. In this section, we describe the mechanisms used for propagating within a single process and obtaining the local control-flow trace. We discuss the general technique for self-propelled instrumentation, explain how to instrument indirect calls and other short instructions, describe our offline code analysis framework that allows us to lower the startup overhead, and finally, present the overhead study. Section 3.3 describes our techniques for propagating across the process and kernel boundaries.

3.2.1 Propagation through the Code

To obtain the control-flow trace, the agent intercepts the execution of an application or the kernel at control-transfer instructions (function calls in the current prototype), identifies where the control flow will be transferred next (finds the function to be invoked), and instruments the next point of interest (all call sites in the identified function). This technique can also be used for obtaining the basic-block trace of an execution by treating all control-transfer instructions, including conditional branches, as interception points.

Two techniques distinguish our intra-process propagation approach from previous work on dynamic binary translation [17, 28, 73, 75, 89]: in-place instrumentation and offline code analysis. Their combination allows us to lower the overhead of inserting instrumentation. With in-place instrumentation, we modify the original code of the application and avoid the expense of generating its copy in the fragment cache. Offline code analysis allows us to avoid expensive run-time code discovery. As a result, we introduce little perturbation even to infrequently executed code.

As the application executes, our prototype propagates through its call graph, moving from a function into its callees, and into their callees. Suppose that spTracer received control before entering Function A, and Function A contains a call to Function B. To intercept the call to B, spTracer replaces the "*call B*" instruction with a jump to a run time-generated patch area. The patch area contains calls to tracing routines and the relocated call to B: "*call tracerEntry(B); call B; call tracerExit(B)*". When the *tracerEntry* routine is reached, it emits a trace record for B's entry and propagates the instrumentation into B. (It uses pre-computed code analysis results to locate all call sites in B, generates patch areas for each site, and replaces the call instructions with jumps to the corresponding patch areas). When the *tracerExit* routine is reached, it simply emits a trace record for B's exit.

In a previous implementation of our tracer, we generated no per-site patches and simply replaced the "*call B*" instruction with "*call tracerEntry*". When *tracerEntry* was reached, it used the return address to look up the original destination of the call (Function B) in a hash table and jumped to B, returning control to the application. To intercept the flow of control when B exits, *tracerEntry* modified B's return address on the stack to point to *tracerExit*. Although this scheme is more memory-efficient (no need for per-site patches), it had substantially higher run-time overhead for two reasons. First, performing a hash lookup on every function call is expensive. Second, modifying B's return address on the stack resulted in mis-predicting the destination of the corresponding return instruction by the CPU. This penalty can be significant on modern processors [75].

3.2.2 Instrumenting Indirect Calls

Intercepting regular call instructions is straightforward. On platforms with fixed-size instructions, intercepting indirect calls (e.g., calls through function pointers and C++ virtual functions) can be equally simple: we could still overwrite an indirect call with an unconditional branch to a patch area. However, our target platform, x86, has a variable-length instruction set. On x86, an indirect call instruction can be as small as two bytes, so we cannot replace it with the long five-byte jump. A similar problem occurs in basic block-level tracing when instrumenting short conditional branches.

To address this problem, we use existing code-relocation techniques and further improve them to be applicable in more scenarios. Our relocation approach is most similar to that of Dyninst API [18]: we copy basic blocks that contain short indirect call instructions to the heap. First, we identify the basic block that contains the indirect call instruction. Second, we copy the block contents to a heap location, adjusting position-dependent instructions appropriately. When copying the indirect call, we also insert calls to *tracerEntry* and *tracerExit* around it. Finally, we write a jump to the relocated block at the beginning of the original block, such that further execution flows through the relocated block.

Note that block relocation is not always possible. This technique needs to overwrite the entry point of a block with a jump to the relocated version. Blocks longer than four bytes can therefore be relocated with a single long jump. Blocks longer than one byte often can be relocated with a short jump to a springboard location [124], which contains the long jump to the relocated version. Finally, relocating one-byte blocks is not possible with our technique. Fortunately, we do not need to instrument such blocks as they cannot contain call instructions. For other purposes, we used trap-based instrumentation and whole-function relocation in the past.

Finding space for the springboard to instrument 2–4-byte basic blocks is usually possible by displacing a long block nearby (10 bytes or longer, enough to write two long jumps). Dyninst uses a similar approach by using the function's entry as a springboard if there are no branch targets within the first 10 bytes of the entry. Our technique increases the applicability of this approach by using any long nearby basic block as a springboard. We note that Dyninst can also be augmented with this functionality to instrument locations that are not reachable from the entry point with a short jump. At present, Dyninst performs more complex whole-function relocation in such scenarios.

Block relocation is also similar to building a copy of the code as performed by DynamoRIO [17], DIOTA [75], PIN [73], and Valgrind [89]. Unlike these systems, we do not relocate the entire code of an application. Only blocks that contain short indirect calls are moved to the heap. Furthermore, we perform most of the analysis and code generation offline, before the application starts running. These factors allow us to lower the run-time overhead of generating relocated code.

3.2.3 Code Analysis

To perform instrumentation as described above, the tracer needs to parse the binary code of an application and the kernel. In particular, it must be able to find all call sites in any function being instrumented and determine their destinations (i.e., it needs to know the call graph of the application). When instrumenting indirect calls, it relocates the corresponding basic blocks of a function to the heap. To support this functionality, it also needs to obtain the control flow graph for any function with indirect calls.

Our prototype obtains both the call graph and the control flow graphs before the application starts running. This technique is crucial for on-demand deployment of the agent in production systems as it allows us to lower the activation overhead considerably. We leverage the code analysis infrastructure of Dyninst API [18] to retrieve and save the analysis results to disk.

To support tracing of different applications, we maintain a system-wide repository that keeps results of code analysis for all applications of interest accessible by the name of a binary. When spTracer is injected into an application, it determines the names of all loaded modules using information from the */proc* filesystem [71]. For each module, it then retrieves the code analysis results from the repository and begins using them when the user enables tracing.

3.2.4 Overhead Study

To estimate the run-time overhead of self-propelled instrumentation, we applied it to two applications: a popular multimedia player called *MPlayer* (Version 1.0pre2) [88] and a custom microbenchmark. The former test allowed us to study the impact of our instrumentation on a compute-intensive and time-sensitive real-world application. The overhead on I/O-bound applications is likely to be lower than that observed for *MPlayer*. The latter test allowed us to estimate the upper bound of our overhead by instrumenting an application that made calls to an empty function in a tight loop. We ran these experiments on an IBM Thinkpad T41 laptop with a Pentium-M 1.6 GHz CPU and 1 GB of RAM under RedHat Linux 9.

We used *MPlayer* to display a short DVD movie clip. This workload is a good test of our overhead as *MPlayer* makes many short function calls when decoding the MPEG2 stream of the clip. To measure the slowdown, we modified the *MPlayer* source code to count cycles spent processing each video frame. Note that to show frames at a regular interval ($1/25^{th}$ of a second for our video clip), *MPlayer* inserts a calculated delay after decoding a frame, but before showing it on the screen. The adjustable nature of the delay may mask the overhead that we are measuring. Therefore, we excluded the time spent sleeping from the total time of processing a frame.

We compared self-propelled instrumentation to two other approaches: dynamic binary translation with *DIOTA* (Version 0.9) [75] and load-time total instrumentation. To perform total instrumentation, we used the self-propelled framework, but inserted trace statements into all functions at startup, before starting measurements, and disabled propagation of instrumentation. With total instrumentation and self-propelled instrumentation, we called empty trace routines on entry and exit from every function in the application. Since *DIOTA* did not support function-level tracing, we measured the lower bound of its overhead — the overhead of following the control flow without executing any trace calls.

To model trace collection for a short interval during execution, we measured the overhead at three stages: warm-up (the first frame after enabling tracing), steady-state (after being enabled for a while), and inactive (after disabling or before enabling tracing). To measure the warm-up overhead of self-propelled instrumentation, we enabled tracing for a randomly-chosen frame (here, Frame 100) and measured the time to process it. Since *DIOTA* did not support run-time activation, we measured its warm-up overhead on start-up (Frame 1). Note however, that Frame 1 was substantially different from other frames in the stream as *MPlayer* performed additional I/O activities for Frame 1. These activities consumed substantial time, but executed little code, thus lowering the overhead of all instrumentation methods. Therefore, the warm-up overhead of *DIOTA* on Frame 100 is likely to be higher than that measured for Frame 1.

The overhead for all approaches proved to depend on the compiler version used for building *MPlayer*. In particular, the overhead varied greatly depending on how many functions a compiler decided to inline. Tables 3.1a and 3.1b present the highest

	Frame 100 (Warm-up)	Average (Steady-state)	Inactive
Total instrumentation	29%	29%	29%
DIOTA	95%+	58%	58%
Self-propelled instrumentation	39%	34%	0%

a: *Compiled with gcc-3.2.2*

	Frame 100 (Warm-up)	Average (Steady-state)	Inactive
Total instrumentation	10%	10%	10%
DIOTA	89%+	23%	23%
Self-propelled instrumentation	20%	15%	0%

b: *Compiled with gcc-2.95.3*

Table 3.1: *Instrumentation-induced increase in time MPlayer takes to process a video frame. Warm-up is the overhead for the first frame after enabling tracing, steady-state is the overhead for an average frame, and inactive is the overhead after disabling or before enabling tracing.*

and the lowest overhead results respectively. Both results were obtained for *MPlayer* binaries compiled with the default optimization options chosen by the *MPlayer* configuration process, but different versions of the gcc compiler: 3.2.2 and 2.95.3. Further discussion applies to both runs, as all instrumentation methods showed similar relative performance.

Total instrumentation demonstrated reasonable performance at the warm-up and steady-state stages. However, total instrumentation cannot be rapidly enabled on the fly – even the same-address space instrumentation of our framework required approximately 45 milliseconds to instrument the more than 11,000 functions in *MPlayer* and supporting libraries. This time exceeded the 40-millisecond interval allotted to displaying a video frame. For larger applications, the delay would be even more pronounced. An alternative approach is to pre-instrument every function on startup, but it will force the user to endure the steady-state overhead for the entire running time of the system. These observations show the importance of rapid on-demand activation of tracing for analysis of production systems.

DIOTA also did not support on-demand activation, although it could be implemented. Therefore, its non-zero overhead at the inactive stage is less problematic. More importantly, *DIOTA* had significant warm-up overhead, resulting in dropped frames on startup even in the lower overhead run. We believe that building the code cache on the fly may be unacceptable for analysis of time-sensitive applications. It may introduce intermittent delays upon run-time activation of the tracer and also whenever the flow of control reaches previously-unseen code. In contrast, the warm-up overhead of self-propelled instrumentation was significantly lower than that of *DIOTA*, since we performed code analysis in advance and did not need to populate the code cache. We also imposed no overhead at the inactive stage, as tracing could be rapidly activated on demand.

To measure the upper bound of steady-state overhead of self-propelled instrumentation, we traced a microbenchmark that was invoking an empty function in a tight loop. With self-propelled instrumentation, the benchmark ran 500% slower. Such pathological cases are rare in real applications, but they need to be addressed. The dynamic nature of self-propelled instrumentation makes it easy to design an adaptive scheme that would detect short frequently-executed functions and avoid tracing them to lower the overhead. The efficacy of such a scheme is yet to be studied.

3.3 Inter-process Propagation

Our diagnostic approach requires the capability to construct a causally-ordered path between the start and end event. If the path never crosses the component boundary (e.g., all relevant events happen in the same process), problem diagnosis can be achieved with local tracing, as described previously. When the path spans multiple components (threads or processes), possibly running on different hosts, this approach needs to be extended to collect traces in all components that may be affected by the start event. By tracing all components along the path, we can obtain enough data for problem diagnosis.

Our key contribution is the design of an approach that can be deployed on-demand in a distributed system and collect events on an execution path between the start and the end event, even if the path spans multiple processes and hosts. We discover such paths automatically by following the natural flow of execution in the system. We follow the flow of control within a process where the start event happened, and carry the tracing over into another process when the two processes communicate. A similar approach can also be used for inter-thread communications. The key steps in propagating from one component to another are:

1. Detect the initiation of communication at the local site.

2. Identify the name of the destination component.

3. Inject the agent into the destination component (possibly, on a remote host).

4. Detect the receipt of communication by the destination component.

5. Start following the flow of control in the destination component.

We begin by presenting our techniques for performing these steps for the foundational case of inter-process communication via TCP sockets. In Sections 3.3.2 through 3.3.4, we generalize these techniques to other types of communication. Note that for some of the above steps, we use TCP-specific methods that have no analogs for other communication mechanisms. In such situations, we propose alternative ways of accomplishing the same goals, where possible. We also identify potential improvements to the operating system interfaces that could make these steps feasible or more efficient for some communication mechanisms.

3.3.1 Propagation through TCP Socket Communications

To enable the agent to propagate across host boundaries, we run daemon processes, called *spDaemons*, on all hosts in the system. These daemons can be started at system boot time, or they can be started by the agent on-demand, if nodes in the system support remote operations such as *SSH* (Secure Shell) [13]. Suppose, process P attempts to send data to process Q over a TCP socket. Figure 3.1 shows how the agent in process P propagates into process Q. First, the agent library, *agent.so*, instruments the *send* and *write* routines in P. When this instrumentation is reached, we determine whether the accessed file descriptor corresponds to a TCP socket. If so, we intercept control of the process before it sends the message.

Second, we determine the name of the peer. The name uniquely identifying a process in our system is a tuple $\langle hostid, pid \rangle$, where *hostid* is the IP address of the host where the process runs and *pid* is the identifier of the process on that host. We will refer to this tuple as UPL (Uniform Process Locator). The UPL for process Q in Figure 3.1 is $\langle ipB, pidQ \rangle$. Note that while the sender can find the $\langle hostid, portid \rangle$

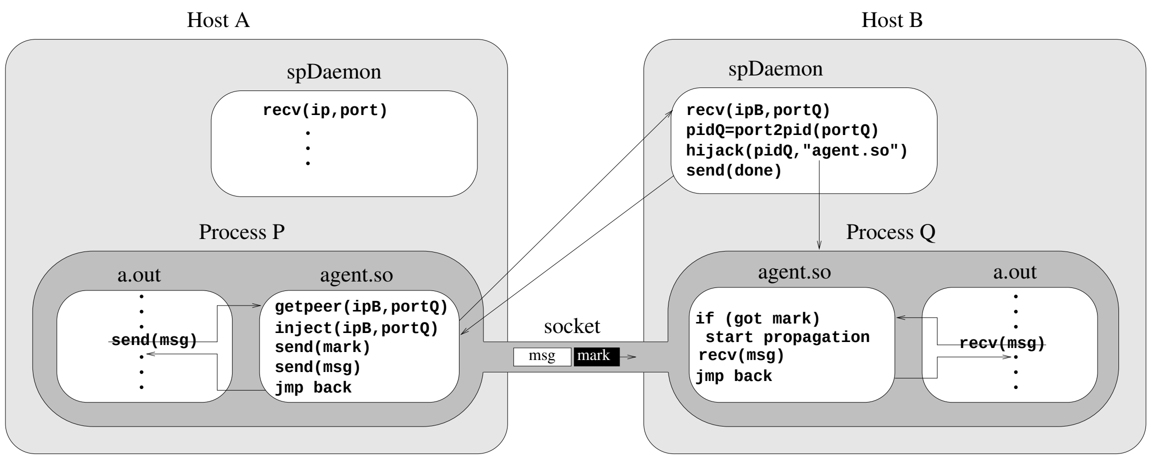


Figure 3.1: *Propagation of the agent from process P on host A to process Q on host B. P attempts to send a message to Q over a TCP socket. Our* agent.so *library in P intercepts control before the send, identifies the IP address (ipB) and the port number (portQ) of the socket endpoint, contacts our spDaemon process on host B to inject the agent into a process that has portQ open, marks the message being sent using the TCP OOB mechanism, and allows the send to proceed. When injected in Q,* agent.so *waits until this marked message is received and starts following Q's flow of control.*

tuple for the peer using *getpeername* [71], where *portid* is a remote port number, obtaining the remote process identifier requires an additional step.

To find the process identifier *pidQ* given the port number of the open socket *portQ*, we use a technique similar to that used by the *netstat* utility [71]. The agent on host A contacts *spDaemon* on host B, and this daemon uses information available in the */proc/net/tcp* file to map the port number to the inode number uniquely identifying the socket. *spDaemon* then scans the */proc* directory tree to locate a process that has a socket with that inode number open. In Figure 3.1, this functionality is encapsulated in *spDaemon*'s function *port2pid*.

Third, *spDaemon* injects a copy of the agent into the identified process using the Hijack mechanism [137]. Fourth, when the injection procedure completes, *spDaemon* notifies the agent on the sender host. The agent then uses the TCP OOB (Out-of-band) mechanism to mark a byte in the channel to synchronize the state of channel endpoints, as we describe below, and lets the application proceed with the original send. The injected agent on the receiver side has already instrumented the entry points for *recv* and *read* system calls to identify the moment when the message arrives.

Finally, when the *recv* or *read* instrumentation is executed, it checks whether the file descriptor corresponds to the original socket and whether the OOB mark has arrived. If so, the agent instruments all functions on the stack, from *main* to the *recv* or *read* and starts the propagation process on the receiver side.

3.3.2 Detecting the Initiation of Communication

To detect that one component attempts to communicate with another, we instrument the communication routines. This technique works well for message passing, where the set of communication routines is typically well-defined and application-independent. Knowing the names and the semantics of such application-independent routines allows us to use the same tracing mechanism for many applications. Although this technique can also be applied to some applications that communicate over shared memory using a well-defined interface, designing a mechanism for intercepting shared-memory interactions in the general case remains future work.

Even for message-passing communications, relevant routines are sometimes overloaded to perform activities not always related to communications. Ignoring such activities is usually possible by examining the function call arguments. For example, the *write* system call is used not only for sending data over a socket connection, but also for writing to local files. To distinguish between these uses, we examine the file descriptor argument to *write* and determine whether it corresponds to a socket or a regular file. Similarly, we use this mechanism to distinguish between different types of communication: pipes, TCP sockets, and UDP sockets.

We can also apply this approach to shared-memory communications if the user specifies the names and semantics of all routines where communications occur. For example, the user can indicate that *enqueue* and *dequeue* are communication operations and operation y = *dequeue()* corresponds to an earlier *enqueue(x)* if and only if y is equal to x. When a traced thread executes the *enqueue* operation and later another (untraced) thread executes the matching *dequeue* operation, our agent may start tracing the second thread when *dequeue* completes. Note that unless *enqueue* and *dequeue* are provided by standard middleware, this approach is likely to be application-specific.

The key obstacle to detecting shared-memory communications with an application-independent mechanism is the absence of a standard communication interface. Many applications use shared memory in an unstructured way, where one thread can communicate with another using regular memory accesses. Distinguishing such accesses from operations on local memory may require tracking of most memory accesses. Although race detectors provide similar capabilities [89, 91, 107, 111], their overhead may be prohibitive in production environments.

Some OS-level communication primitives are structured similar to shared memory communications. For example, processes can communicate via a shared file, where a process P writes to a file and a process Q reads the data later. Following this control flow from P to Q may be important for some applications. An application can write periodic checkpoints to a file so that if a process fails, it can restart from the latest checkpoint. Writing to the checkpoint file and reading from it later should be considered part of the same control flow. In contrast, two independent processes reading and writing to a shared file to update system-usage statistics should not be part of the same control flow. Defining what constitutes a communication through such shared-state mechanisms and developing effective techniques for propagation across these communications remains future work.

3.3.3 Identifying the Name of the Peer Host

After detecting the start of communication, the agent needs to locate the peer process to inject itself in its address space and start following the flow of control in both processes. In Section 3.3.1, we showed how to determine the components of a $\langle hostid, pid \rangle$ UPL tuple for communication over TCP sockets. Here, we describe how we determine the *hostid* component for other mechanisms supported by our current prototype: pipes and UDP sockets. We also outline potential solutions for other types of message-passing communications, such as MPI (Message Passing Interface). In the next section, we consider our approach to determining the *pid* component.

We use three simple techniques for identifying the address of the destination host. First, if we determine that the communication mechanism is local (e.g., a pipe or a UNIX-domain socket), we can use the local IP address as the *hostid* component of the tuple. Second, for some remote communication mechanisms, the address is explicitly provided to the communication routine. For example, for interactions over unconnected UDP sockets, we extract the address of the destination from the arguments to the *sendto* system call [71]. Finally, mechanisms that establish explicit communication channels often provide routines for finding the address of the peer. In addition to TCP sockets, we use this mechanism for those UDP sockets whose remote peer was specified via the *connect* call [71].

There are several message-passing mechanisms where determining the host address is difficult. Some communication mechanisms do not know the destination address at send time. For example, the sender of an IP multicast datagram may not know the recipients of the datagram. Other mechanisms know the destination, but do not expose this information at the interface level. For example, nodes of an MPI application refer to each other by ranks, i.e., sequential identifiers from 0 to $N-1$, where N is the number of processes. MPI provides no capabilities for finding the host identifier for another process given its rank.

To support IP multicasts, we would need to inject copies of the agent into all potential receivers. However, identifying the list of multicast-receiving hosts is difficult as accurate group membership information is not maintained by the IGMP protocol. If all multicast peers are known to reside on the same local network, we could broadcast the injection request to all *spDaemons* and let each daemon ignore the request if its host contains no multicast receivers. Since most multicast applications are deployed in wide-area environments, the applicability of this technique is limited. Developing a more general solution remains future work.

If a communication subsystem maintains information about location of the receivers but does not export it via a standard interface, we may be able to identify the receiver at a lower communication layer. For example, if MPI routines are implemented using sockets, we can propagate through the nodes of an application via our socket-based mechanism. Note however that some MPI implementations operate on non-IP high-performance networks. Others support hybrid communications where nodes on the same host communicate via shared memory and use sockets only for inter-host communication. To support such environments in the future, we propose to augment a communication mechanism with capabilities for establishing the mapping between a mechanism-specific process identifier, such as a rank, and a UPL.

3.3.4 Identifying the Name of the Receiving Process

After finding the receiving host, we locate the *pid* of the receiving process on that host. In our experience, standard communication mechanisms do not provide the receiving *pid* information at the sending site. Therefore, we use a two-step approach to identifying the receiving *pid*. First, we determine the name of the communication channel on the sending host. Second, we provide this name to *spDaemon* on the receiving host and the daemon identifies all processes that have this channel open.

To each channel, we assign a name of the form $\langle type, id \rangle$, where *type* is the type of the channel (e.g., a pipe, a TCP socket, or a UDP socket) and *id* is a type-specific channel name. Each channel name must satisfy two requirements. First, the name has to be globally unique: no two channels in the system can have the same name. Second, the name must uniquely identify the channel: two names cannot refer to the same channel. In particular, the name of the channel as determined by the receiver must coincide with the name of the channel as determined by the sender.

Note that our logical channels are uni-directional. To represent bi-directional communications, such as those over a socket, we use a pair of channels. To name each channel for a socket, we use a tuple $\langle sendip, sendport, recvip, recvport \rangle$, where *sendip*, *sendport*, *recvip*, and *recvport* are the IP addresses and port numbers of the sending and the receiving sides respectively. To name a pipe, we use its inode number as reported by the *stat* operating system interface [71].

Once the agent on the sending side determines the name of a channel, it sends this name to *spDaemon* on the receiving side. For pipes and sockets, we then use the same technique to identify processes that access the channel. Namely, *spDaemon* converts the channel name to the inode number and then locates all processes that have a channel with that inode number open, as previously described in Section 3.3.1. Although our technique uses Linux-specific interfaces, similar information is also available on other operating systems, including Windows [110].

Note that we find a process that accesses a given inode by traversing the entire */proc* filesystem. In the future, we envision an operating system extension for identifying a list of processes that have a given file or socket open. This mechanism would considerably speed-up cross-host propagation. This mechanism would also provide for simpler implementation of tools, such as *netstat*, *fuser*, and *lsof*, that report associations between open files or network connections and processes. Identifying processes that listen on a given port is also an important operation for securing systems and detecting compromised systems.

3.3.5 Injecting the Agent into the Destination Component

Note that the above technique can identify multiple processes as potential receivers since the same channel can be shared by multiple processes. For example, daemons of the Apache Web server share the same listening socket to accept and service multiple connections concurrently. *spDaemon* needs to inject the agent into each of these processes as any of them can consume the data sent.

Furthermore, new processes sharing the channel may be created after *spDaemon* scanned the */proc* tree. To inject the agent into such processes, *spDaemon* repeats the scanning and injection procedure multiple times until no additional processes are discovered. In practice, the procedure typically converges after one iteration: the second iteration does not discover any additional processes. Note that if we manage to inject the agent into all processes that share the socket, further children of these processes will be created with the agent already injected.

There are cases however, when the agent cannot identify all processes accessing a channel. Consider a scenario when two processes communicate over a FIFO (a named pipe). We trace one of the processes, identify the moment when it sends a message to another process, and start tracing both processes. Then, a third (untraced) process opens the FIFO and starts consuming the data from the shared stream. Since neither of the traced processes can detect that the third process is now accessing the same stream, our agent cannot propagate into the new process. To address this limitation, we envision an operating system event notification interface that could inform our agent that a process joined an already-open data stream. Since we have not observed this scenario in practice, implementation of this interface remains low priority.

3.3.6 Detecting the Receipt at the Destination Component

After injecting the agent, we need to identify the moment when the process receives the message so that we can start to follow the flow of control. That is, we need to find a receive event that corresponds to the first observed send event (the send that initiated the injection procedure). Finding this event is a special case of a more general problem of matching all observed sends and receives. In this section, we describe our solution to the general problem since our data analyses require matching of all send and receive operations to construct an accurate PDG.

Our solution is most straightforward for order-preserving channels that are created after we started tracing. If a traced process issues a *connect* request to another process, we can start following the control flow of the peer when it completes an *accept* call with the matching IP address and port. Sends and receives on this channel can be matched using *byte-counting*: counting the number of bytes sent and received by each endpoint. This technique works for TCP sockets, pipes and other stream-based channels if they are created after activation of tracing. It may not work for UDP sockets since they do not preserve the order of messages, i.e., messages can arrive not in the order sent.

If an order-preserving channel had been created before we started tracing, we need to account for *in-flight messages*, messages that have been sent before the first observed *send* but received after we injected the agent into the receiver. In presence of in-flight messages, the first observed *recv* might not correspond to the first observed *send* but to an earlier *send* on the same channel. This situation can occur when the receiver process has a backlog of requests on a channel.

As described in Section 3.3.1, we address this problem for TCP sockets with a *byte-marking* technique. We use the TCP OOB (out-of-band) mechanism to place a mark on the first byte of the message being sent. When the receiver reads a message from the channel, our agent checks whether the message starts with a mark. If so, we can match the *send* that placed the mark with the *recv* that retrieved it.

To match subsequent *send* and *recv* operations, we use the byte-counting technique described above. Note that a channel may be shared by multiple senders and receivers. To address this scenario, we keep the byte counts in shared memory and each agent updates them atomically. This technique assumes that we injected the agent into all senders and receivers for the channel. Our receiver-identification mechanisms described in Section 3.3.6 can be used to find both the receivers and the senders.

The OOB mechanism is only supported for TCP sockets. For local order-preserving communication mechanisms such as pipes and UNIX-domain sockets we can use a different technique for dealing with in-flight messages. After injection, the agent in the receiving process can use operating system facilities to determine the size of the backlog (the number of bytes queued in the channel) via the *FIONREAD ioctl* interface [120]. Our byte-counting technique can then be extended to subtract this number from the running total of bytes received to match *send* and *recv* operations. Note that this mechanism may not work for cross-host communications where some in-flight messages may not have arrived at the destination host yet and are not counted as the backlog.

To match *send* and *recv* operations on UDP sockets, we can use a *datagram-*

marking technique. To allow for possible reordering, we need to place marks on all datagrams, not only on the first observed one. We also need to extend each mark with a sequence number to make each mark unique. On each *send* operation, we can then record the mark in the local trace and increment its current value so that all datagrams have unique mark numbers. On each *recv* operation, we can extract the mark from the datagram and append it to the local trace. At the analysis time, *send* and *recv* operations with equal mark values can be matched together.

We can encode the mark into each datagram by using IP options. The current TCP/IP implementation in Linux supports the TS (timestamp) option among others. The TS option causes each router to append a timestamp to a packet as it is being forwarded through the network. When sending a datagram, we can create a TS option and insert our mark as the first 32-bit timestamp in the option. To prevent routers along the path from appending their timestamps to our option, we can specify that the option does not contain any additional free space.

Note that for security reasons, some routers and firewalls remove IP options from forwarded packets or drop packets with options. To avoid losing application data in such cases, our agent can use a probe datagram to test whether datagrams with IP options are delivered to the endpoint. Determining how common the environments that drop datagrams with IP options are and developing solutions for such environments remain subjects for future work.

3.4 Summary of Data Collection Approaches

To summarize, the key property of our data collection approach is its ability to be deployed on-demand in a distributed system. Our agent propagates through the code of a process by following the flow of control. It propagates into other processes when a traced process communicates with them. As the agent propagates, it collects the control-flow trace of each process and records additional information about communication operations to construct the PDG for the execution.

We developed a general framework for propagation across message-passing communications and demonstrated the feasibility of our approach for three foundational communication mechanisms: TCP sockets, UDP sockets, and UNIX pipes. We also proposed several improvements to the operating system interfaces to make the propagation more efficient. Other message-passing communication mechanisms may benefit from similar techniques. However, efficient support for shared-memory communications remains an open question.

Chapter 4

Causal Flow Reconstruction

Our observation is that bugs and performance problems in complex systems are often manifested by deviations of control flow from the common path. To identify such deviations and determine the root cause of a problem, we collect control-flow traces and analyze them manually or automatically. In distributed environments however, collected traces may contain events that correspond to different concurrent activities, possibly of multiple users. For example, Web servers often service multiple concurrent HTTP requests. Events that correspond to processing the same request are not explicitly labeled and can occur in the trace interleaved with events from other requests. Therefore, manual or automated analysis of such interleaved traces is difficult for several reasons.

For manual examination, the presence of unlabeled events from multiple unrelated activities is confusing. The analyst needs to know which event belongs to which user activity. For automated analysis, interleaving events from multiple concurrent activities increases trace variability. Similar to other techniques for automated trace analysis [11, 25], we identify anomalies, i.e., unusual traces that are different from the common behavior. Due to concurrent processing, events from different activities may appear in a different order in different runs. As a result, even a normal trace may appear substantially different from previous ones and will be marked as an anomaly.

To perform trace analysis in such environments, we adopt the approach of Magpie [11] and Pinpoint [25]: we decompose the execution of an application into units that correspond to different activities. These units are easier to analyze manually and automatically than the original trace. They contain logically-related events and there is little trace variability within a unit. Such decomposition is straightforward in systems where each unit is encapsulated in a separate process (e.g., parallel numeric codes or some Web server farms where each request from the beginning to the end is handled by one process). If a process in the application interleaves processing of multiple activities, decomposition becomes more complex: the process can no longer be attributed to a single unit. In client-server systems for example, multiple units correspond to multiple requests that can be processed by a single server process.

We formulate the problem of constructing such units as a graph problem. First, we represent a system-wide control-flow trace as a PDG. Then, we partition the PDG into disjoint subgraphs that we call *flows*. Each flow represents a single instance of a semantic activity so that flows can be compared to each other. Sections 4.1 through 4.4 introduce our notation, define the flow-construction problem, and present techniques for solving it. Our techniques improve the accuracy of previous application-independent approaches [81] and have little reliance on user help, unlike application-specific approaches [11, 25].

Section 4.5 contains a glossary of the terminology we introduce in this chapter.

4.1 Definition of Flows

Our data analyses operate on flows, subgraphs of the PDG, that represent collections of logically-dependent events. We introduce flows with the following definitions.

Definition 1 (Basic block). Following [2], we define a *basic block* as a sequence of instructions in which flow of control enters at the beginning and leaves at the end without halting or branching except at the end.

Definition 2 (Event, Basic block event). We define an *event* as an execution instance of one or more instructions in an application. If these instructions form a basic block, we refer to this event as a *basic block event*. Such an event can be identified by a tuple $\langle addr,\ seq\_id \rangle$, where *addr* the address of a basic block and *seq_id* is a sequence number of this event, i.e., the number of executions of this basic block in the history preceding this event. For example, event $\langle$0x1234,10$\rangle$ refers to the 10^{th} execution of a basic block that starts at address 0x1234.

Definition 3 (Message-passing event). We assume that processes communicate by passing messages. Application code in our environment contains two special instructions, *send* and *recv*, whose execution represents sending and receiving a message. In real systems, such instructions can correspond to system call trap instructions. We place each *send* and *recv* in its own basic block and refer to their executions as *message-passing events*. Note that according to this definition, each message-passing event is also a basic block event.

Definition 4 (Local control flow trace). A *local control flow trace* of process p, denoted T^p, is the sequence of basic blocks events that occurred in process p. The i^{th} basic block event in T^p is denoted T^p_i.

The *send* and *recv* instructions operate on an object called *communication channel*. A communication channel is a uni-directional queue connecting a sender process with a receiver process. Depending on a communication mechanism, this queue can provide a reliable or unreliable delivery service, maintain the ordering of messages or allow out-of-order arrival, and preserve or disregard message boundaries.

- *sid = send(ch, buf, n)* writes n bytes of buffer *buf* into channel *ch* and returns a message identifier *sid*. In our system, each message sent has a globally-unique name that we call a *sent-message identifier*. These identifiers allow us to define the correspondence between messages sent and received. For example, for a message sent over a TCP socket, this identifier is a tuple $\langle sstart, ssize \rangle$, where *sstart* is the total number of bytes sent over the socket before this message and *ssize* is the number of bytes in the message itself. Although many existing communication mechanisms do not provide such identifiers, they can be constructed using techniques described in Section 3.3.6.

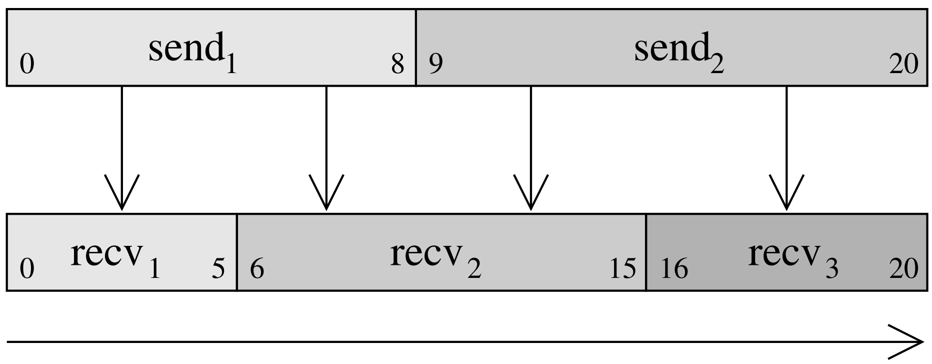
- $\langle$*rid, buf*$\rangle$ = *recv(ch)* waits for a message to arrive from channel *ch*, and returns *rid*, a *recv-message identifier*, and the message contents *buf*. A recv-message identifier for a message received from a TCP socket is a tuple $\langle rstart, rsize \rangle$, where *rstart* is the total number of bytes received from the socket before this message and *rsize* is the number of bytes in the message itself.

Definition 5 (Matching message-passing events). For each communication mechanism, we assume the availability of a *message-matching function* μ*(sid,rid)* that operates on a pair of message identifiers *sid* and *rid*. A send event s^p in process p and a receive event r^q in process q are *matching* if $\mu(sid, rid) = true$, where sid is the message identifier returned by s^p and rid is the message identifier returned by r^q.

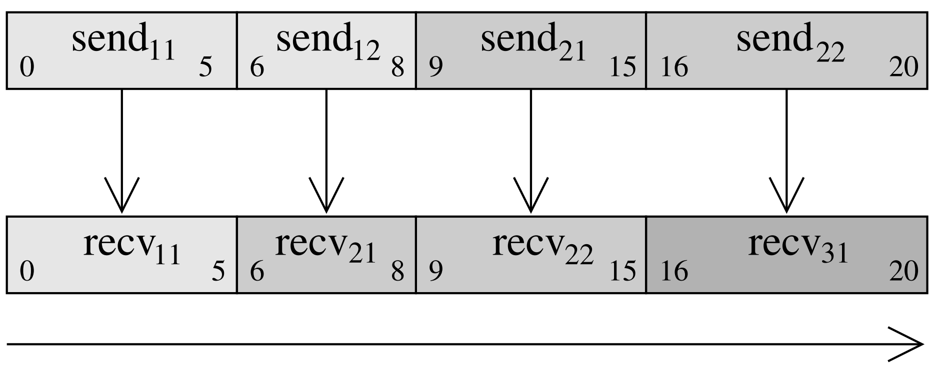
For UDP sockets and other unreliable channels that preserve message boundaries, a unique message identifier can be encoded in each datagram, as described in Section 3.3.6. In that case, the message-matching function determines whether the identifiers sent and received are equal: $\mu(sid, rid) = (sid = rid)$.

For TCP sockets and other reliable channels that do not preserve message boundaries, the message-matching function determines whether the ranges of bytes sent and received overlap: $\mu(sid, rid) = (sstart < rstart + rsize) \wedge (sstart + ssize > rstart)$. Assume that a process p sent a total of N bytes over a TCP socket during execution; a process q received N bytes from the same socket. The *sstart* components of sent-message identifiers partition the $[0, N)$ interval of bytes into *sent-intervals*. Figure 4.1a shows two sent-intervals, $send_1$ and $send_2$. Similarly, the *rstart* components of recv-message identifiers partition the $[0, N)$ interval into *recv-intervals* ($recv_1$, $recv_2$, and $recv_3$ in Figure 4.1a). Events $send_i$ and $recv_j$ match if and only if the sent-interval of $send_i$ overlaps with the recv-interval of $recv_j$.

Note that the relationship μ between *send* and *recv* events can be many-to-many. Data sent via one invocation of *send* may be received with multiple *recv* calls (e.g., $send_1$ in Figure 4.1a matches $recv_1$ and $recv_2$). A single *recv* call may also receive data from multiple *send* calls (e.g., $recv_2$ matches $send_1$ and $send_2$). Below, we present an algorithm that replaces *send* and *recv* events with equivalent sequences of events that have the one-to-one mapping between *send* and *recv* events. The assumption of one-to-one correspondence between message-passing operations simplifies further



a: *The correspondence between original* send *and* recv *events*



b: *Partitioning message-passing operations into fragments to establish one-to-one mapping between the fragments. A fragment marked* $send_{ij}$ *or* $recv_{ij}$ *represents the* j^{th} *fragment of the original operation* $send_i$ *or* $recv_i$.

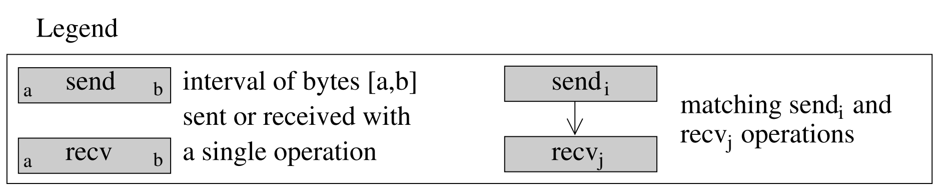


Figure 4.1: *The correspondence between* send *and* recv *events for TCP sockets before and after partitioning. Each event is shown as an interval of bytes sent or received.*

presentation of our techniques.

Consider the partition of the $[0, N)$ byte-range into intervals formed by all *sstart* and *rstart* components. Figure 4.1b shows this partition for the previous example shown in Figure 4.1a. The union of all intervals between two successive *sstart* values corresponds to an original sent-interval (e.g., the union of $send_{11}$ and $send_{12}$ corresponds to $send_1$). The union of all intervals between two successive *rstart* values corresponds to an original recv-interval (e.g., the union of $recv_{21}$ and $recv_{22}$ corresponds to $recv_2$). By replacing each original *send* event with multiple *send* events and each original *recv* event with multiple *recv* events that operate on such intervals, we arrive at two sequences of matching message-passing events.

Definition 6 (Parallel Dynamic Program Dependence Graph). A *parallel dynamic program dependence graph (or parallel dynamic graph, PDG)* of an application's execution is a DAG (N, E) where N is a set of nodes and $E \subseteq N \times N$ is a set of edges. The set of nodes N consists of all basic block events in all processes in the application. There exists an edge $(u, v) \in E$ from node u to node v (denoted $u \rightarrow v$) if either condition holds:

1. $\exists p \in P, i > 0$ s.t. $u = T_i^p$, $v = T_{i+1}^p$, where P is the set of all processes in the application. That is, consecutive basic block events u and v in the same process are connected by an edge. This edge represents local control flow of a process.

2. u and v are matching message-passing events. This edge represents inter-process communications.

Note that unlike the previous definition of PDG as introduced by Choi et al. [29], our PDG contains not only communication events but also local control flow events.

Such PDG represents a program execution with higher precision.

An important property of the PDG is its ability to represent Lamport's happened-before relation [61] between events. Namely, an event u happened before another event v if and only if v is reachable from u in the PDG, i.e., $u \rightarrow^* v$, where $\rightarrow^*$ is the reflexive transitive closure of the $\rightarrow$ relation defined by the PDG. Informally, u happened before v if there exists a path in the PDG from u to v.

If event u happened before event v, then u could causally affect v. This statement has two important implications that can be used in debugging. First, if v corresponds to the occurrence of a failure, then the root cause of the failure is likely to be one of the events that happened before v. This observation has been used in automated debugging tools that perform flowback analysis [10, 29] and backward slicing [37, 56, 131]. Second, if event u corresponds to a fault-inducing input to the system (e.g., a keypress that causes an application to fail) then the first symptom of the problem is likely to happen after u. This symptom is often useful for locating the root cause of the problem. The combination of these implications has been used in *chopping* techniques to identify faults that are caused by an input event and that cause the failure [44].

Similar to chopping, we define a set of events that can reach a given stop event (e.g., a failure or a normal end-request event) and that are themselves reachable from a given start event (e.g., a user input).

$$\text{Chop(u,v)} = \{x \in N \text{ s.t. } u \rightarrow^* x, x \rightarrow^* v\}$$

We refer to events u and v as *chop endpoints*.

Chops can be used to represent activities that have well-defined start and stop points. For example, events that correspond to processing an HTTP request by a Web

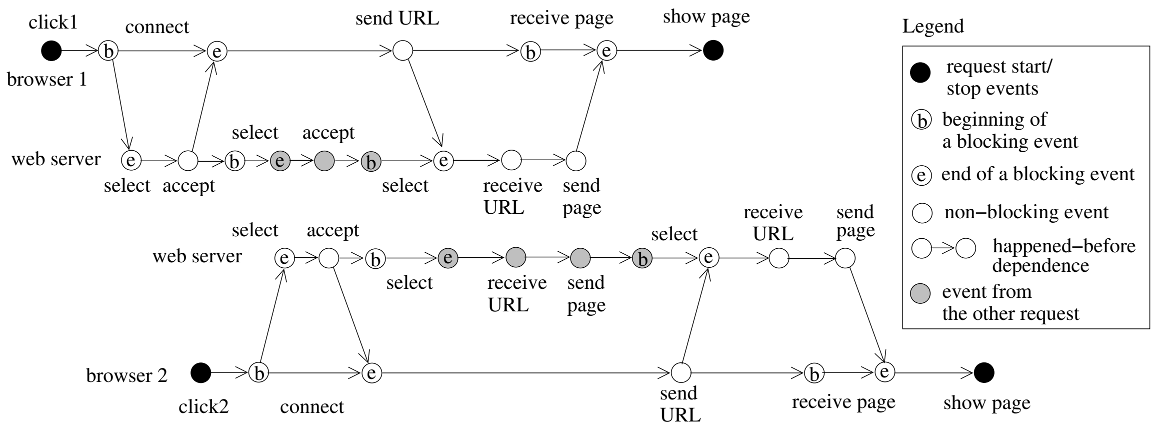


Figure 4.2: *Chops for a Web server executing requests from two Web browsers*

server happen after the server accepts a user connection and before the server closes the connection on replying to the user. Connection opening and closing events can be used as the chop endpoints. Events that belong to the HTTP request will belong to the chop between these endpoints.

Note however that in presence of multiple concurrent activities, a chop built between the endpoints of one activity may contain some events that belong to a different activity. That is, the chop is a superset of events in the first activity. Suppose a single-process Web server accepted an HTTP connection. Before the client sent a URL, another client issued a second connection thus starting another request. The PDG for this scenario was previously shown in Figure 1.1.

The chops constructed for the endpoints of both requests are shown in Figure 4.2. Note that each chop contains some events from the other request (shown as gray dots). These events would not be recorded if the requests were submitted sequentially. Also note that the obtained chops are different: the gray dots in the first chop correspond to accepting a new connection; the gray dots in the second chop correspond to receiving a URL and sending the reply. The variability of chops would complicate automated anomaly identification in the collected data. The presence of events from one request in the chop for another request would also complicate manual data examination.

Our goal is to decompose the constructed PDG into units that can be examined manually and also automatically compared to each other. Any observed difference between two units should be caused by the logic of the application.

Definition 7 (Logical flow). A *logical flow*, or simply *flow*, is a set of events that correspond to a semantic activity in the execution.

4.2 Properties of Flows

Flows are application-specific objects that depend on how activities are defined in the system. In our experience however, flows satisfy four application-independent conditions that we call *flow properties*. An algorithm that constructs sets of events satisfying these conditions is able to determine flows with little knowledge of application internals. Flow Properties 1, 2, and 3 are satisfied for activities in all systems. Flow Property 4 held in example systems that we studied, including cluster management tools, Web servers, and GUI applications. We also outline techniques for detecting when this property is violated and addressing such violations.

Flow Property 1 (Start-reachability). Each flow has a single start event that happens before all other events in the flow. This property corresponds to the assumption that many activities have a well-defined start event. If set Φ is a flow, we denote its start event as $\Phi.start$. In this notation, if event $x \in \Phi$, then $\Phi.start \to^* x$. That is, all events in a flow must be reachable from its start event.

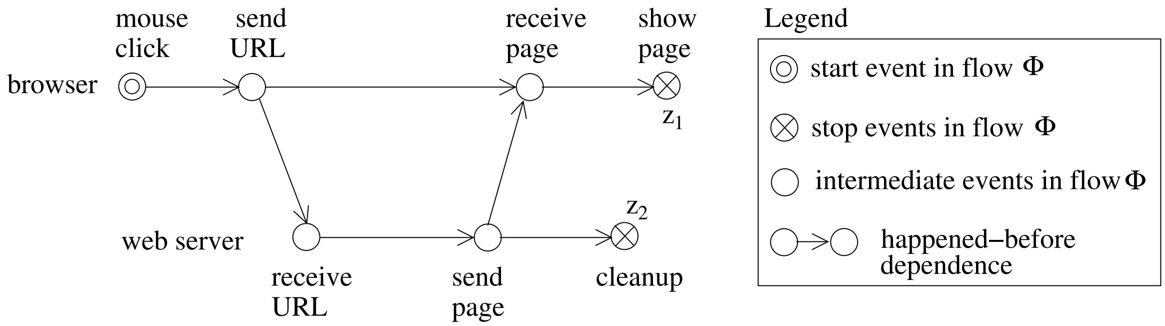


Figure 4.3: *Start and stop events in a client-server system*

For all the activities that we have observed in real-world systems, we were able to identify a single event that happens before all other events. In client-server systems for example, the request arrival event happens before all other events in processing the request. In a distributed job scheduler, the job submission event happens before all events in scheduling and executing the job.

Flow Property 2 (Stop-reachability). Each flow has one or more stop events that happen after all other events in the flow. This property corresponds to the assumption that each recorded activity has finite duration: it can complete successfully, fail, or the user can explicitly stop trace collection. Consider an example shown in Figure 4.3 where a client sends an HTTP request to the Web server, receives a reply, and displays the received page on the screen. The mouse click event corresponds to $\Phi.start$. Events z_1 (the client showing the page) and z_2 (the server performing cleanup, e.g., closing the network connection) correspond to stop events since all other events happen before z_1 or z_2. Note that events z_1 and z_2 are not ordered with respect to each other, and flow Φ cannot have a single stop event.

This property is satisfied for any constructed PDG. Assume that a semantic activity is represented by a set of events Φ. Stop events correspond to events in Φ that have no successors in Φ. Since the PDG is a DAG, Φ contains at least one event that has no successors or whose successors do not belong to Φ.

Flow Property 3 (Completeness). Each event in the PDG must belong to a flow. This property represents the fact that each event in the execution can be attributed to at least one semantic activity. If $\Phi_1, \ldots, \Phi_f$ represent all flows, then $\cup_{i=1}^{f} \Phi_i = N$, where N is the set of nodes in the PDG. Note that the completeness property is satisfied in all systems: for any event in the execution, we must be able to identify at least one activity to which this event corresponds.

Flow Property 4 (Non-intersection). Flows are disjoint, i.e., no event can belong to more than one flow. This property corresponds to the assumption that each thread of control in the system works on a single semantic activity at a time. If sets Φ_1 and Φ_2 represent different flows then $\Phi_1 \cap \Phi_2 = \phi$. Combining Flow Properties 3 and 4, we conclude that each event x in the PDG belongs to exactly one flow. We denote this flow $\Phi(x)$.

There are scenarios where the non-intersection property may not hold. For example, some systems use *message bundling* to send multiple messages with one *send* operation. In Figure 4.4, two messages from different flows are bundled into a larger message. The *send* and *recv* operations belong to both requests, the intersection of flows Φ_1 and Φ_2 is not empty, and Flow Property 4 is violated. Note that we have not seen this scenario in practice, since bundling is typically performed for messages within the same request.

A simple technique for addressing this problem is to modify the PDG to produce

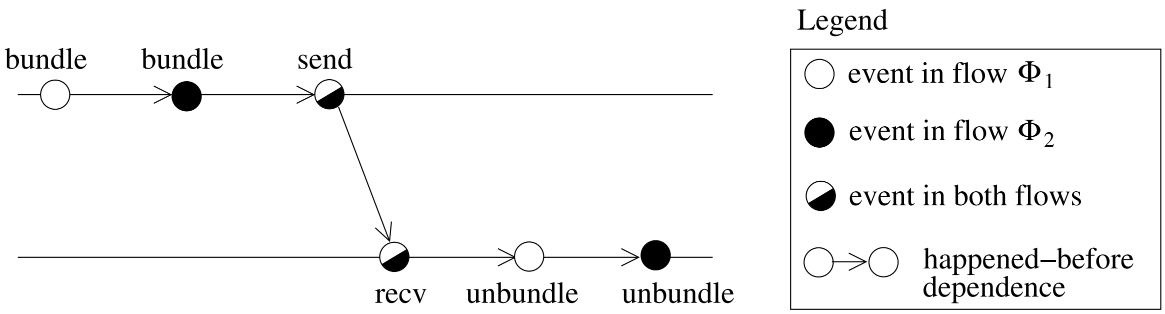


Figure 4.4: *The PDG for a system that uses message bundling. The send and recv events belong to both requests and violate Flow Property 4.*

an equivalent execution where bundling does not occur. For example, we can rely on the user to mark all locations in the code where bundling and unbundling occur. This step can be accomplished with user directives described in Section 4.3.4. We can then replace each bundled *send* or *recv* event with a pair of events, and attribute each event to a separate flow. This step is similar to establishing one-to-one correspondence between message-passing operations by partitioning them into unit operations, as shown in Figure 4.1.

Another potential violation of the non-intersection property is execution of initialization code that can be used for multiple flows. For example, multiple Condor jobs with similar requirements can be executed in a sequence on the same host. This host is reserved (claimed) once for the first job and the claim is reused for subsequent jobs. Events involved in acquiring the claim correspond to handling more than one job. They are shared by multiple flows and thus violate the non-intersection property.

Similar to the bundling problem, we can restore the non-intersection property by replacing each shared event in the PDG with multiple artificial events, one for each flow. This approach would produce identical flows for all jobs. Alternatively, we can attribute all shared events to the first job, thus introducing the difference between the first and the subsequent jobs. This alternative approach proved useful in Section 6.3, where the root cause of the problem was located in the initialization code and the difference between flows enabled us to locate it. Detecting violations of the non-intersection property automatically and choosing the appropriate strategy for handling the violations remain subjects for future work.

4.3 Flow-construction Algorithm

We designed an algorithm that constructs sets of execution events that satisfy the four flow properties. As described in Section 3.1, our tracing techniques require the user to provide the set of start events S. We then collect all events that happen after any event in S until the user decides to stop tracing. The collected traces allow us to match *send* and *recv* events and construct the PDG. Note that all events in our PDG are reachable from at least one event in S.

We begin describing our algorithm by assuming that the user specified a complete set of start events for all activities recorded in the PDG. That is, if the PDG contains an event x from flow Φ, then $\Phi.start$ also belongs to the PDG, and the user identified $\Phi.start$ as one of events in S. The completeness of S implies Flow Property 1. In Section 4.3.3, we extend our algorithm to operate correctly when the user specifies an incomplete set S.

In Section 4.2, we showed that Flow Property 2 is satisfied for any set of events in the PDG. To satisfy Flow Properties 3 and 4, we formulate a graph transformation problem: given a PDG and a set of start events S, we transform the PDG into a new dependence graph composed of several disjoint subgraphs that contain all nodes of the original PDG. We will refer to this dependence graph as a *Flow PDG* or *FPDG*. The disjoint subgraphs of FPDG correspond to flows. An example FPDG was previously shown in Figure 1.2. Note that while flows are application-specific objects, in our experience they often can be constructed with application-independent rules.

4.3.1 Types of Graph Transformations

We write $G \xrightarrow{\psi} G'$ to denote a graph transformation ψ. Here, $G = (N, E)$ is a PDG; $G' = (N, E')$ is a transformed PDG with a modified set of edges $E' \subseteq N \times N$; our transformations do not modify the set of nodes N. The nodes correspond to observed events and cannot be added or deleted. Overall, our goal is to find a sequence of transformations $\psi_1, \ldots, \psi_m$ such that $G \xrightarrow{\psi_1} G_1 \xrightarrow{\psi_2} \ldots \xrightarrow{\psi_m} G_m$ where G_m is the FPDG. Below, we discuss the main types of our transformations.

We provide several rules for transforming the PDG. However, all our transformations belong to two types. First, our key transformation type is the removal of an edge in the PDG. We need to remove edges that connect unrelated flows. That is, if we determine that nodes u and v in the PDG belong to different flows, we introduce the following transformation:

Transformation 1. (Edge removal between nodes u and v)

$$G = (N, E) \xrightarrow{\psi} G' = (N, E'), \text{ where } E' = E \setminus \{(u, v)\}$$

We determine where to apply this transformation (what edges to remove) using several rules and user-provided directives; Sections 4.3.2 and 4.3.4 describe these rules and directives.

By removing edges in the PDG, we can produce a graph where events from different flows are not connected. However, these transformations may make an event unreachable from the start event of the same flow, thus violating Flow Property 1. This situation could not occur in the original PDG as all events of a flow are assumed to happen after the start event. If we determine that events u and v belong to the same flow, v was transitively reachable from u in the original PDG, and v is not reachable from u in the transformed PDG, we can add the (u, v) edge to the PDG:

Transformation 2. (Insertion of a transitive edge between nodes u and v)

$$u \to^* v \text{ in PDG, } u \not\to^* v \text{ in } G',\ G' = (N, E') \xrightarrow{\psi} G'' = (N, E''), \text{ s.t. } E'' = E' \cup \{(u, v)\}$$

To determine where to apply Transformations 1 and 2, we have developed several techniques that can be combined with each other to construct a composite transformation. Each technique operates on a pair of connected PDG nodes and determines whether they belong to the same flow, different flows, or the technique is unable to classify them. We distinguish two classes of techniques: rules and user directives. Our rules work for most message-passing applications; they are application-independent, but may occasionally mis-attribute some events to a wrong flow. Directives provide a framework for the user to improve the accuracy of the rules, but they may require detailed knowledge of application internals.

4.3.2 Rules

We use two rules in our analyses. First, the *communication-pair* rule dictates that the pair of matching communication events belongs to the same flow:

$$\text{if } s^p, r^q \text{ are matching } \textit{send} \text{ and } \textit{recv} \text{ events, then } \Phi(s^p) = \Phi(r^q) \tag{4.1}$$

Figure 4.5: *The communication-pair rule*

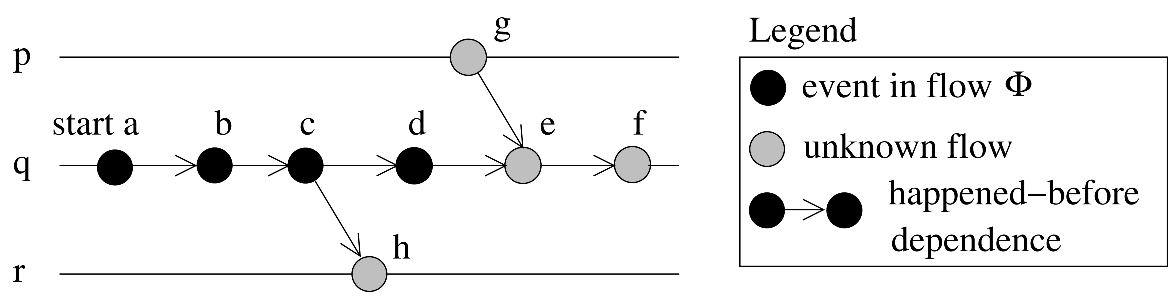


Figure 4.6: *The message-switch rule*

That is, if we know that a *send* event s^p belongs to flow Φ then the matching *recv* event r^q also belongs to Φ. In Figure 4.5, we assume that *send* event s^p belongs to flow Φ and use the communication-pair rule to assign the *recv* event r^q to flow Φ. Note that this rule does not specify whether u^q and other events following r^q in process q belong to flow Φ. Process q might receive the message on flow Φ, store it in the queue, and return to servicing another request.

The communication-pair rule specifies that nodes connected by an inter-process edge in the PDG belong to the same flow. Some intra-process edges however, connect events from different flows. For example, a Web server process can interleave processing of multiple requests, i.e., temporarily switch from working on one request to another. To identify intra-process PDG edges that connect same-flow events, we introduce the *message-switch rule*:

$$\text{if } deg^+(T_i^p) = 1 \text{ then } \Phi(T_i^p) = \Phi(T_{i-1}^p) \tag{4.2}$$

Here, $deg^+(x)$ is the in-degree of x (i.e., the number of immediate predecessors of node x in the PDG). If a node has only one predecessor (i.e., it is not a *recv* node), then it belongs to the same flow as the previous event in the same process. This rule suggests that a process can switch from one flow to another only on receiving a message.

Our PDG has nodes of three types: nodes with the in-degree 0, 1, and 2. Nodes with the in-degree equal to 1 have a single predecessor; they correspond to non-message-passing events and *send* events. Nodes with the in-degree equal to 2 have two predecessors: one in the local process, another in remote; such nodes correspond to *recv* events. Finally, nodes with the in-degree equal to 0 have no predecessors; they correspond to user-provided start events. A start event v may also have a non-zero in-degree (e.g., it occurs in the same process as an earlier start event u). In that case, we apply Transformation 1 to remove all edges that end at v. A start event must not be reachable from events in other flows.

Figure 4.6 shows an example of applying the message-switch rule. Events b, c, and d in process q have in-degrees equal to 1. They happen after start event a and are assigned to flow Φ. Event e is a receive; it has an in-degree equal to 2 and its flow is unknown. Event f belongs to the same flow as e. Events g and h belong to other processes and this rule does not specify their flows.

The message-switch rule implies that an intra-process edge $x \rightarrow y$ in the PDG can be removed by our graph transformations only if node y has an in-degree equal to 2 (i.e., y is a *recv* event). If node y has an in-degree equal to 1, it belongs to the same flow with its predecessor x and the edge between them need not be removed.

The combination of the communication-pair rule and the message-switch rule defines our main approach to flow separation. We begin with the complete PDG G and the set of all start events S provided by the user. We apply Transformation 1 to remove all edges that connect a *recv* event with its preceding event in the same process. Edges between other pairs of connected events should not be removed as such events belong to the same flow, according to the communication-pair or the message-switch rules. We denote the transformed graph G'. All *recv* events in the transformed PDG have the in-degree equal to 1.

For each start event $u \in S$, we denote the set of all events reachable from u in the transformed graph G' as $R(u, G') = \{x | u \rightarrow^* x \text{ in } G'\}$. We can then use $R(u, G')$ to represent the flow that starts at u. Figure 1.2 previously showed the results of this algorithm in separating two concurrent requests in a Web server.

Our algorithm constructs flows with desired properties. By definition, each $R(u, G')$ satisfies Flow Property 1 as event u happens before all other events in $R(u, G')$. Since we operate on graphs with finite number of nodes, Flow Property 2 is also satisfied. To see that sets $R(u, G')$ satisfy Flow Properties 4 and 3, consider the following claims.

Claim 1 (Non-intersection). Sets $R(u, G')$ and $R(v, G')$ are disjoint:

$$\text{if } u, v \in S, u \neq v, \text{ then } R(u, G') \cap R(v, G') = \phi$$

Proof. Assume that $R(u, G') \cap R(v, G') \neq \phi$, i.e., $\exists x \in R(u, G') \cap R(v, G')$. The in-degree of x in G' is either 0 or 1, since Transformation 1 removes one of the incoming edges from all nodes with the in-degree of 2. If $deg^+(x) = 1$, then x cannot be the same as u or v since $deg^+(u) = deg^+(v) = 0$. Therefore, event u is either the immediate predecessor of x or the path from u to x contains the immediate predecessor of x, node w. Suppose u is the immediate predecessor of x. Node x is reachable from v, x has only one predecessor, and $v \neq x$, therefore, u must be reachable from v. We arrive at a contradiction since $deg^+(u) = 0$ and $u \neq v$.

Suppose the path from u to x contains node w, the immediate predecessor of x. Since x is reachable from v, x has only one predecessor, and $v \neq x$, then w must be reachable from v. Therefore, w is reachable from both u and v, i.e., $w \in R(u, G') \cap R(v, G')$. If $deg^+(w) = 1$, we continue traversing G', moving from a node with the in-degree equal to 1 to its immediate predecessor until we reach a node t with the in-degree equal to 0. Such a node must exist since G' is a DAG as it was obtained from the PDG by removing edges. Similar to w, node t must belong to $R(u, G') \cap R(v, G')$. Since its in-degree is 0, t must correspond to both u and v. However, $u \neq v$ and we arrive at a contradiction. Therefore, $R(u, G') \cap R(v, G') = \phi$. □

Claim 2 (Completeness). For all $u \in S$, sets $R(u, G')$ contain all events in the PDG:

$$\bigcup_{u \in S} R(u, G') = N$$

Here, N is the set of all events in the PDG.

Proof. Assume that there exists an event x that is unreachable from any start event: $\forall u \in S,\ x \notin R(u, G')$. Similar to the proof of Claim 1, we traverse G' from node x moving to its predecessors until we reach a node t with the in-degree equal to 0.

Node t is unreachable from all $u \in S$ (otherwise, x would also be reachable from some u). Since its in-degree is 0 in G', its in-degree was also 0 in the original PDG (our transformation only affects nodes with the in-degree equal to 2). Therefore, it was also unreachable from any $u \in S$ in the original PDG. By construction however, each event in the original PDG is reachable from at least one start event. We arrive at a contradiction and thus, each event must belong to $R(u, G')$ for some $u \in S$. $\square$

4.3.3 Supporting an Incomplete Set of Start Events

In the above discussion, we assumed that the user specified a complete set of start events for all activities recorded in the PDG. That is, if the PDG contains an event x from flow Φ, then $\Phi.start$ also belongs to the PDG and the user identified $\Phi.start$ as one of events in S. When this assumption is violated, we can construct flows that satisfy all four properties but attribute some events to the wrong flow. Here, we present such a scenario and generalize our algorithm to operate with incomplete information from the user.

Consider the example in Figure 4.7 where a Web server processes requests from two users. The first user specified that we should start tracing the execution of $browser_1$ on detecting a mouse click on a Web link. The detected click event is the start event for flow Φ_1. The second user did not start tracing and we cannot record the second start event. However, since the Web server receives the second request after the first one, we record all events of the second request in the Web server, and subsequently, in $browser_2$.

Observed events in the second request must be attributed to a separate flow. However, since the second URL-receive event does not have a matching *send*, its in-

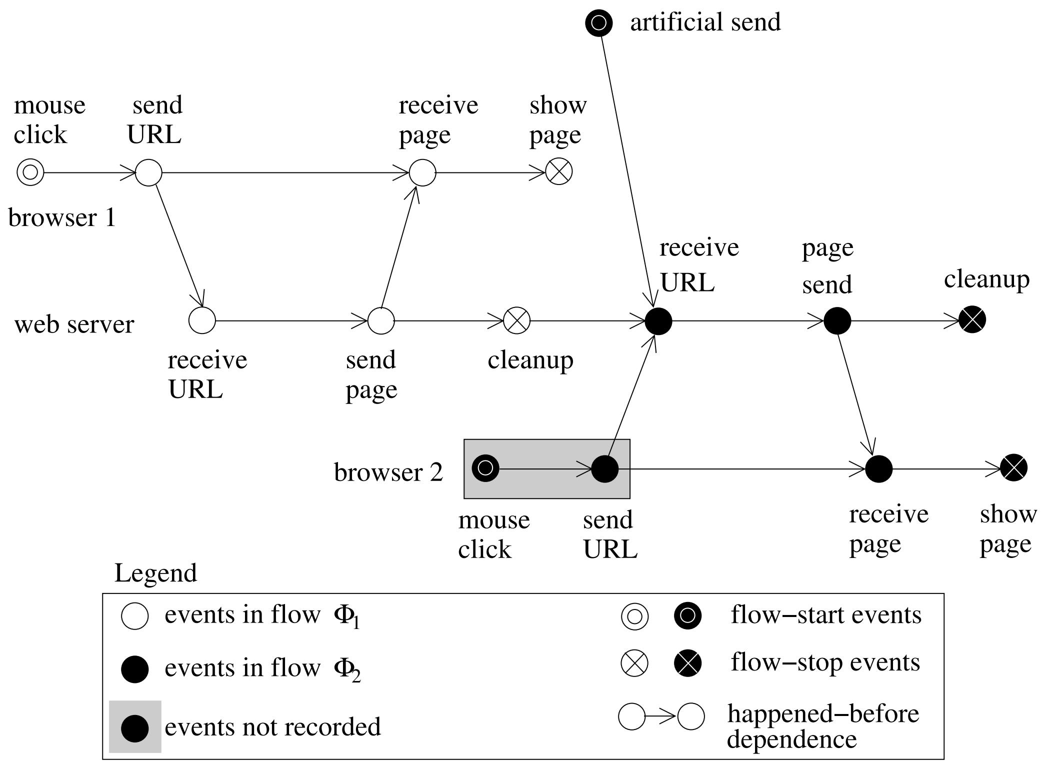
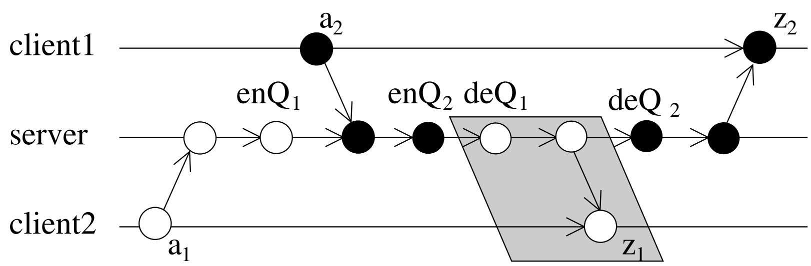
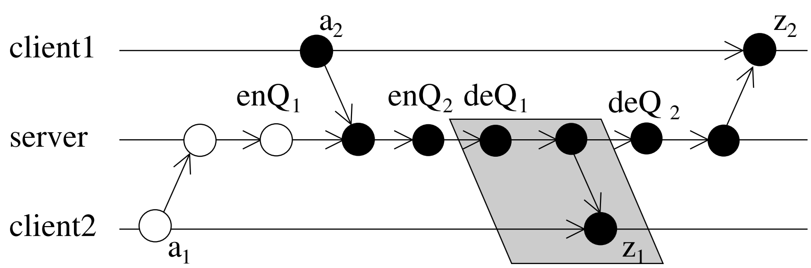


Figure 4.7: *The PDG for a Web server that executes requests from two browsers. The user identified only one start event, from flow* Φ_1*. Events from flow* Φ_2 *in the Web server are recorded but the corresponding start event is not. We detect this situation and introduce an artificial send event for flow* Φ_2*.*



a: *The desired mapping of events to flows*



b: *The incorrect mapping constructed by our rules*

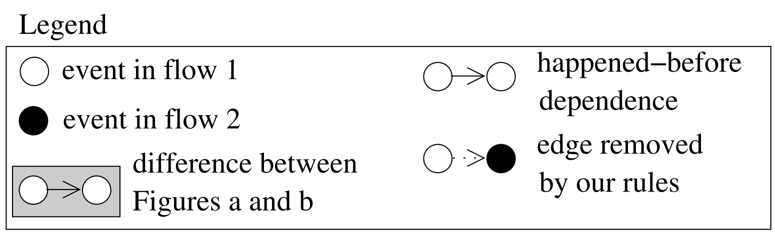


Figure 4.8: *The PDG for a system that uses a queue to hold and process incoming requests. Events enQ$_i$ and deQ$_i$ correspond to enqueueing and dequeueing request i.*

degree in the PDG is 1. Our algorithm would incorrectly attribute this and further events of the second request to Φ_1. To address this limitation, we identify each *recv* event that does not have a matching *send* event, create an artificial matching *send* event, and add this *send* to the set of start events for the PDG. In Figure 4.7, this technique allows us to create flow Φ_2 with events of the second request.

4.3.4 User Directives

Even when the user-provided set of start events is complete, our algorithm may attribute some events to the wrong flow. Consider a scenario where a server can receive requests with high or low priorities and the priority of a request can only be determined after receiving it. Figure 4.8a shows an example where the server process starts servicing a low-priority request 1, detects that another request is available, saves request 1 in a queue, retrieves request 2, determines that it also has the low priority, and returns to servicing requests in their order of arrival (request 1 then request 2).

Figure 4.8a shows the correct mapping of events to flows that was done manually. Figure 4.8b shows the automated mapping constructed by our algorithm. Note that the switch from the second request to the first request occurs on the application-specific enQ_2 event, without a receive. As a result, our algorithm does not detect the change in flows and incorrectly attributes all events after enQ_2 to the second flow. Here, we describe our techniques for addressing such problems with user help.

We developed a simple framework that allows the user to provide application-specific knowledge to the analysis in the form of *mapping directives*. These directives have structure similar to our rules: they identify pairs of events that should belong to the same flow. Such directives allow us to insert additional transitive edges into the PDG. Note that directives have precedence over our rules. If a rule and a directive assign an event to two different flows, we choose the flow specified by the directive and remove the PDG edge that caused our rule to assign the event to the wrong flow. Operationally, this step corresponds to applying transformations specified by directives before transformations specified by the rules.

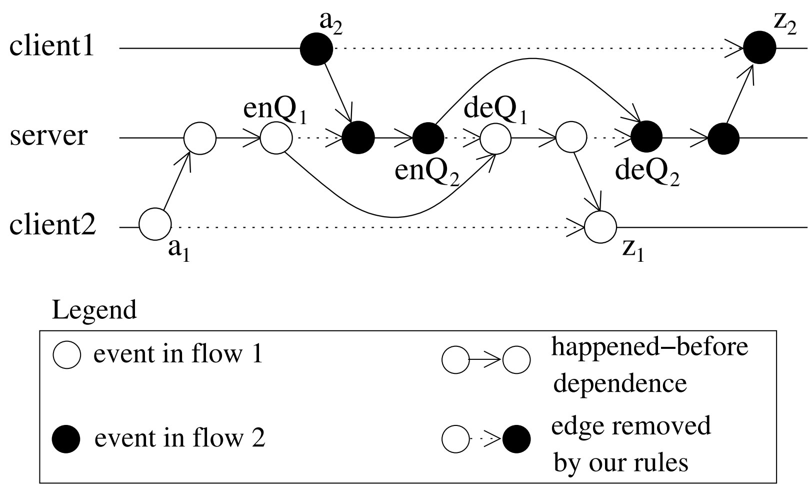


Figure 4.9: *The mapping constructed by rules and directives*

Consider the scenario previously shown in Figure 4.8a. If the user specifies that events enQ_i and deQ_i must belong to the same flow, we introduce two transitive edges to the PDG: $enQ_1 \rightarrow deQ_1$ and $enQ_2 \rightarrow deQ_2$, as shown in Figure 4.9. Note that event deQ_1 now has the in-degree equal to 2. As a result, we can remove the edge between enQ_2 and deQ_1, and these events are correctly assigned to different flows. Note that we use directives to match events within a process and introduce intra-process transitive edges. If the need for matching events in different processes arises, our approach could be extended to provide such capabilities.

To specify directives, we use an approach similar to Magpie [11]. Our directives have the form $\langle bb, jattr \rangle$, where bb is the address of a basic block in the code, *jattr* is a so-called *join attribute*. The result of this directive is labelling a basic block event corresponding to execution of bb with the join attribute of *jattr*. Basic block events u and v with the same value of the join attribute are assigned to the same flow:

$$\text{if } v.jattr = u.jattr, \text{ then } \Phi(v) = \Phi(u) \tag{4.3}$$

We translate each directive into a code fragment that is inserted into the basic block *bb*. When executed, the fragment saves *bb* and the value of *jattr* in the trace. At analysis time, events with the same value of *jattr* are then assigned to the same flow.

For example, to match invocations of *enQ* and *deQ* routines in Figure 4.9, the user can provide two directives: ⟨*enQ:entry,arg*⟩ and ⟨*deQ:exit,ret_val*⟩. These directives assume that the argument to the *enQ* routine is the address of the request structure to enqueue; the *deQ* routine returns that address on exit. At analysis time, we identify trace records where the argument to *enQ* was equal to the return value of *deQ* and introduce an artificial edge between them.

This example uses a global identifier (the request address) that is valid for the duration of the flow as the join attribute. Similar directives can be created in cluster-management tools (e.g., using the job identifier in the Condor system) and other client-server environments (e.g., using the I/O request address in the file server [60]). Note however that join attributes need not be valid for the entire duration of a flow. In our experience, temporary identifiers that are created and deleted during flow execution may also be useful for matching related events together.

In our studies, temporary identifiers proved useful in the Condor environment, where daemons use custom timers to defer execution of some actions until a later time. A daemon registers a timer handler function with the middleware and receives a temporary timer identifier. The middleware then invokes the installed handler when specified. We address such deferred control transfers by using directives to match timer identifiers at registration and handler invocation times.

Since multiple timers can be created and run within a single flow, a timer identifier can be reused within a flow or across flows. Suppose a process works on flow

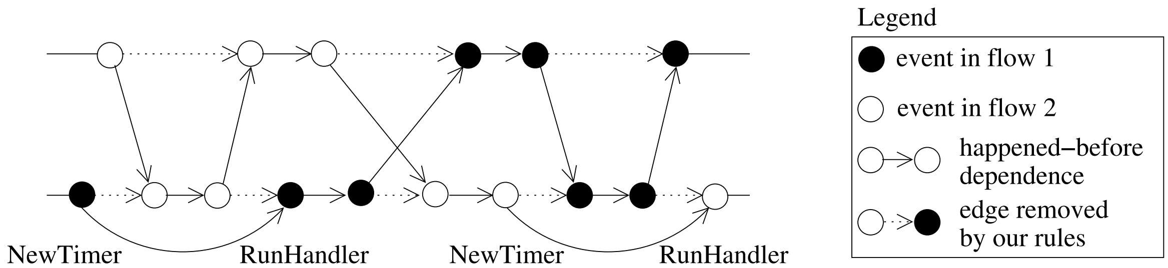


Figure 4.10: *Using timer directives to match NewTimer and RunHandler*

Φ_1, creates a new timer with identifier *id* and runs its handler. Later, this process starts working on flow Φ_2, creates another timer, and the timer receives the same identifier *id*. As a result, the new timer creation and execution events are attributed incorrectly to the same flow as the previous ones. Therefore, temporary identifiers must be protected against reuse across flows.

We address the reuse problem with an approach similar to that of Magpie. For each directive that uses a temporary attribute, the user must specify whether it creates a new scope of attribute validity (a *begin-scope directive*) or uses the attribute and then destroys the current scope (an *end-scope directive*). Since all our directives are intra-process, events with the same value of the join attribute are totally ordered. Therefore, begin-scope and end-scope events form natural matching pairs: each end-scope event is matched with the latest preceding begin-scope event that has the same value of the join attribute.

Consider a simplified example of timer directives where the *NewTimer* function installs a handler and returns a temporary timer identifier, *RunHandler* accepts the timer identifier as the argument, calls the corresponding handler, and destroys the timer. These operations can be matched using two directives: the begin-scope

$\langle$*NewTimer:exit,ret_val*$\rangle$ directive and the end-scope $\langle$*RunHandler:entry, arg*$\rangle$ directive. That is, *NewTimer* and *RunHandler* events are matched if the return value of *NewTimer* equals the argument of *RunHandler*.

Figure 4.10 shows an example where two flows create two timers with the same identifier in the same process. However, since the *NewTimer* and *RunHandler* directives begin and end the validity scope of the join attribute, the first invocation of *NewTimer* is matched only with the first invocation of *RunHandler*; the second invocation of *NewTimer* is matched only with the second invocation of *RunHandler*.

4.4 Putting the Flow Construction Techniques Together

To construct FPDG, we begin with a set of start events S. The start events are specified by the user, and we use them to start tracing distributed control flow, as discussed in Chapter 3. The collected traces allow us to match *send* and *recv* events and construct the PDG. The following steps decompose the PDG into flows.

1. Identify *recv* events that are not matched with any *send* event. Such events can occur in environments where the user was unable to identify start events for some activities (e.g., activities of other users). For each unmatched *recv* event, create an artificial *send* event and add this *send* event to set S. This step allows us to create a complete set of start events.

2. Identify pairs of events matched by user-provided directives. Apply Transformation 2 to insert edges between them. This step allows us to handle scenarios where flows switch not on message-passing events.

3. Identify all start events with an in-degree greater than 0. Apply Transforma-

tion 1 to eliminate all edges that are incidental to such nodes. A start event must not be reachable from events in other flows.

4. Identify PDG nodes with in-degree greater than 1. Apply Transformation 1 to eliminate intra-process edges that are incidental to such nodes and that were present in the original PDG. Denote the constructed graph as G'. The pairs of connected nodes in G' satisfy our rules and user-specified directives.

5. For each start event $u \in S$, construct the set of events $R(u, G')$ reachable from u. For different u, sets $R(u, G')$ satisfy the properties of flows.

Overall, our flow-construction approach is most similar to that of DPM [81] and Magpie [11]. As discussed in Chapter 2, there are two main differences of our technique from DPM. First, we support applications that perform intra-request concurrent processing. The message-switch rule attributes events after a *send* to the same flow with the *send*, so that a flow continues on two paths and is represented as a dependence graph. In contrast, DPM assumed that the system only contains inter-request concurrency; events within each request are serialized and represented as strings.

Second, we support applications that may violate one of the rules. Our mapping directives allow the user to provide additional information to the flow-construction algorithm. A similar technique has been used by Magpie. Magpie also constructs a dependence tree similar to a single-flow subgraph of our FPDG. Note however that Magpie does not use this tree to separate events from different requests and instead relies on application-specific rules provided by the user. In contrast, we resort to user's help only when our application-independent rules are insufficient for proper flow construction.

4.5 Glossary of Terminology

- **Basic block:** a sequence of instructions in which flow of control enters at the beginning and leaves at the end without halting or branching
- **Event:** an execution instance of one or more instructions
- **Basic block event:** an execution instance of a basic block
- **Message-passing event:** an execution instance of a special *send* or *recv* instruction
- **Local control flow trace of process p:** the sequence of basic blocks events that occurred in process p, denoted T^p.
- **Parallel dynamic graph, PDG:** directed acyclic graph $G = (N, E)$, where N is a set of nodes representing all basic block events in all processes, and E is a set of edges connecting adjacent events within each process and matching message-passing events across processes.
- **Immediate predecessor:** event a is an immediate predecessor of event b, denoted $a \rightarrow b$, if $(a, b) \in E$.
- **Happened-before relation:** event a happened before event z in the PDG if $a \rightarrow^* z$, where $\rightarrow^*$ is the reflexive transitive closure of the $\rightarrow$ relation.
- **Chop between events u and v in the PDG:** a set of events x, such that x is reachable from u and x reaches v in the PDG. $Chop(u, v) = \{x \in N \text{ s.t. } u \rightarrow^*$

$x, x \rightarrow^* v\}$, where N is the set of all nodes in the PDG.

- **Logical flow:** a set of events that correspond to a semantic activity in the execution. **Start event** for flow Φ, denoted $\Phi.a$ is an event $a \in \Phi$ such that $\forall x \in \Phi, a \rightarrow^* x$. **Stop event** for flow Φ, denoted $\Phi.z$ is an event $z \in \Phi$ such that $\forall x \in \Phi, x \rightarrow^* z$.

- **PDG transformation ψ:** mapping $G \xrightarrow{\psi} G'$, where $G = (N, E)$ is a PDG, $G' = (N, E')$ is a transformed PDG with a modified set of edges $E' \subseteq N \times N$.

Chapter 5

Locating Anomalies and Their Causes

Many bugs and performance anomalies in real-world systems are manifested by unusual control flow: executing functions that are never executed during normal runs or spending time in unusual parts of the code. The root causes of such problems often can be located by collecting a detailed control flow trace for a faulty execution. Control-flow tracing however, may generate large amounts of data. Manual examination of such traces may be difficult, especially in distributed systems, where the data represent control flows of multiple concurrently-executing and communicating processes. In this chapter, we present our techniques for finding the causes of problems via automated trace analysis. In Sections 6.2 and 6.3, we show how to apply these techniques to real-world scenarios.

Our analyses work for systems that demonstrate repetitive behavior during normal execution and deviate from such behavior when a bug or a performance problem occurs. For example, many systems contain distributed groups of identical processes that perform similar activities. Such cross-process repetitiveness is common in cluster management tools [52, 72], Web server farms, and parallel numeric codes. Problems in these systems are often manifested as anomalies: the behavior of one or several processes becomes different from the rest of the group. Other systems demonstrate cross-time repetitiveness: Web and database servers perform a large number of similar requests during the execution. Problems in such systems often cause the processing of a request to differ from the common path.

In our experience, flows constructed with techniques presented in Chapter 4 often represent natural units of repetitive behavior. Different flows can be compared to each other to identify the cause of a problem. Our analysis approach contains two phases. First, we perform *data categorization*: identify anomalous flows, one or several flows that are different from the rest. Such flows may correspond to failed hosts in a cluster or failed requests in a client-server system. Second, we perform *root cause identification*: examine the differences between the anomalous flows and the normal ones to help the analyst identify the causes of the anomalies.

Finding anomalous flows is a non-trivial task in presence of silent and non-fail-stop failures. In this step, we define a *distance (dissimilarity) metric* between pairs of flows. A pair of flows that exhibit a similar behavior should produce a lower distance value than a pair with a dissimilar behavior. We then use this pair-wise metric to construct the *suspect score* for each flow. The further a flow is from the common behavior, the higher its suspect score should be. Flows with the highest suspect scores are the most unusual and we report them to the analyst.

To find the cause of a problem, we determine why a flow was marked as an anomaly. If our algorithm determines that a problem has the fail-stop property, we report the last (or recent) function executed as a likely location of the problem. If a problem has the non-fail-stop property, we identify a function with the highest contribution to the suspect score of the anomalous flow. Depending on how the suspect score is computed, this function may consume an unusual amount of time in the anomalous flow, or it may be present in the anomalous flow and missing from the normal ones. Therefore, the behavior of this function can be regarded as the most visible symptom of the problem. In our experience, such symptoms are often sufficient for finding the root cause of the problem.

In Section 5.1, we describe main classes of problems that can be located with our automated approach. We then present our techniques for a collection of identical processes that perform similar activities. Each flow in such a system corresponds to execution of a single process. Section 5.2 describes how we locate an anomalous process and Section 5.3 describes how we locate the cause of the anomaly in such a process. In Section 5.4, we extend our techniques to handle the general case, where flows can span multiple processes.

5.1 Fault Model

By looking at differences in the control flow across activities, we can find several important classes of problems. Some of these problems are bugs, others are performance problems. While some of these problems could have been located with standard debugging and profiling techniques, others are more difficult to locate. We present a uniform tracing approach that addresses all these problems. For each described problem, we can identify the failed process and the function where the failure happened.

Non-deterministic fail-stop failures. If one process in a distributed system crashes or freezes, its control-flow trace will stop prematurely and thus appear different from those processes that continue running. Our approach will identify that process as an anomaly. It will also identify the last function entered before the crash as a likely location of the failure. These problems can typically be found using distributed debuggers such as TotalView [39].

Infinite loops. If a process does not crash, but enters an infinite loop, it will start spending an unusual amount of time in a particular function or region of the code. Our approach will identify such process and find the function where it spends its time. Finding such problems in sequential applications is often possible by obtaining multiple snapshots of the call stack or single-stepping through the code with an interactive debugger, such as gdb [118]. Such approaches are substantially more complex for large-scale parallel applications, requiring the use of scalable data aggregation and visualization techniques [39, 109, 114].

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

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

Note that there are several types of problems that may not be detected with our approach. First, we may not be able to find massive failures. If a problem happens on all flows in the system, our approach would consider this normal behavior. Fortunately, such problems are typically easier to debug than anomalies. Second, we may not detect problems with no change in control flow. If an application fails on a particular input, but executes the same instructions as in a normal run, we would not be able to locate the root cause of the problem. In our experience however, such problems seem uncommon. Third, we may not be able to locate the causes of latent problems where the fault occurs long before the failure and before the user decides to start tracing. Our techniques, however, may be able to find an early enough symptom of such a problem to help the analyst locate its cause.

5.2 Finding an Anomalous Single-process Flow

Once the control-flow traces are obtained and gathered at a central location, we split them into separate flows and start analyzing the per-flow traces. We begin by presenting our techniques for a scenario where each flow does not cross the process boundaries. We refer to such flows as *single-process flows.* This scenario is common in systems structured as distributed collections of identical processes, such as cluster management tools, Web server farms, and parallel numeric codes. In this section, we present techniques for finding anomalous single-process flows. Section 5.3 describes how we determine the causes of these anomalies. In Section 5.4, we generalize our approach to find anomalies in multi-process flows.

To find anomalous single-process flows, we use two techniques: identification of a process that stopped generating traces first and identification of traces that are the least similar to the rest. The first technique locates processes with fail-stop problems; the second focuses on non-fail-stop problems. Note that none of these techniques use an external fault-detection mechanism. Instead, they locate the misbehaving process from the traces themselves. External fault detectors typically look for known symptoms, such as network time-outs or node crashes [24]. In contrast, our approach is able to find silent failures and failures with symptoms never seen before.

5.2.1 The Earliest Last Timestamp

A simple technique for finding the failed process is to identify the process that stopped generating trace records first. The trace whose last record has the earliest timestamp is reported to the analyst as an anomaly. In general, the technique may be effective for detecting fail-stop problems, such as application crashes or indefinite blocking in system calls. In Section 6.2, we show how it enabled us to find the cause of a problem in the SCore distributed environment [52].

To compare last timestamps across processes on different hosts, we convert them to the absolute time, assuming that the system clocks of all hosts are synchronized. Next, we compute the mean and standard deviation of last timestamps across all processes. If the earliest timestamp is substantially different from the mean and the delta significantly exceeds the attainable clock synchronization precision [83], we assume the fail-stop scenario and report the process with the earliest timestamp to the analyst. Otherwise, we assume that the problem has the non-fail-stop property. Problems such as performance degradations, livelocks, starvation, infinite loops in the code are often harder to debug than fail-stop crashes, and we locate them with the following technique.

5.2.2 Finding Behavioral Outliers

To find anomalous processes that exhibit non-fail-stop behavior, we identify *outlier traces*, i.e. individual traces or small collections of traces that appear different from the rest. To identify such traces, we use a distance-based outlier detection approach [14, 98]. First, we define a *pair-wise distance metric* that estimates the dissimilarity between two traces. Then, we construct a *suspect score* that estimates the dissimilarity between a trace and a collection of traces that we consider normal. We use the suspect score as a *rank* of a trace. Traces with the highest rank are of interest to the analyst as they correspond to processes whose behavior is most different from normal.

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

To eliminate such false positives, our approach uses data from known-correct previous runs, if available. If a previous correct run contained a trace similar to that of the MPI master node in the current run, the master will not be flagged as an outlier. Our analyses provide a uniform framework to handle both the case when we have data only from the failed execution (the *unsupervised* case [47]) and when we also have data from a known-correct previous run (the *one-class ranking* case [126]).

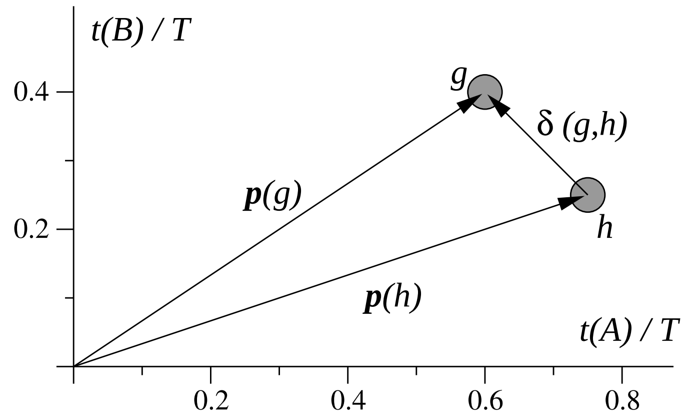


Figure 5.1: *The two data points correspond to processes g and h with profile vectors $\boldsymbol{p}(g)$ and $\boldsymbol{p}(h)$. The coordinates of the points are determined by the relative contributions $t(A)$ and $t(B)$ of functions A and B to the total time T in each process. $\boldsymbol{\delta}(g,h)$ is the difference vector between $\boldsymbol{p}(g)$ and $\boldsymbol{p}(h)$.*

5.2.3 Pair-wise Distance Metric

To define the distance between traces for two processes, g and h, we use a two-step process similar to Dickinson et al. [36]. First, we construct per-trace function profiles that serve as a summary of behaviors in each trace. Then, we define the distance metric between the profiles. The *time profile* for process h is a vector $\boldsymbol{p}(h)$ of length F, where F is the total number of functions in the application.

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

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

The profile vector defines a data point in an F-dimensional space, where the i^{th} coordinate is $t(f_i)/T$, that is the time spent in function f_i as a fraction of the total time. Figure 5.1 shows the geometric interpretation of profiles for an example application that has two functions, A and B (i.e., $F = 2$). Process g has a profile of $(0.6, 0.4)$, representing the fact that it spent 60% of the time in A and 40% in B, and process h has a profile of $(0.75, 0.25)$.

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

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

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

5.2.4 Other Types of Profiles Used

Our algorithms are independent of the type of profile used. Any vector that can characterize the behavior of an application may be useful for finding anomalies and possibly explaining their causes. In studies described in Chapter 6, we have used time profiles, coverage profiles, and communication profiles. Time profiles proved effective for finding the cause of the problem described in Section 6.2. Unlike other common types of profiles, such as count and coverage profiles, time profiles are suitable for detecting indefinite blocking in system calls, infinite loops, and performance problems.

Our definition of time profiles can include *call path profiles*. We use the term *call path to an invocation of function* X to represent the sequence of function calls from *main* to the invocation of X. That is, a call path to X corresponds to functions currently on the call stack when X is called. A call path to X is *complete* if this invocation of X returns without making function calls; a call path to X is *incomplete* if this invocation of X makes at least one function call.

To construct call path profiles, we treat different call paths to each function as separate functions. Assume that an execution trace h contains two paths to function C: $(main \rightarrow A \rightarrow C)$ and $(main \rightarrow B \rightarrow C)$, where $(A \rightarrow C)$ denotes a call from A to C. We can consider these two paths as two different functions, f_1 and f_2. The time $t(h, f_1)$ attributed to f_1 is equal to the time spent in C when it was called from A when A was called from *main*, and the time $t(h, f_2)$ is equal to the time spent in C when it was called from B when B was called from *main*. 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. The experiments in Sections 6.2, 6.3, and 6.4 use the path-based method due to its higher accuracy.

While time profiles often are useful for finding anomalies and their causes, some problem causes proved difficult to detect from time profiles. In Section 6.3 for example, we describe a problem where a short-running function was not called in the anomalous flow — the application continued running but produced incorrect results. The time spent in this function was only a small fraction of the total run time of the application. Therefore, time profiles could not identify the absence of this function as the root cause of the anomaly.

To address this limitation of time profiles, we can use *coverage profiles*. A coverage profile is a boolean vector such that the i^{th} element of the vector determines whether function f_i is present in the trace. For time profiles, the absolute value of delta between an element of one profile and an element of another profile depends on the fraction of the time spent in the corresponding function. For coverage profiles, the absolute value of delta is always 0 or 1. That is, all differences between elements of two coverage profiles have equal contribution to the distance metric. Therefore, unlike time profiles, coverage profiles may locate some problems that do not affect the execution time significantly.

The behavior of a distributed application often can be characterized by its communication activities. In Section 5.4.2, we show how to summarize such activities in a *communication profile*. Any other metric that can be normalized to be included in our profiles also can be used. For example, the number of cache misses in each function, the number of floating-point operations, or I/O statistics can be used to summarize the behavior of an application. Determining the effectiveness of these metrics for problem diagnosis remains future work.

5.2.5 Suspect Scores and Trace Ranking

Our goal is to compute a suspect score for each trace. Traces with the largest scores will be of primary interest to the analyst as probable anomalies. We present two algorithms for computing such scores. The first algorithm applies to the unsupervised case where we have trace data from only the failed execution. The second algorithm applies to the one-class classifier case [126] where we have additional data from a known-correct previous run. This additional data allows us to identify anomalies with higher accuracy. Both algorithms are based on the same computational structure.

In the unsupervised case, we operate on a set of traces U collected from the failed run. We assume that U consists of one or more types of common behaviors and a small number of outliers. Ramaswamy et al. detected outliers by ranking each data point according to its distance to its k^{th} nearest neighbor [98]. We use this idea to compute our suspect scores in the unsupervised case and extend it to handle the one-class classified case.

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

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

$$U_d(h) = \langle h_1, h_2, \ldots, h_{|U|} \rangle \tag{5.3}$$

Here, $d(h, h_i) \leq d(h, h_{i+1}),\ 1 \leq i \leq |U|$. The suspect score for trace h is then defined

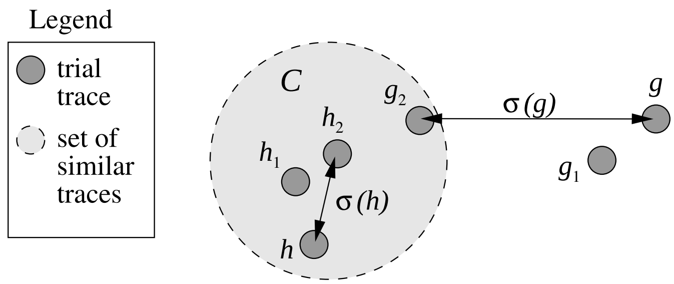


Figure 5.2: *Computing the suspect scores $\sigma(g)$ and $\sigma(h)$ for an anomalous trace g and normal trace h. C is a large collection of traces with similar behaviors; g_i and h_i are the i^{th} trial neighbors of g and h respectively.*

as the distance of h to its k^{th} nearest neighbor h_k:

$$\sigma(h) = d(h, h_k) \tag{5.4}$$

A trace is considered normal if it has a low suspect score and an anomaly if it has a high suspect score. Informally, $\sigma(h)$ tries to quantify the dissimilarity between h and the nearest common behavior, since we consider all common behaviors as normal. Parameter k determines what is a common behavior: a trace must have at least k close neighbors to receive a low suspect score and be considered normal.

Figure 5.2 shows two examples to illustrate the algorithm when $k = 2$: Trace h is part of a large set C, representing one of the common behaviors; h has a large number (more than $k = 2$) of close neighbors. As a result, its k^{th} neighbor, h_2, will be relatively close, the suspect score $\sigma(h) = d(h, h_2)$ will be low, and h will be considered a normal trace. In contrast, trace g is an outlier. It is far away from any common behavior and has only one (i.e., less than k) close neighbor. As a result, its k^{th} neighbor, g_2, will be far away, the suspect score $\sigma(g) = d(g, g_2)$ will be high, and g will be considered an anomaly.

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

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

Similar to sequence $U_d(h)$ defined by Equation 5.3, sequence $N_d(h)$ arranges all traces in N in the order of their distance to h:

$$N_d(h) = \langle n_1, n_2, \ldots, n_H \rangle \tag{5.5}$$

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

$$\sigma(h) = \min\{d(h, h_k),\ d(h, n_1)\} \tag{5.6}$$

Figure 5.3 shows two examples to illustrate this idea. Similar to Figure 5.2, trace h will be considered normal, as it is part of a large collection of trial traces. Trace g is an outlier. It is far away from any common behavior and has only one (i.e., less

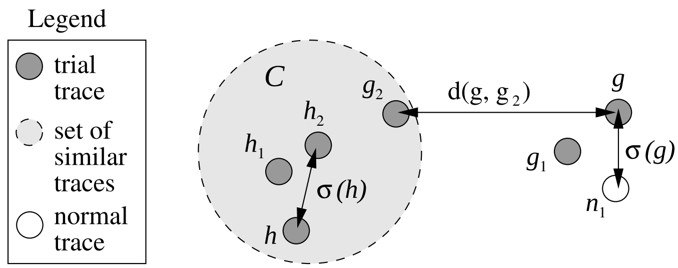


Figure 5.3: *Computing the suspect scores $\sigma(g)$ and $\sigma(h)$ where both g and h are normal, but g is unusual; n_1 is the closest normal neighbor of g.*

than k) close trial neighbor. However, it has a normal trace n_1 nearby. Therefore, its suspect score $\sigma(g) = d(g, n_1)$ will be low, and g will also be considered normal.

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

5.3 Finding the Cause of the Anomaly

Once the anomalous trace is found, we have developed several techniques for locating a function that is a likely cause of the problem. To define the concepts used in this section, we consider the following steps of problem evolution [6]. The first step is the occurrence of a *fault*, also referred to as the *root cause* of a problem. In the second step, the fault causes a sequence of changes in the program state, referred to as *errors*. Finally, the changes in the state cause the system to fail: crash, hang, or otherwise stop providing the service. We refer to such an event as a *failure*.

5.3.1 Last Trace Entry

A simple technique for identifying a function that caused the found process to behave abnormally is to examine the last entry of the process's trace. The technique can be viewed as a natural extension of our earliest last timestamp approach from Section 5.2.1. We identify not only the process that stopped generating trace records first, but also the last function executed by that process. Such a function may be a likely cause of the failure. The technique is often effective for identifying the causes of crashes and freezes, but may not work for non-fail-stop problems.

5.3.2 Maximum Component of the Delta Vector

To locate the causes (or at least symptoms) of non-fail-stop problems, we developed an approach that is a natural extension to the outlier-finding approaches described in Section 5.2.2. After locating outlier traces, we identify the symptoms that led us to declare those traces as outliers. Specifically, we find a function whose unusual behavior had the largest contribution to the suspect score for the outlier trace.

Recall that in Equations 5.4 and 5.6, the suspect score of a trace h is equal to the distance $d(h, g)$ of h to another trace g, where g is either the k^{th} trial neighbor of h or the first known-normal neighbor of h. In turn, the distance $d(h, g)$ is the length of the delta vector $\boldsymbol{\delta}(h, g)$. Component δ_i of $\boldsymbol{\delta}(h, g)$ corresponds to the contribution of function f_i to the distance. Therefore, by finding δ_i with the maximum absolute value, we identify a function with the largest difference in behavior between h and g:

$$anomFn = \operatorname*{argmax}_{1 \leq i \leq F} |\delta_i| \tag{5.7}$$

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

5.3.3 Anomalous Time Interval

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

5.3.4 Call Coverage Analysis

By analyzing time profiles, we are able to find the most visible symptom of an anomaly. Often, such symptoms are sufficient for finding the root cause of the anomaly. As described in Section 5.2.4 however, some anomalies are difficult to distinguish from normal behavior using time profiles. Even if the anomalous flows are correctly identified, the root causes of some anomalies are not easily located using time profiles.

These causes often can be located by analyzing coverage profiles, as described below.

Consider an intermittent problem where an allocated memory region is freed twice in some executions, corrupting memory and causing a later crash. Assume that in other executions, the memory region is freed once and the application terminates normally. We can distinguish normal executions from the anomalies since time profiles for crashed executions are likely to be different from the rest. However, determining that the cause of the problem is the additional call to *free* may be difficult. The time spent in that call is only a small fraction of the total run time of the application.

If this call to *free* was made from an unusual code location that is never reached in normal runs, we can detect that it is correlated with the occurrence of the problem by using coverage profiles. We construct call path coverage profiles by replacing non-zero elements in call path time profiles with the number one and leaving zero elements unmodified. Then, we apply our standard anomaly detection algorithm described in Section 5.2 to identify anomalous profiles, i.e., profiles with unusual call path coverage.

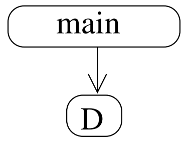
Next, we construct the set Δ_a containing call paths that are present in the anomalous profiles and absent from all normal ones, the set Δ_n containing call paths that are present in the normal profiles and absent from all anomalous ones, and their union $\Delta = \Delta_a \cup \Delta_n$. Paths in Δ are correlated with the occurrence of the anomaly. Since the second *free* was called from an unusual code location and was present only in anomalous runs, the corresponding call path will be included in the constructed set. Therefore, by inspecting all paths in Δ, the analyst could find the root cause of the problem.

In our experience however, the number of call paths in Δ is often large. In addition to the call path corresponding to the root cause of the problem, Δ may contain subsequent symptoms of the same problem. We refer to such call paths as *problem-induced variations in coverage*. Δ may also contain unrelated paths that are caused by slight differences in system load or program input between normal and anomalous flows. We refer to such call paths as *normal variations in coverage*. Distinguishing problem-induced variations from normal variations requires user help. However, techniques presented here can eliminate some variations of each type automatically thus reducing the number of paths that must be inspected manually.

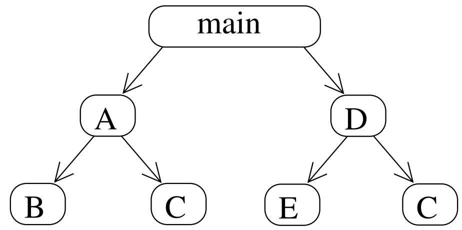
In the study discussed in Section 6.3, set Δ contained more than 21,000 differences, i.e., call paths present in the normal flows and absent from the anomalous ones or vice versa. Our techniques eliminated differences that were mere effects of earlier differences in coverage. This step reduced the number of call paths to 107. To simplify further examination, our techniques ranked the remaining call paths by the time of their first occurrence in the trace. By inspecting the source code for each call path in the decreasing order of their rank, we eliminated the 14 earliest paths. The 15^{th} call path proved to be the cause of the problem. Below, we provide a more detailed discussion of our techniques for constructing set Δ, eliminating some paths from Δ, and ranking the remaining paths to simplify their manual examination.

Identifying Differences in Coverage

Consider an example single-process application that contains an intermittent problem. Some runs of this application complete successfully, the others crash. Each run corresponds to a separate flow. Assume that the anomalous flow contains four call paths that are complete, as defined in Section 5.2.4: $main \rightarrow A \rightarrow B$, $main \rightarrow A \rightarrow C$, $main \rightarrow D \rightarrow E$, and $main \rightarrow D \rightarrow C$; the normal flow contains only one complete



a: *Normal flow*



b: *Anomalous flow*

Figure 5.4: *Prefix trees for call paths in normal and anomalous flows.*

call path: $main \to D$.

We begin by constructing sets $Anom$ and $Norm$ to represent call paths observed in anomalous and normal flows respectively. Each set is constructed as a *prefix-closure* [99] of the set of complete call paths: it contains all complete call paths and all shorter prefixes of these paths (i.e., all incomplete paths). In our example, $Norm = \{main, main \to D\}$ and $Anom = \{main, main \to A, main \to A \to B, main \to A \to C, main \to D, main \to D \to C, main \to D \to E\}$. Including incomplete call paths in $Anom$ and $Norm$ allows us to identify the first unusual call rather than a consequence of this call (e.g., focus on why $main$ called A only in the anomalous flow rather than focusing on why A later called B).

Similar to the *CPProf* profiler [45], we visualize $Anom$ and $Norm$ as *prefix trees* [59]. Figure 5.4 shows prefix trees constructed for our example. Each tree node represents a distinct function invocation. A branch from node A to node B indicates that function A invoked function B. We consider two invocations of the same function distinct if they occurred on different call paths. In Figure 5.4b for example, there are two nodes corresponding to function C: C was invoked on $main \to A \to C$ and

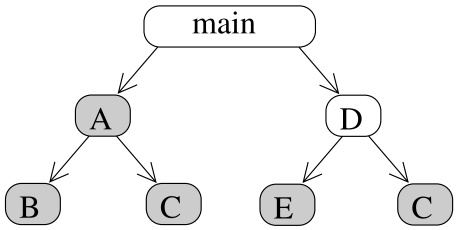
$main \rightarrow D \rightarrow C$. The $main$ function is the root node of the tree. Since $Norm$ and $Anom$ are prefix-closed [99], the path from the root to any node in the tree corresponds to a call path (complete or incomplete) in $Norm$ or $Anom$.

Next, we compute a set difference $\Delta_a = Anom \setminus Norm$ between $Anom$ and $Norm$. Δ_a contains all paths present in the anomalous flow and absent from the normal flows. Determining why these call paths were not taken in normal flows often allows us to find the cause of the problem. In our example, $\Delta_a = \{main \rightarrow A, main \rightarrow A \rightarrow B, main \rightarrow A \rightarrow C, main \rightarrow D \rightarrow E, main \rightarrow D \rightarrow C\}$, as shown in Figure 5.5a. Similarly, we construct set $\Delta_n = Norm \setminus Anom$ to represent call paths present in all normal flows and absent from the anomalous one. Determining why these paths are not taken in the anomalous flow also may point to the cause of the problem. In our example however, Δ_n is empty. Below, we show how to localize the root cause of the problem on the example of Δ_a; the same techniques also apply to Δ_n.

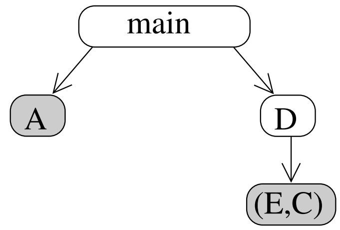
Unlike sets $Norm$ and $Anom$, Δ_a is not prefix-closed [99]: if a path π is in Δ_a, its prefixes may or may not be in Δ_a. In our example, Δ_a contains path $main \rightarrow D \rightarrow C$ but not its shorter prefixes $main \rightarrow D$ and $main$. As a result, if we visualize Δ_a as a prefix tree, some incomplete paths in the tree may not correspond to call paths in Δ_a. To accurately represent Δ_a with a prefix tree, we show the last call for each path in Δ_a as a gray node in the tree. In Figure 5.5a for example, node A is gray since the path $main \rightarrow A$ is in Δ_a (observed only in the anomalous flow); node D is white since the path $main \rightarrow D$ is not in Δ_a (observed in both normal and anomalous flows).

Eliminating Effects of an Earlier Difference

Multiple call paths in the tree representing Δ_a can have a single cause. We



a: *Original tree*



b: *Transformed tree*

Figure 5.5: *Prefix trees for Δ_a before and after the transformations. All call paths in Δ_a end at gray nodes in the tree. Call paths that end at white nodes are present in both normal and anomalous flows.*

attempt to retain one path for each cause by using two transformations of set Δ_a. First, if an incomplete path π_1 is part of Δ_a, then any longer path π_2 that contains π_1 can be removed from Δ_a thus producing Δ'_a:

$$\text{if } \pi_1, \pi_2 \in \Delta_a, \pi_2 = \pi_1 x, \text{ then } \Delta'_a = \Delta_a \setminus \{\pi_2\} \tag{5.8}$$

Here, $\pi_2 = \pi_1 x$ denotes that π_2 is a concatenation of π_1 and one or more calls referred to as *suffix* x. In our example, $main \rightarrow A$ is part of Δ_a, i.e., $main$ called A in the anomalous flow but not in any normal ones. If we explain why $main$ called A only in the anomalous flow, we will likely explain why each longer path that contains $main \rightarrow A$ is present only in the anomalous flow. Therefore, the entire subtree rooted at A can be removed from the tree. In contrast, the path $main \rightarrow D$ was present in both normal and anomalous flows. Therefore, its subtree must not be removed.

Second, we merge call paths in Δ'_a that differ only in the last call (the leaf of the tree) as such paths typically correspond to a single cause. If paths $\pi_1, \pi_2 \in \Delta'_a$ have a common prefix π: $\pi_1 = \pi u$, $\pi_2 = \pi v$, where u and v are leaves, then we replace them in Δ'_a with a merged path $\pi_3 = \pi w$, where w is a new composite leaf, $w = (u, v)$:

$$\Delta''_a = \Delta'_a \cup \{\pi_3\} \setminus \{\pi_1, \pi_2\} \tag{5.9}$$

Since π is a prefix of π_1 and π_2, then $\pi \notin \Delta'_a$. Otherwise, the subtree rooted at π would have been pruned by the first transformation. Therefore, the leaf nodes in Δ'_a distinguish the call paths in the anomalous flow from the normal ones. However, the reason why a leaf function was invoked is located in its parent function, thus making the parent, not the leaf, a potential problem location, and allowing us to merge the leaves together. If the application exhibited more than a single problem, and if the problems manifest themselves separately in end leaf functions, then this optimization might hide that situation. However, we believe this case to be quire rare and the merging optimization to be widely beneficial.

After applying both transformations, we inspect the paths in Δ''_a manually. For each path, we use the source code of the system to determine why the last call in the path was made by its parent function in the anomalous flow but not in the normal flows. That is, we explain why each leaf is present in the tree for Δ''_a. Figure 5.5b shows this tree for our example application. To obtain this tree, our first technique pruned the subtree of function A. Our second technique merged the children of D into a single composite node (E, C). To locate the root cause of the problem, the analyst would need to examine the source code for *main* to determine why function A was called only in the anomalous flow, and the source code for D to determine why E and C were called only in the anomalous flow.

Ranking the Remaining Call Paths

In the current example, our techniques reduced the number of call paths to inspect from 5 to 2. In the real-world studies discussed in Sections 6.3 and 6.4, the reduction was a factor of 26–200. However, the number of remaining call paths may still be large and identifying the path corresponding to the root cause of a problem may be difficult. We further simplify call path examination with two ranking techniques. We arrange the paths from the transformed Δ_a and Δ_n into a single list, mark each path to indicate whether it belongs to Δ_a or Δ_n, and order the list to estimate the importance of each path for problem diagnosis.

The first technique arranges the call paths in the order of their first occurrence in the trace. It assumes that the earlier differences between anomalous and normal flows are more likely to correspond to the root cause of a problem. This assumption holds for problem-induced variations in coverage (since all symptoms happen after the root cause). However, it may be violated for normal variations (a change in user input may introduce unrelated coverage differences that happen before the root cause of the problem). Distinguishing between problem-induced and normal variations requires user help. Therefore, when applied to a collection of problem-induced and normal variations, our automated ranking may order some normal variations before the root cause. In Section 6.3, we eliminated 14 such variations via manual examination.

The second technique orders shorter call paths before longer ones. It assumes that shorter paths are more likely to correspond to the root cause of a problem. In practice, shorter paths often represent more substantial differences in the execution than longer paths. Whether more substantial differences are more likely to correspond to the root cause remains to be seen. In Section 6.3, this technique required manual examination of 30 other differences before the root cause. Both rankings can be combined to order the shortest paths first and then arrange the paths of the same length earliest-first. Determining which ranking technique works better in most environments remains future work.

5.4 Finding an Anomalous Multi-process Flow

The techniques presented thus far in this chapter worked well for flows that do not cross the process boundary. Here, we generalize our approach to handle flows that involve multiple processes. In our system, such flows are represented as disjoint subgraphs of the PDG, as discussed in Chapter 4. Our general approach to finding the cause of a problem uses the same two steps as for single-process flows: we identify an anomalous flow and then determine why it was marked as an anomaly. The algorithms used to perform these steps are also unchanged: we summarize each flow as a fixed-length vector (a profile), define a distance metric between pairs of profiles, find a profile that is most different from the rest, and determine why it was considered different.

The key distinction between our solutions for single-process and multi-process flows is how we construct a profile for a flow. For single-process flows, we used call time profiles, as defined by Equation 5.1. In studies described in Section 6.2, such profiles accurately represented the function call behavior of a process. To summarize multi-process flows, we use *composite profiles.* Such profiles capture both the function call behavior and the communication structure of a distributed application. The addition of the communication structure allows us to detect anomalies that cause little change in the function call behavior.

To construct a composite profile, we concatenate two vectors: a *multi-process call profile* and a *communication profile.* The multi-process call profile captures the function call behavior of processes in a flow. In Section 5.4.1, we show how to construct such profiles by generalizing single-process call profiles. The communication profile captures the communication structure of a flow. We show how to construct such profiles in Section 5.4.2. Once the composite profiles are obtained, anomaly detection techniques described previously can be used to find anomalous flows and determine why the flows are anomalous.

For multi-process flows, we do not use the Earliest Last Timestamp technique discussed in Section 5.2.1. This technique works well for collections of long-running identical processes. If one of the processes stopped producing trace records substantially earlier than the others, we consider it a symptom of a fail-stop problem. However, this technique is less applicable to a client-server system, the typical environment with multi-process flows. Such systems exhibit cross-request rather than cross-process repetitiveness. Requests (flows) in such systems terminate at different times even under normal conditions. Therefore, the last timestamps for individual flows cannot be compared to each other. Below, we focus on the distance-based approach for finding anomalous flows.

5.4.1 Multi-process Call Profiles

A *multi-process call profile* for a flow Φ is a vector containing F elements, where F is the total number of functions in all binaries of the application. Each element of the vector corresponds to a function in one of the binaries. We evaluated two variations of call profiles: *time profiles* and *coverage profiles*. The time profile $\boldsymbol{p}^t(\Phi)$ of a flow Φ is defined similar to the single-process flow case:

$$\boldsymbol{p}^t(\Phi) = \left(\frac{t(\Phi, f_1)}{T(\Phi)}, \ldots, \frac{t(\Phi, f_F)}{T(\Phi)}\right), \text{ where } T(\Phi) = \sum_{i=1}^{F} t(\Phi, f_i) \tag{5.10}$$

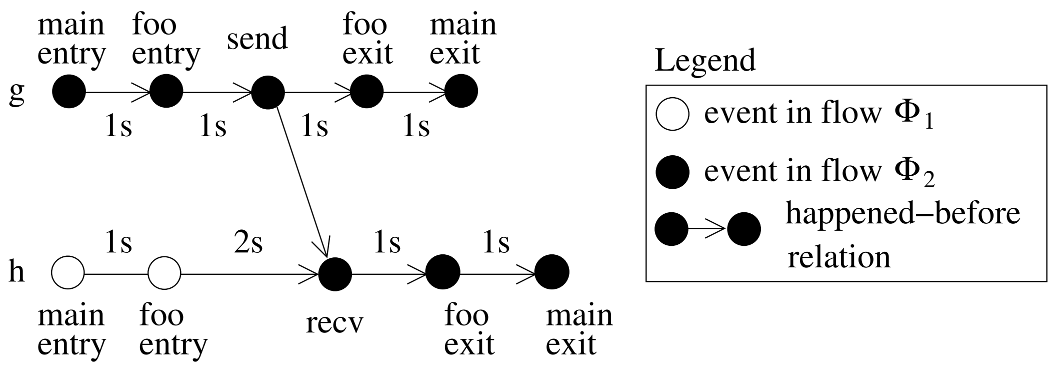


Figure 5.6: *Dividing the time spent in a function among two flows*

The i^{th} element of the time profile is the time spent in function f_i while executing on flow Φ normalized by the total time spent on Φ.

The coverage profile $\boldsymbol{p}^v(\Phi)$ of a flow Φ is defined as:

$$\boldsymbol{p}^v(\Phi) = (v_1, \ldots, v_F)\,, \text{ where } v_i = \begin{cases} 1, & f_i \text{ was invoked on flow } \Phi \\ 0, & \text{otherwise} \end{cases} \tag{5.11}$$

The i^{th} element of the the coverage profile is a boolean value determining whether function f_i was invoked on flow Φ. As discussed in Section 5.3.4, some anomalies are more accurately identified using time profiles, others are more accurately identified using coverage profiles. However, a time profile can be used to obtain a coverage profile by replacing all non-zero elements with 1. Below, we focus on the more general techniques of constructing time profiles.

Time profiles computed for multi-process flows are different from those for single-process flows in two respects. First, multiple processes executing on the same flow can now contribute to the flow's profile vector. If these processes correspond to different binaries, they affect different elements of the vector. However, if these processes correspond to the same binary, the time for each function in that binary is the sum of times spent in that function by each process.

Second, if a process switches from working on flow Φ_1 to working on flow Φ_2 while in function *foo*, the time spent in *foo* must be correctly distributed between Φ_1 and Φ_2. Consider an example shown in Figure 5.6, where processes g and h execute the same binary with functions *main* and *foo*. Process h switches from flow Φ_1 to flow Φ_2 on a *recv* call. The time from the *foo*'s entry to the *recv* event (2 seconds) should be attributed to *foo* on flow Φ_1; the time from the *recv* event to the exit of *foo* (1 second) should be attributed to *foo* on flow Φ_2.

Overall, on flow Φ_1, *main* spends 1 second (before entering *foo* in process h) and *foo* spends 2 seconds (before calling *recv* in process h). On flow Φ_2, *main* spends 3 seconds (1 second before entering *foo* in process g, 1 second after exiting *foo* in process g, and 1 second after exiting *foo* in process h) and *foo* spends 3 seconds (2 seconds in process g and 1 second in process h after *recv*). The durations of flows Φ_1 and Φ_2 are: $T(\Phi_1) = t(main, \Phi_1) + t(foo, \Phi_1) = 3$ and $T(\Phi_2) = t(main, \Phi_2) + t(foo, \Phi_2) = 6$. The time profiles are: $\boldsymbol{p}^t(\Phi_1) = (1/3, 2/3)$ and $\boldsymbol{p}^t(\Phi_2) = (3/6, 3/6) = (1/2, 1/2)$.

In our experience, time profiles accurately summarize the control-flow behavior of individual flows and often can be used to find anomalies and their causes. Some anomalies however, may not result in a substantial change to time profiles. For example, consider a system with two processes that can deadlock. After the deadlock, both processes are blocked in the *select* system call, waiting for messages from each other. During normal execution under light load, each process also spends most time blocked in the *select* system call, waiting for external requests. In this scenario, the call time profile for the deadlocked flow will be similar to normal and we may not mark this flow as an anomaly.

5.4.2 Communication Profiles

To improve the accuracy of anomaly detection in distributed systems, we augment call time profiles with summaries of communication behavior. Such summaries allow us to detect anomalies that include a change in the communication behavior rather than only the function call behavior of an application. For a given flow Φ, we construct a *communication profile vector* $\boldsymbol{p}^c(\Phi)$ of length F, where F is the total number of functions in all binaries of the application:

$$\boldsymbol{p}^c(\Phi) = \left(\frac{s(\Phi, f_1)}{S(\Phi)}, \ldots, \frac{s(\Phi, f_F)}{S(\Phi)}\right), \text{ where } S(\Phi) = \sum_{i=1}^{F} s(\Phi, f_i) \tag{5.12}$$

The i^{th} element of the communication profile is $s(\Phi, f_i)$, the number of bytes sent by function f_i on flow Φ, normalized by $S(\Phi)$, the total number of bytes sent on Φ. This definition is structurally identical to the definition of time profiles, as introduced by Equation 5.10. In our experience, such profiles are useful for detecting anomalies; they also provide detailed function-level data useful for locating the cause of the anomaly.

Recall that our time profiles can be computed for functions or call paths in the execution. For communication profiles, the call path-based approach is substantially more useful that the function-based one. Partitioning the number of bytes sent by function is ineffective, since most communications occur in one or several standard functions, such as *send* and *write*. If an application calls *send* from several locations, profile elements for all functions except *send* would remain zero. In contrast, the call path approach attributes data sent from different locations in the code to different components of the communication profile and provides a detailed account of communication activities by code location.

To represent the number of bytes received rather than sent by each function, we can introduce *receiver-side communication profiles*. Since matching *send* and *recv* operations always belong to the same flow, the number of bytes sent by each flow is equal to the number of bytes received. Therefore, receiver-side communication profiles are likely to be equally effective for detecting anomalies in communication behavior. Below, we assume that communication profiles represent the number of bytes sent by each function, as defined by Equation 5.12.

5.4.3 Composite Profiles

After constructing a call time profile $\boldsymbol{p}^t(\Phi)$ and a communication profile $\boldsymbol{p}^c(\Phi)$ for a flow Φ, we concatenate them to obtain a composite profile, a single vector $\boldsymbol{p}(\Phi)$ that represents both the function-call behavior and the communication behavior of the flow. The length of $\boldsymbol{p}(\Phi)$ is $2P$, where P is the total number of observed call paths in all binaries in the application:

$$\boldsymbol{p}(\Phi) = \left(p_1^t(\Phi), \ldots, p_P^t(\Phi), \quad p_1^c(\Phi), \ldots, p_P^c(\Phi)\right)$$

Once the composite profiles are obtained, anomaly detection techniques previously described in Section 5.2 can be used to find anomalous flows. Such techniques are independent of the type of profile used. To determine why these flows are anomalous, Section 5.5 generalizes techniques previously described in Section 5.3. Sections 6.3 and 6.4 show how we used these techniques to find two real-world problems in the Condor system.

5.5 Finding the Cause of a Multi-process Anomaly

After locating an anomalous multi-process flow Φ_a, we focus on finding the cause of the anomaly. Our approach is similar to that used for single-process flows, as described in Section 5.3. Namely, we determine why our anomaly detection algorithm marked a given flow as anomalous. That is, we determine why the suspect score of flow Φ_a was higher than that of all other flows. According to Equations 5.4 and 5.6, the suspect score $\sigma(\Phi_a)$ is the length of the delta vector $\boldsymbol{\delta} = \boldsymbol{p}(\Phi_a) - \boldsymbol{p}(\Phi_n)$ between the profile $\boldsymbol{p}(\Phi_a)$ and the profile $\boldsymbol{p}(\Phi_n)$ for another flow Φ_n that is considered normal. Therefore, the component δ_m with the maximum absolute value corresponds to the largest difference between Φ_a and Φ_n.

Unlike the single-process case, we now operate on composite profiles containing both time profiles and communication profiles. Therefore, δ_m may correspond to the largest difference either in the call time behavior or in the communication behavior between Φ_a and Φ_n. If the difference is in the call time behavior, δ_m corresponds to a call path that consumes an unusual amount of time on Φ_a compared to the other flows. If the difference is in the communication behavior, δ_m corresponds to a call path sending an unusual amount of data on Φ_a. Typically, by determining why these call paths were executed, we can locate the root cause of the problem.

Similar to the scenario described in Section 5.3.4, some anomalies in multi-process flows cannot be located by examining time or communication profiles. However, if the call path corresponding to the cause of a problem is not present in normal flows, we can identify it by using call coverage profiles. We construct a coverage profile for each flow, identify anomalous profiles, and locate call paths that are present in all anomalous profiles and absent from all normal profiles or vice versa.

To reduce the number of paths to inspect, we represent them as a forest, a collection of prefix trees for different processes in the flow and transform the trees, as described in Section 5.3.4. These transformations can be applied to the prefix tree of each process independently. Next, we extract the remaining paths from each tree and use our ranking techniques to order these paths by their length or by the time of occurrence. To order call paths in different processes by the time of occurrence in systems without a global clock, we can use a total ordering consistent with the partial happened-before ordering [61]. These techniques proved effective for finding the causes of two problems described in Sections 6.3 and 6.4.

Chapter 6

Experimental Studies

To evaluate the effectiveness of our techniques, we applied them to finding the causes of several problems in complex software. Some of these problems were software bugs, others were performance problems. Some of them have been found previously by system developers, others represented new problems. In all scenarios, our tracing approach allowed us to locate problem causes with little knowledge of system internals. Some of the causes were found in the application code, others were in the kernel.

Depending on the target environment, we categorize our studies in three classes. The first class contains problems that we detected in single-process applications. These early studies evaluated the effectiveness of self-propelled instrumentation for on-demand data collection within a process and across the kernel boundary. The second class contains problems that we detected in distributed collections of identical processes. These studies evaluated the effectiveness of our techniques for automated analysis of single-process flows. The third class contains a problem that we detected in a distributed system where flows spanned several processes. This study evaluated the effectiveness of our cross-process propagation, separation of activities from different flows, and automated analysis of multi-process flows.

6.1 Single-process Studies

To evaluate our techniques for propagation within a process and across the process-kernel boundary, we applied them to analysis of two performance problems: choppy DVD playback in *MPlayer* and non-responsive GUI in *Microsoft Word* running under the *Wine* emulator, and also locating a system crash. These three studies primarily demonstrate the tracing capabilities of self-propelled instrumentation. The data analysis in this section is performed by hand using trace visualization.

6.1.1 Analysis of a Multimedia Application

MPlayer [88] is a popular multimedia player for Linux and several other platforms. It supports a wide variety of video and audio formats and works well even on low-end consumer systems. However, its quality of DVD playback on some systems was not adequate — the player often dropped frames, resulting in user-visible freezes in the playback. Here, we demonstrate how we used self-propelled instrumentation to identify the root cause of this problem.

To investigate the problem, we used *MPlayer* (Version 1.0pre2) to display a short DVD trailer on a low-end Pentium II 450 MHz desktop computer running Linux. The system satisfies typical hardware requirements for DVD playback [35]. We verified that the hardware capabilities of the machine were adequate for the task: the trailer displayed accurately under two commercial DVD players in Microsoft Windows. At the same time, playback under Linux was choppy and *MPlayer* reported frames being dropped.

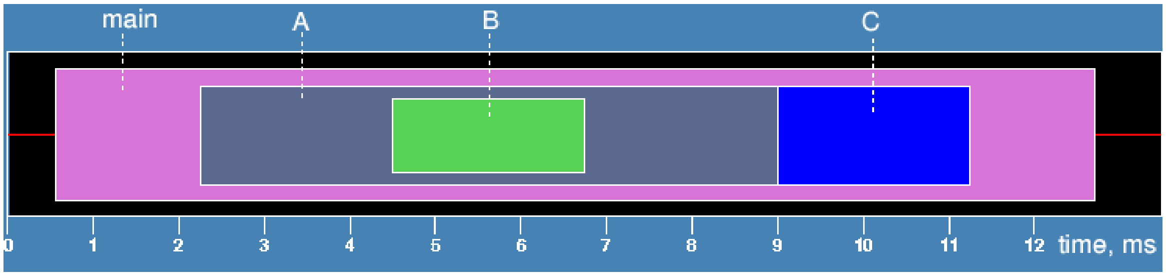


Figure 6.1: *A sample trace of functions*

To find the cause of the problem, we subjected *MPlayer* to self-propelled tracing as follows. When the player starts, we pre-load the tracer library into it using the *LD_PRELOAD* mechanism. When the library initializes, it requests an alarm signal to be delivered to it 10 seconds later. The signal arrives when *MPlayer* is performing playback. The handler in the tracer library catches the signal and initiates instrumentation. It collects the trace for 200 milliseconds (several frames of the trailer), removes the instrumentation, and lets the test run to completion. When *MPlayer* terminates, we save the trace to disk for later visualization.

To visualize function-level traces, we use the *Nupshot* tool [49], whose format is illustrated by the example in Figure 6.1. In this example, there are four functions: *main*, *A*, *B*, and *C*. Each function in the trace is represented by a rectangle. The width of the rectangle is the duration of the function's execution in milliseconds. Nested rectangles represent nested function calls. Function *main* calls function *A*, *A* calls *B*, *B* returns to *A*, *A* returns to *main*, and then *main* immediately calls *C*.

Figure 6.2 shows a fragment of the user-level trace of *MPlayer* corresponding to processing four video frames. We annotated the trace to show the boundaries of each frame and the names of several functions discussed below. We determined frame

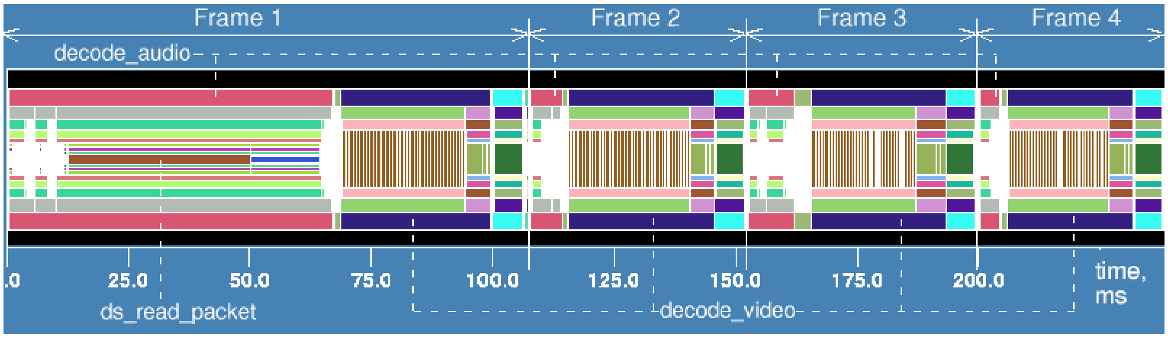


Figure 6.2: *An annotated user-level trace of MPlayer*

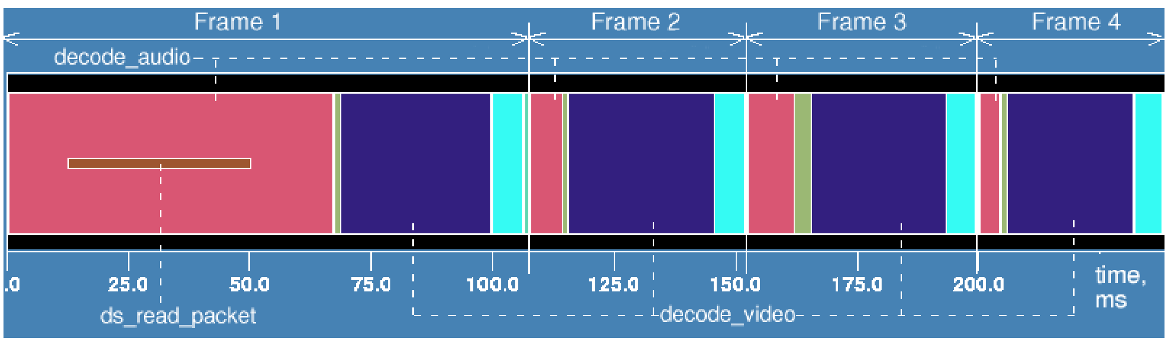


Figure 6.3: *A user-level trace of MPlayer showing functions called directly from main and the anomalous call to ds_read_packet*

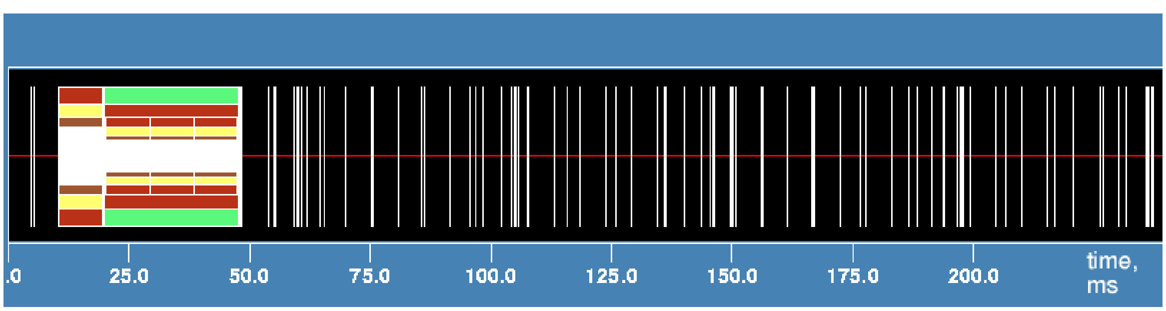


Figure 6.4: *Kernel activity while MPlayer is running*

boundaries using the *MPlayer* source code to identify functions that start processing each video frame. To simplify trace interpretation, Figure 6.3 displays functions called directly from *main* and a few other relevant details. Note that the timing of events in the first frame is clearly different from the rest of the frames. Frames 2, 3, and 4 spend on average 7 milliseconds in *decode_audio* and more than 27 milliseconds in *decode_video*. While Frame 1 spends a similar amount of time (31 milliseconds) in *decode_video*, it spends significantly more (66 milliseconds) in *decode_audio*.

The trace reveals that more than 50% of the time spent in *decode_audio* is attributed to a single call to the *ds_read_packet* function. This finding is surprising for two reasons. First, the trace shows that the time was spent in *ds_read_packet* itself, not in its callees. Second, the time was not spent executing in a loop. Although the binary code of *ds_read_packet* does contain one loop, it calls *memcpy* on every iteration and the trace did not contain repetitive calls to *memcpy*.

The long delay in a short straight-line code sequence suggests that *MPlayer* might be preempted in this function. After collecting additional instances of the trace, we saw similar long delays in other parts of the code, both expected (e.g., the *read* system call) and surprising (e.g., in calls to the *malloc* memory allocation routine). To verify the preemption hypothesis, we use the results of kernel tracing. Again, to hide irrelevant details, we capture only preemption events by starting kernel instrumentation from the *do_IRQ* kernel entry point. Figure 6.4 shows that the system is mostly idle, except periodic timer interrupts triggered every 10 milliseconds. However, there is a burst of kernel activity right at the time when *ds_read_packet* was executing. Therefore, the time attributed to *ds_read_packet* was spent in the kernel, handling hardware interrupts.

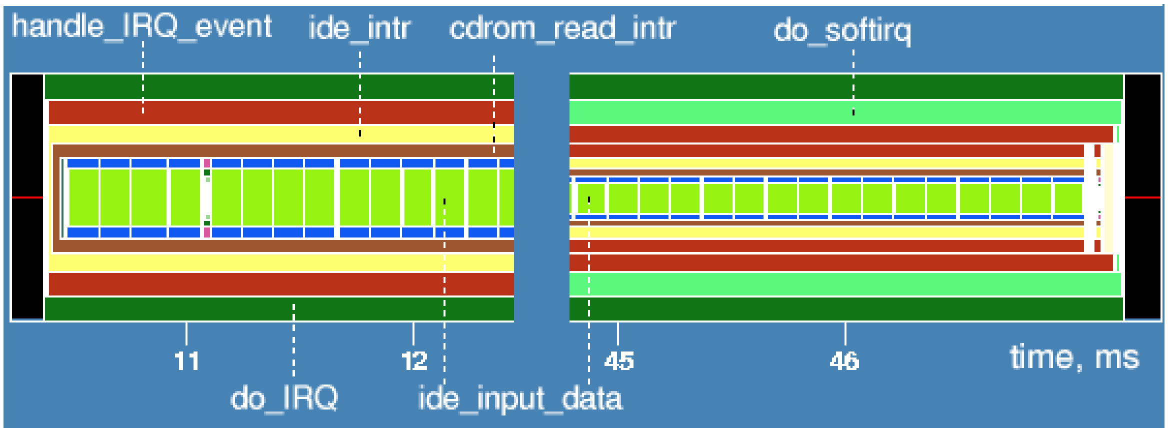


Figure 6.5: *Kernel activity under magnification*

To find the cause of preemption, we examine the kernel trace at a closer range. Figure 6.5 shows that the time is spent in *cdrom_read_intr*, called from *ide_intr*. This fact suggests that the system was handling an IDE interrupt, that signaled that the DVD drive had data available to be read. To handle the interrupt, *cdrom_read_intr* called *atapi_input_bytes* in a loop, which immediately called *ide_input_data*. By looking at the source code for *ide_input_data*, we found that it spent the time reading the data from the drive four bytes at a time with *insl* (input long from port) instructions.

The fact that fetching the data from the drive buffers is the bottleneck suggests a fix for the problem. By enabling DMA (Direct Memory Access), we allow the DVD drive to put the data in main memory without CPU intervention. The Linux IDE driver allows the superuser to change this setting at run time with the *hdparm* utility. As soon as we enabled DMA transfers for the DVD drive, *MPlayer* stopped dropping frames and the problem disappeared. New kernel traces reveal that DMA greatly offloads the CPU and allows the system to spend substantially less time handling IDE interrupts. The time to handle a single IDE interrupt (one invocation of *ide_intr*) has decreased from 9.0 milliseconds to 0.6 milliseconds.

A search on the Internet confirmed that problems with choppy DVD playback on Linux often are solved by enabling DMA. DMA is not used by default as certain old IDE chipsets are reported to cause data corruption in these conditions [96]. To summarize, self-propelled tracing was able to efficiently collect the data that we needed to identify the cause of a short and intermittent performance problem. Its ability to cross the kernel boundary proved to be a powerful mechanism that let us trace the problem into the kernel.

6.1.2 Analysis of a GUI Application

Wine [133] is an application that allows users to run Microsoft Windows applications in the X Windows environment. While it works well in most cases, *Wine* is not always able to match the native execution speed for some Windows applications. Often, GUI applications under *Wine* become noticeably less responsive. For example, *Wine* (Developers Release 20020411) sometimes takes several seconds to display a drop-down menu in Microsoft Word 97. This behavior is a minor nuisance as it happens only on the first access to the menu bar after the application starts. Nevertheless, it lets us demonstrate the general approach to pinpointing such sporadic problems with self-propelled instrumentation.

We initiate tracing when the user hits a particular key combination for the first time after launching *Wine* (we use “Alt-F” to open the drop-down File menu). We stop tracing when the user hits any key again. Although the key press activation method worked for other applications, we encountered a serious problem with *Wine* — our stack walking routine was unable to walk the stacks of *Winword*. The Microsoft

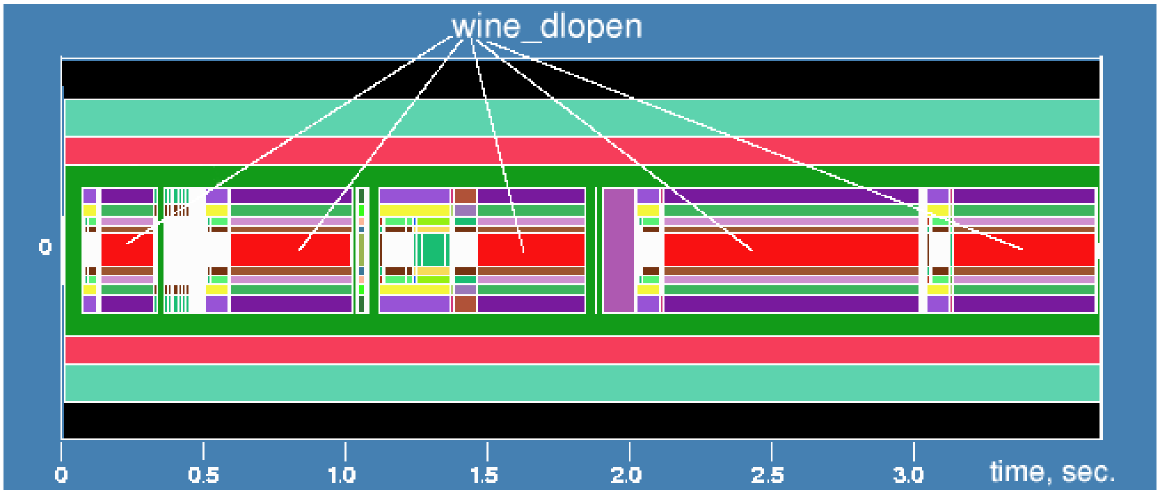


Figure 6.6: *Trace of functions on opening a drop-down menu (low-level details hidden)*

Windows binary of *Winword* is executed directly on Linux, and portions of its code and supporting libraries were optimized not to use the standard frame pointer, making stack walks virtually impossible.

In the future, the technique described in Section 3.1.1 will eliminate our reliance on stack walks. As an interim workaround, we use the following mechanism to trace *Wine*. We activate the self-propelled agent on application startup to follow the flow of control in *Wine*, but emit no trace records. When the "Alt-F" combination is pressed, all functions currently on the call stack are already instrumented. Therefore, the agent is able to start tracing without walking the stack.

A function-level trace of *Wine* obtained with this technique is displayed in Figure 6.6. The data show that *Wine* spent most time in five calls to *wine_dlopen*. After examining its source code, we found that this function loads dynamic libraries at run time. Apparently, on the first access to the menu bar, Microsoft Word needs to load five dynamic libraries from disk. This operation is expensive and it produces a user-visible freeze while displaying the menu. Further accesses to the menu bar will trigger no calls to *wine_dlopen*, as all required libraries will have been loaded by that time.

To identify the libraries that *Winword* tries to load, we modified the source code for *wine_dlopen* to print the name of a library being loaded. The majority of libraries loaded on a menu click were multimedia-related, including *winmm.dll*, a standard component of Windows Multimedia API, and *wineoss.drv.so*, the Wine OSS (Open Sound System) driver. Apparently, when the user opens a drop-down menu in *Winword*, the system tries to play an audible click. To play a sound, *Wine* needs to load multimedia libraries. After discovering the cause of the problem, we found that similar problems have been reported in the native Windows environment with another application (*Explorer.exe*) [79].

If we could pre-load these five libraries when *Wine* starts the application, the problem might go away. The users will take a relatively insignificant performance hit at startup, instead of taking the same hit at run time, when it is highly visible. To test this idea, we modified *Wine*'s source to perform the pre-loading and the problem was eliminated, reducing the time it takes to display the menu from several seconds to a fraction of a second.

Although *spTracer* was able to collect the data necessary to identify the cause of the performance problem, the study revealed a limitation of our current prototype. Our code analysis tool was unable to discover parts of code in the *Winword* executable and native Windows DLL libraries. As a result, our tracer did not follow some call chains, producing blind spots in the trace. A close examination of the trace revealed that the call chain leading to *wine_dlopen* passed through such a blind spot. Fortunately, functions after the blind spot on that chain were already instrumented at that time, because Wine executed them when loading several shared objects at startup. This fact allowed *spTracer* to receive control again, notice the calls to *wine_dlopen*, and identify the cause of the performance problem.

Discovering all code with offline analysis is not always possible [97]. Furthermore, offline tools cannot analyze dynamically-generated code. We plan to address these limitations with a hybrid approach that will combine our offline code analysis with on-the-fly capabilities similar to those of *Dynamo* [8], *DIOTA* [75], and *PIN* [73]. By maintaining a repository of code analysis results for known applications, we will retain the low-overhead feature of offline code analysis. Only when an unknown code fragment is discovered will our tracer parse it on the fly and add the results to the persistent repository. A similar approach has proved viable for dynamic binary translation in *FX!32*, where the repository was used for keeping translated code [28].

6.1.3 Debugging a Laptop Suspend Problem

Locating the cause of a problem in an unfamiliar piece of kernel code is a challenging task. Here, we describe how self-propelled instrumentation helped us investigate a power-management problem in an IBM Thinkpad T41 laptop computer running Linux. When left idle for a certain amount of time, the laptop would go into a power-saving standby mode. However, it could not wake up to resume normal execution after standby — the screen remained blank, it did not respond to keyboard or mouse events, and did not accept network connections. The freezes happened with the two versions of the Linux kernels that we tried (2.4.26 and 2.6.6). However, the original kernel that was pre-installed with the RedHat 9 distribution (2.4.20) did not exhibit this behavior.

To locate the cause of the problem, we used self-propelled instrumentation to trace kernel events that happen when the system transitions into the standby state. To initiate tracing, our tool sends a *KTRACE_START ioctl* request to */dev/ktracedrv* and then executes the *"apm -S"* command to initiate standby. The system starts generating the trace, runs for less than one second, and then locks up. Fetching the trace from memory of the frozen system may not be possible, so we resorted to printing trace records to a serial connection as they were being generated. (The T41 laptop does not have a serial connector, but its infrared port can emulate a serial port and connect to a similarly-configured machine on the other end).

Although the approach worked well for short traces, sending the complete record of all kernel function calls over a slow 115-Kbit serial connection introduced unacceptably high overhead. The system was unable to reach the standby state in 20 minutes, while without tracing it takes less than 1 second. To reduce the amount of trace data that needs to be sent over the serial connection, we restricted the scope of tracing to the *APM* power-management code in the Linux kernel. Instead of instrumenting the standard entries for hardware interrupts and system calls, we injected initial instrumentation at entry points of the *APM* device driver, */dev/apm_bios*, and did not trace functions that do not belong to the address range of the driver.

Figure 6.7 shows the trace of APM-related events immediately preceding the freeze. Apparently, the system locks up in the *apm_bios_call_simple* function as the call to *apm_bios_call_simple* never returns. By looking at the source code for *apm_bios_call_simple*, we confirmed that it invokes APM routines in the BIOS firmware. The chain of calls indicates that the driver was servicing the *STANDBY ioctl* request from

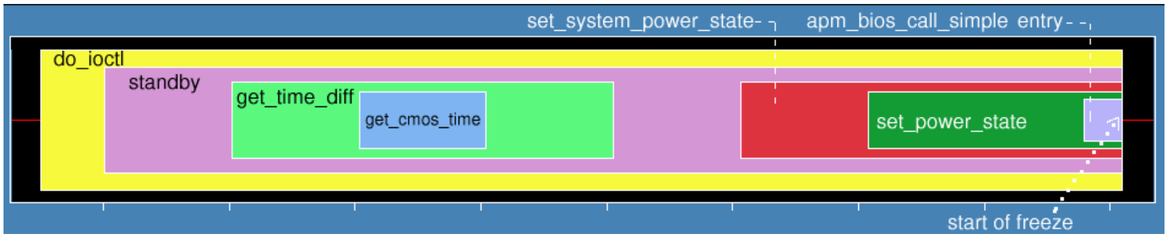


Figure 6.7: *A trace of APM events right before the freeze in Kernel 2.4.26*

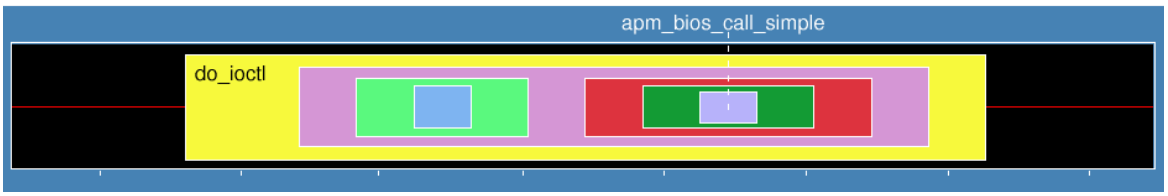


Figure 6.8: *A normal sequence of standby APM events in Kernel 2.4.20*

a user-level program. Compare it to Figure 6.8, that shows the record of APM events in the 2.4.20 kernel, which does not freeze. We see an almost-identical trace of calls, but *apm_bios_call_simple* returns normally.

While the system is executing APM BIOS calls, the interrupts are typically disabled. This fact may explain why the system does not respond to keyboard or mouse wake up events. To confirm that interrupts are not being serviced after the freeze, we adjusted our tracing approach to inject instrumentation into the *do_IRQ* hardware interrupt entry point after the flow of control reaches the *standby* function. The new traces showed no interrupt activity after the call to *apm_bios_call_simple*.

To verify whether the interrupts are not being serviced because they are disabled, we made the system generate periodic Non-Maskable Interrupts (NMI) by booting the kernel with the *nmi_watchdog=2* flag [100]. NMI interrupts are not affected by *cli* instructions and should be delivered even if other interrupts are disabled. To our surprise, even NMI interrupts are no longer generated when *apm_bios_call_simple* is entered. This behavior cannot be caused by disabling the maskable interrupts, and it made us suspect that the interrupt controller stopped functioning properly after the *STANDBY* call in *apm_bios_call_simple*.

Recent x86 systems are typically equipped with two different interrupt controllers. SMP machines require the use of APIC (Advanced Programmable Interrupt Controller), while uniprocessors can use either the new APIC or the original PIC (Programmable Interrupt Controller). APIC provides several new useful features, while hardware and software support for PIC appears to be better tested. To verify whether our problem is an APIC-stability issue, we recompiled the 2.4.26 kernel to use the standard PIC. With the new kernel, the system was able to enter the *STANDBY* mode and wake up on user activity properly.

The same problem might be found with traditional methods. One could step through execution of *"apm -S"* with an application debugger to identify the problem *ioctl* call and then step through *do_ioctl* in the *APM* driver with a kernel debugger. Further hypotheses could be tested by instrumenting the kernel source code and sending traces through the serial connection. However, the flexibility and dynamic nature of self-propelled instrumentation allowed us to find this problem without modifying the kernel.

6.2 Finding Bugs in a Collection of Identical Processes

Manual trace examination proved useful for finding the causes of problems in single-process applications and the kernel. However, this approach is difficult to use in large distributed systems where finding the failed process and the cause of a problem may require examination of many traces. In this section, we evaluate the effectiveness of our automated techniques for diagnosing distributed collections of identical processes using single-process flows. These techniques allowed us to locate the causes of two bugs in a distributed cluster management system called SCore [52].

6.2.1 Overview of SCore

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

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

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

In three months of monitoring a public installation of SCore v5.4 on a 129-node computational cluster at the Tokyo Institute of Technology, we witnessed several failures. To investigate the causes of such failures, we applied spTracer to collect function-level traces from *scored* processes on every node in the cluster. When each *scored* process starts, we inject our tracer agent into its address space and it starts collecting records of function calls and returns made by *scored*. To provide for continuous, always-on tracing of each node, we modify our approach to collect traces into a circular buffer: after reaching the end of the buffer, spTracer starts adding new trace records from the beginning. When *sc_watch* times out waiting for a patrol message, spTracer saves the accumulated trace buffers to disk.

Here, we describe two problems that we found with our automated data analyses. The first problem exhibited a fail-stop behavior where one of the cluster nodes terminated because of a time-out at the network link layer. The second problem exhibited a non-fail-stop behavior where one of the cluster nodes entered an infinite loop and stopped responding to the patrol messages. For both problems, we used techniques described in Section 5.2 to find the anomalous hosts. Then, we applied techniques described in Section 5.3 to find the causes of the anomalies.

6.2.2 Network Link Time-out Problem

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

Our earliest last timestamp approach described in Section 5.2.1 determined that the failure exhibited a clear fail-stop behavior. We identified that host `n014` stopped generating trace records more than 500 seconds earlier than any other host in the cluster. We examined the last trace entry on host `n014` and found that *scored* terminated voluntarily by calling the *score_panic* function and eventually issuing the *_exit* system call. Figure 6.9 visualizes the trace for host `n014`. The format of this visualization is the same as the one described in Section 6.1.1, but we used the newer *Jumpshot* tool [22] rather than *Nupshot* [49] due to its more efficient support for large traces.

We could not find the caller of *score_panic* due to the fixed-size trace buffer. The entire buffer preceding *score_panic* was filled with calls to *myri2kIsSendStable*, evicting the common caller of *myri2kIsSendStable* and *score_panic* from the buffer. Future versions of our tracer will address this limitation by maintaining a call stack for the most recent trace record and reconstructing the call stacks for earlier records.

For the problem at hand, we used the source code to find that *myri2kIsSendStable* and *score_panic* were called from *freeze_sending*. This finding suggests that *scored* waited until there were no more in-flight messages on the host's Myrinet-2000 NIC. When this condition did not occur after numerous checks, *scored* aborted by calling *_exit*. We reported the results of our analyses to the SCore developers and their feedback confirmed our findings. They had observed such symptoms in Ethernet-based networks, but our report showed them that a similar problem exists in Myrinet networks.

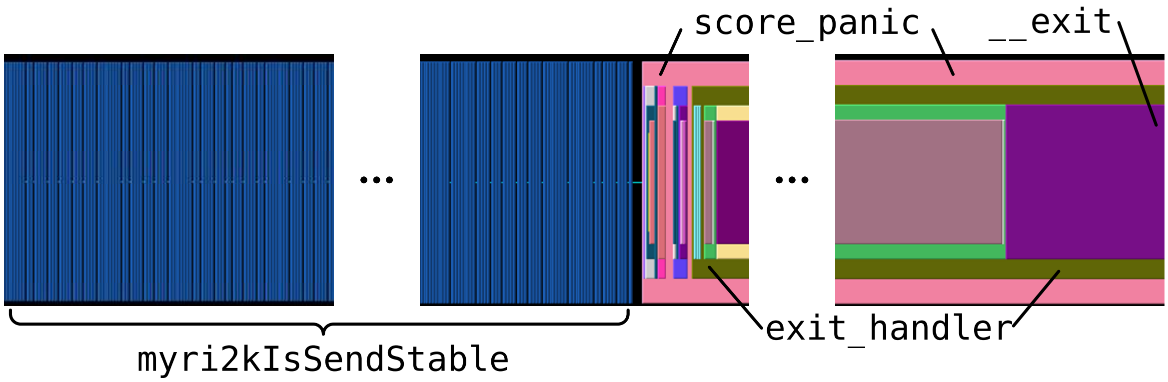


Figure 6.9: *Trace of scored on node* `n014` *visualized with Jumpshot*

6.2.3 Infinite Loop Problem

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

First, we decided that the problem did not exhibit a fail-stop behavior: the maximum difference between the last timestamps was only 20 seconds; the earliest host terminated less than a second earlier than the second earliest. Having identified that the problem was not fail-stop, we used our path-based ranking algorithms with results shown in Figure 6.10. For each point, the x-coordinate corresponds to the

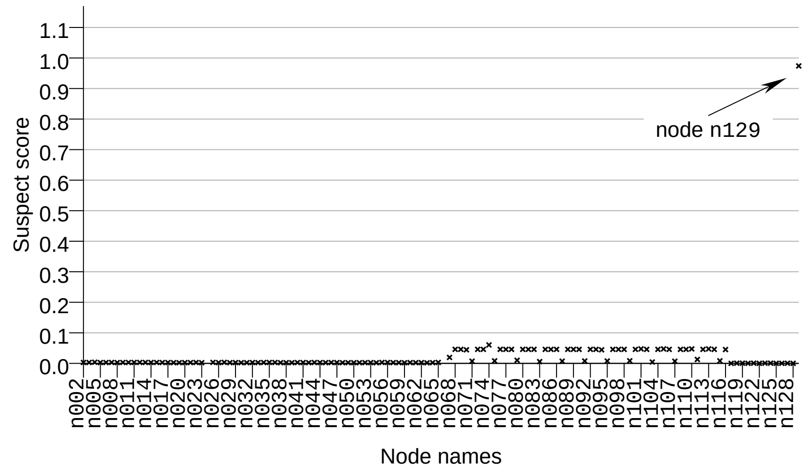


Figure 6.10: *Suspect scores in the scbcast problem*

name of a node, and the y-coordinate to its suspect score. As we see, the suspect score of node `n129` is substantially higher than those of other nodes, making it a likely candidate for detailed examination. To obtain Figure 6.10, we used traces from the failed run, and also added reference traces from previous runs. The reference traces for previous faulty runs with unrelated symptoms proved especially useful as they allowed us to eliminate a second node, whose anomalous behavior was a response to more than one type of problem, and therefore could not point to a specific problem cause.

In contrast, the behavior of `n129` has not been seen in other executions. Our algorithm explained the cause of such behavior by identifying that the path (*output_job_status* → *score_write_short* → *score_write* → *__libc_write*) had the largest contribution to the node's suspect score. Of 12 minutes of execution, *scored* spent 11 minutes executing in *__libc_write* on that path. Figure 6.11 visualizes the trace of node `n129` with the *Jumpshot* tool. This figure shows that *scored* entered *score_write* and started calling *__libc_write* in a loop, never returning from *score_write*. As a re-

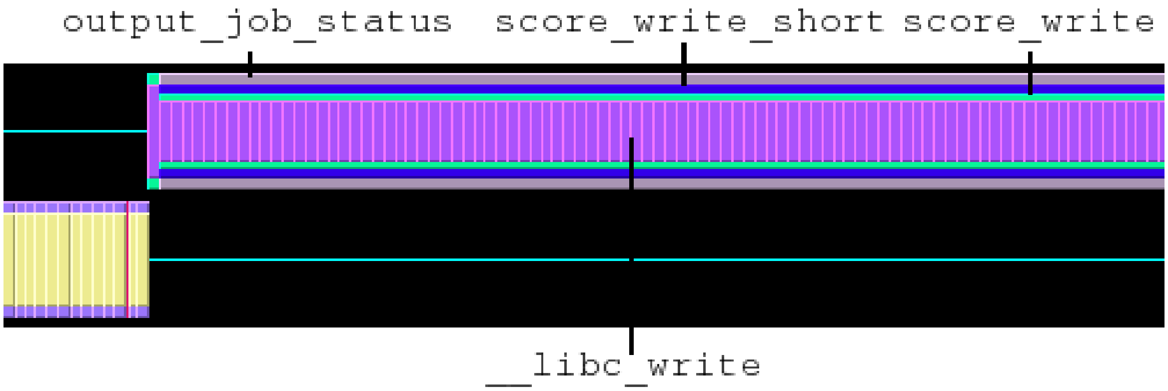


Figure 6.11: *Fragment of the scored trace from node* `n129`*. Two parallel timelines represent two user-level threads in scored.*

sult, *sc_watch* received no patrol message for more than 10 minutes, killed all *scored* processes, and tried to restart them. We see that `n129` was a true anomaly; our outlier-identification approaches were able to locate it, and the maximum-component approach found the correct symptom. Inspecting all traces manually would have required substantially more time and effort.

Examination of the source code revealed that *scored* was blocking while trying to write a log message to a socket. Finding the peer process would have been easy with our current prototype that is able to propagate across process and host boundaries. This study however, had been performed before cross-process propagation was implemented. Therefore, we did not trace the peer and had to examine the *scored* source code to determine that this socket was connected to the *scbcast* process. A write to a socket would block if *scbcast* froze and stopped reading from the socket [71]. If other *scored* nodes output a sufficient number of log messages, they would have also blocked. The fact that *scored* was not prepared to handle such failures pointed to a bug in its code. We reported the bug and it is being fixed for a future release of SCore.

6.3 Locating the File-Transfer Bug with Multi-process Flows

To evaluate our techniques in environments with complex multi-process flows, we used our prototype to investigate a file-transfer problem in the Condor distributed scheduling system (Version 6.7.17) [72, 127], where the output of a user job was placed in a wrong directory. Condor provides batch scheduling, execution, and monitoring facilities for high-throughput computing tasks. It can schedule sequential and parallel MPI and PVM jobs. Condor operates in a variety of environments from loosely-connected networks of desktop workstations to large-scale supercomputers and the Grid.

Unlike SCore where the main functionality is implemented in a single *scored* daemon, Condor is implemented as a collection of many different daemons. Each host that can submit user jobs runs a *schedd* daemon; each host that can execute user jobs runs a *startd* daemon. One of the hosts in a cluster is designated a *central manager*. The central manager runs two additional daemons: *collector* and *negotiator*. These daemons and several auxiliary tools are involved in accepting, scheduling, and controlling each user job. Being able to examine how these components interact is critical for finding the causes of problems in this environment.

Figure 6.12 shows key steps in executing a Condor job. In a typical scenario, the user creates a *job description file* and runs the *condor_submit* command with the created file to submit the job to the system. *Condor_submit* contacts the *schedd* on the local host, and *schedd* adds the job to its scheduling queue. Next, the *schedd* contacts the *negotiator* to find a matching execution host for the job. The *negotiator*

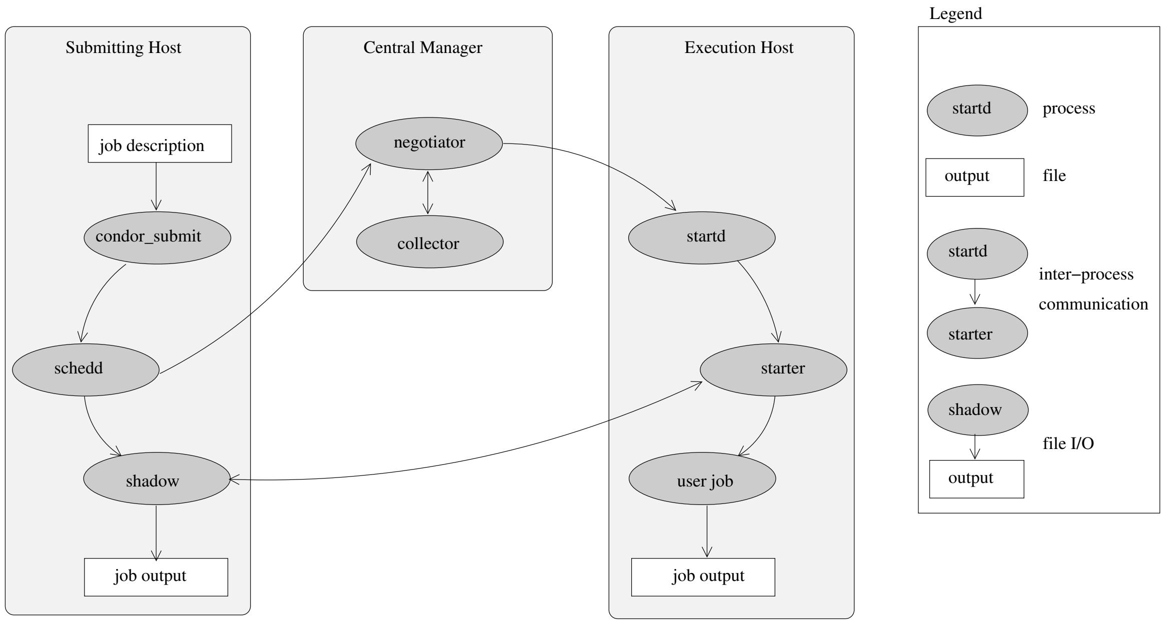


Figure 6.12: *Job submission and execution process*

uses the system information maintained by the *collector* to find an idle execution host.

Once the host is found, the *negotiator* contacts the *startd* on that host to establish a match between the submitting machine and the execution machine. Next, the *startd* spawns an auxiliary process called the *starter*, and the *schedd* on the submitting machine spawns an auxiliary process called the *shadow*. The *starter* and the *shadow* communicate to transfer the job input files to the execution machine, start and monitor the job, and transfer the output files to the submitting machine on job completion. Finally, Condor sends a notification email to the user.

6.3.1 Overview of the File Transfer Problem

The job description file can specify multiple jobs to be run as part of a single submission. In Condor terminology, these jobs are referred to as a *cluster*. Each job in a cluster is identified by a sequential process identifier. The output of each job typically is redirected to a separate output file whose name is specified in the job description file. To distinguish output files from different jobs, the user can specify a file name template that refers to environment variables, such as the sequential process identifier. The actual file name for each job is constructed at execution time by substituting the value of the job's process identifier in the template.

A recent bug in the Condor system caused the output files for some jobs in a cluster to be created in a wrong directory. The output for the first job is placed at the correct location, but the output for all subsequent jobs are created in the current working directory rather than the directory specified by the user. This bug had been located and fixed by Condor developers. Here, we describe how our automated analyses allowed us to find the same cause with little manual effort.

To locate the cause of this problem, we submitted a cluster of five jobs to Condor and collected the traces starting from *condor_submit*. Our tracer propagated through all the daemons and auxiliary programs involved in handling the jobs in the submitted cluster. These daemons and programs included: *schedd*, *shadow*, *collector*, *negotiator*, *startd*, *starter*, the *mail* client, and the user job. When the last job in the cluster terminated, we saved the traces to disk. Next, we attributed collected events to separate flows, where each flow corresponded to processing a single user job. Finally, we compared the call path coverage for the first (correct) flow with the coverage for the other (anomalous) flows. One of the differences in coverage corresponded to the actual cause of the problem. Below, we provide a more detailed discussion of how we applied our techniques to the Condor environment.

6.3.2 Collecting the Traces

We injected our agent into *condor_submit* using the *LD_PRELOAD* mechanism and instructed the agent to begin tracing when control flow reached the *NewProc* function. *NewProc* was called several times in *condor_submit* to reserve a sequential process identifier for each job and we used these events as flow starting points. The *condor_submit* command communicated with the *schedd* and terminated. Our agent however, propagated into *schedd* and the other daemons and continued tracing them until all the jobs completed. To detect the job completion events, we instructed Condor to use our custom *mail* program instead of the standard *mail* to notify the user. After our custom *mail* had been executed for each job in the cluster, we stopped tracing and began the analysis.

6.3.3 Separating the Flows

To attribute events to flows, we used the techniques described in Chapter 4. We started a flow at each *NewProc* invocation in *condor_submit* and attributed further events to this flow using our rules and two types of directives. Since Condor daemons use timers to defer execution of some actions until a later moment, we used temporary directives to match handler registration and invocation events. We also identified a location in the code where *schedd* can switch from working on one flow to another without a *recv* event. The job identifier was available at this location and we used it as the join attribute. We assigned the event of reaching this location to the same flow as an earlier job-registration event in *schedd* that returned the same job identifier.

To evaluate the accuracy of constructed flows, we submitted several identical jobs to Condor, collected the traces until all the jobs completed, and separated the traces

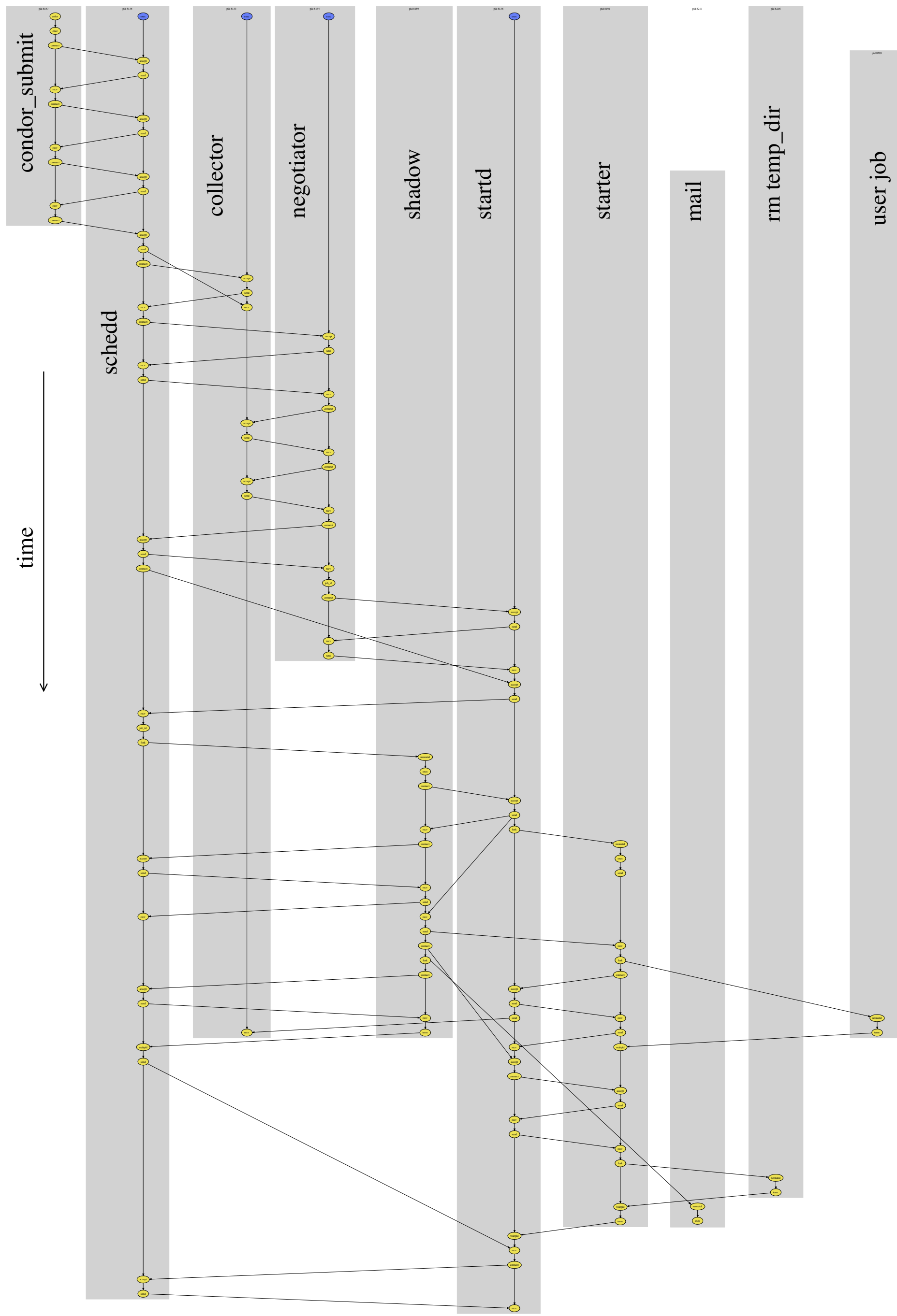
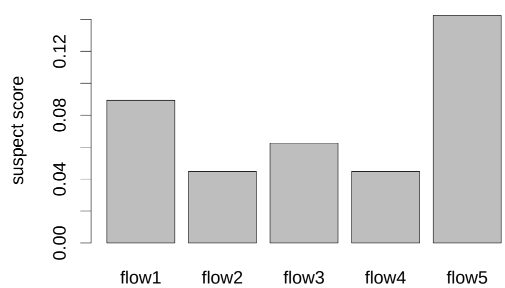


Figure 6.13: *A simplified flow graph for processing a Condor job. Gray rectangles represent timelines of processes. Dots in a rectangle represent communication events.*

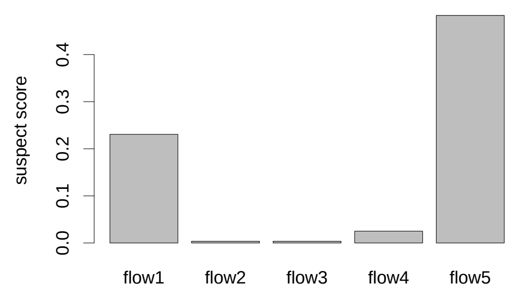
into flows. Then, we compared the flows to each other using two techniques: visual examination of flow graphs and automated comparison of flow profiles. An accurate algorithm should construct similar flows for similar jobs. In practice however, the Condor system performed different activities for different jobs. For example, the first job caused Condor to reserve (claim) an execution machine, the last job caused Condor to release the claim. To estimate the accuracy of our flow separation algorithm, we ignored the first and the last flow and only compared intermediate flows to each other.

Figure 6.13 shows a typical flow graph for a Condor job, as visualized with the *dot* tool from the Graphviz suite [42]. For brevity, we omitted repetitive communications over each channel. The complete graph for a flow is more complex, and we navigate it by zooming in on fragments of interest. Visual comparison of two graphs proved useful for identifying differences between flows. In particular, it enabled us to find the code location where *schedd* switched from working on one flow to another without a *recv* event. We provided a directive for this location, as described above. We have not seen other scenarios where our algorithm would attribute events to a wrong flow.

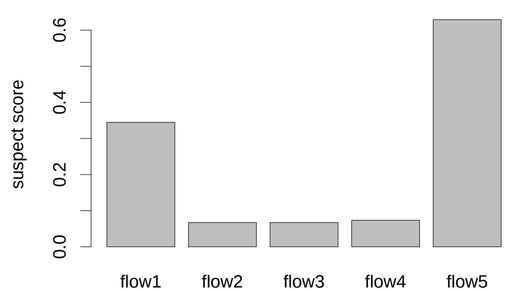
We also can evaluate the accuracy of our flow-construction algorithm using automated comparison of flow profiles. Our techniques for identifying anomalous flows compute the suspect score for each flow. A high suspect score indicates that the corresponding flow is unusual. Therefore, if the flow-construction algorithm produces accurate results, then similar flows corresponding to common behavior should receive low and similar suspect scores. Our algorithm satisfied this criterion and we present our initial experience with this technique at the end of the next section.



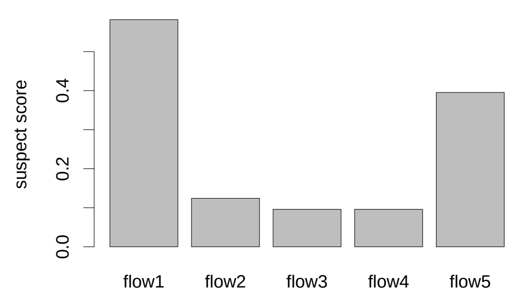
a: *Scores for time profiles*



b: *Scores for communication profiles*



c: *Scores for composite profiles*



d: *Scores for coverage profiles*

Figure 6.14: *Suspect scores for five flows in Condor using different types of profiles.*

6.3.4 Locating Anomalous Flows

Once the flows were constructed, we obtained the profiles for each flow, and analyzed them. In this study, profiles were already classified: the first job corresponded to the normal profile, subsequent ones corresponded to problem profiles. Therefore, we could directly apply our root cause identification techniques to the classified profiles. In this section however, we present our experience with automated anomaly detection techniques and compare their findings with the a priori classification. This comparison evaluated the accuracy of our flow construction and anomaly identification approaches.

To compute the suspect score for each flow, we used four types of profiles: time, communication, composite (time and communication), and coverage. Figure 6.14 shows the suspect scores for each flow and each type of profile. For all types of profiles, the scores for Flows 1 and 5 are higher than those for Flows 2–4. That is, Flows 2–4 correspond to common behavior, Flows 1 and 5 correspond to unusual behaviors. Furthermore, Flows 1 and 5 correspond to different types of unusual behavior.

Using a priori knowledge of the problem, we know that Flow 1 corresponds to the correct behavior (the first job created the output in the correct directory); Flows 2–5 correspond to the incorrect behavior. In Chapter 5, we presented our techniques for a scenario where the common behavior is correct and the unusual behavior is incorrect. However, these techniques also can be applied to the inverse scenario without any modification. By determining what distinguishes the unusual correct behavior (Flow 1) from the common incorrect behavior (Flows 2–4), we can find the cause of the problem. Section 6.3.5 presents our experience with this approach.

For time, communication, and composite profiles, the score for Flow 5 is higher than that for Flow 1. That is, Flow 5 is more different from the common behavior than Flow 1. However, since Flow 5 is known to exhibit the same problem as Flows 2–4, all differences between Flow 5 and the common behavior of Flows 2–4 represent normal rather than problem-induced variations in traces. Using manual examination of the most significant differences on Flow 5, we confirmed that they corresponded to normal tasks that needed to be done once for the cluster of jobs. Therefore, we did not use Flow 5 for problem diagnosis.

Overall, our anomaly detection techniques proved useful for identifying common and unusual flows. These techniques also allowed us to evaluate the accuracy of our flow-construction algorithm. Our key criterion is that an accurate algorithm should construct similar flows for similar activities. All intermediate jobs in the cluster did not reserve or clean-up per-cluster resources and should be similar to each other. Flows 2–4 have low suspect scores for all types of profiles. Therefore, these flows are similar to each other and our algorithm satisfies the accuracy criterion.

6.3.5 Root Cause Analysis: Time, Communication, Composite Profiles

We attempted to locate the root cause of the problem using each type of profile. For time, communication, and composite profiles, we used the technique presented in Section 5.3.2 to identify the call path with the highest contribution to the suspect score of Flow 1. Although it did not allow us to locate the actual root cause, it proved useful for understanding the key differences between observed flows. Here, we briefly outline the main findings for these types of profiles. In the next section, we describe how we located the actual cause of the problem using coverage profiles.

The suspect score of a flow is the distance to its k^{th} nearest neighbor, where $k =$

$\lfloor N/4 \rfloor$ and N is the number of flows. In this study, $N = 5$, thus $k = 1$, and the suspect score of a flow is the distance to its nearest neighbor. According to Equation 5.7, the call path with the maximum contribution to the score of Flow 1 corresponds to the maximum component of the delta vector between the profile of Flow 1 and its nearest neighbor. This call path corresponds to the most visible difference between flows. For communication, time, and composite profiles, our algorithm identified the following paths with the largest contribution to the suspect score:

1. For communication profiles, the nearest neighbor of Flow 1 was Flow 4. The largest difference between their communication profiles was a call path executed by *condor_submit*: *main* → *read_condor_file* → *queue* → *SetExecutable* → *SendSpoolFileBytes* → *ReliSock::put_file(size, name)* → *ReliSock::put_file(size, fd)* → *ReliSock::put_bytes_nobuffer* → *condor_write* → *__wrap_send* → *Generic_send*. By examining the source code for *condor_submit*, we determined that this call path transferred the job executable to the *schedd*. This activity needed to be done only for the first job in the cluster. Therefore, the most visible difference between Flow 1 and Flow 4 corresponded to a normal variation in traces; it did not allow us to locate the cause of the problem.

2. For time profiles, the nearest neighbor of Flow 1 was Flow 3. The largest difference between the time profiles for Flow 1 and Flow 3 was a call path *main* → *DaemonCore::Driver* → *__wrap_select* → *Generic_select* executed by the *starter*. Flow 1 spent 20% of the time on that path while Flow 3 spent only 13%. This difference indicates that *starter* waited longer in the *select* system call for Flow 1. This additional waiting corresponded to noise in the traces: after re-running the

test several times, we also detected the largest difference on other paths. As a result, we could not find the problem cause using time profiles.

3. For composite profiles, the nearest neighbor of Flow 1 was Flow 4. The largest difference corresponded to the call path in *condor_submit* transferring the job binary to the execution host, as previously identified for communication profiles. This difference did not allow us to locate the root cause of the problem.

6.3.6 Root Cause Analysis: Coverage Profiles

Since the most visible differences corresponded to normal variations in traces, we used coverage profiles to identify and inspect less visible differences. We compared the call path coverage profiles for the normal flow (Flow 1), and the anomalous flow with the most similar coverage (Flow 2). Each profile contained more than 80,000 distinct call paths. First, we represented each call path as a string of function addresses, sorted the paths lexicographically, and used the *diff* utility to identify strings present in one flow but not the other. This technique identified more than 21,000 differences between the flows. Of these differences, more than 17,500 call paths were present in the normal flow and absent from the anomalous one; in Section 5.3.4, we denoted the set of such paths Δ_n. More than 3,500 call paths were present in the anomalous flow and absent in the normal flow; we denoted the set of such paths Δ_a.

Next, we applied techniques described in Section 5.3.4 to eliminate differences that corresponded to effects of an earlier difference. The set transformation defined by Equation 5.8 reduced the number of differences to 292, more than a factor of 70. On average, a path in the remaining set of differences was the root of a subtree with

70 longer paths. The set transformation defined by Equation 5.9 further reduced the number of differences to 107, an additional factor of 2.7. On average, there were 2–3 paths that differed only in the last function called.

Finally, we arranged the remaining 107 paths in the order of their first occurrence in the flow and started examining the earliest ones. Most of these paths corresponded to normal variations in coverage: processing of the first job in the cluster requires additional initialization tasks. For example, the earliest difference occurred in the *schedd*: function *IncrementClusterSize* on path (*main* → *DaemonCore::Driver* → *DaemonCore::HandleReq* → *handle_q* → *do_Q_request* → *NewProc* → *IncrementClusterSize*) called *HashTable*⟨*int*, *int*⟩*::insert* in the normal flow but returned without making function calls in the anomalous flows. This difference corresponded to a normal variation in flows: it created an entry in a per-cluster hash table and needed to be performed only for the first job in the cluster.

However, a similar difference in *condor_submit* allowed us to find the cause of the file directory problem. The 15^{th} earliest difference corresponded to the path (*main* → *read_condor_file* → *queue* → *SetTransferFiles* → *InsertJobExpr* → *HashTable*⟨$MyString$, *int*⟩*::lookup*). This path was present in the anomalous flow but not in the normal one. By examining the source code for *SetTransferFiles* and *InsertJobExpr*, we found that the name of the output file was incorrectly registered in a hash table as a per-cluster attribute. Per-cluster attributes are shared among all jobs in a cluster while the output file name has to be unique for each job. Below, we describe why this bug caused the output to be placed in the wrong directory.

Condor transforms the output file names as illustrated in Figure 6.15. First, the output file name specified by the user (e.g., *outdir/job0.out*) is mapped to an internal

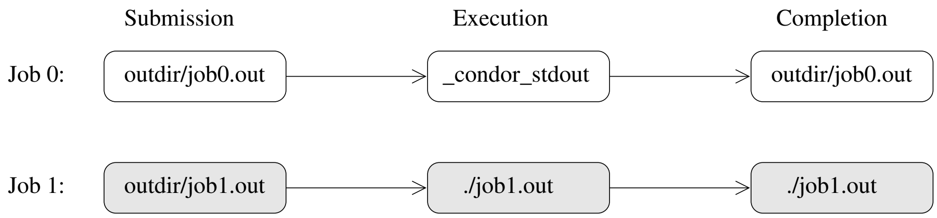


Figure 6.15: *Output file names at job submission, execution, and completion phases. White boxes represent the names for the first job. Gray boxes represent the names for the second job.*

file name (*_condor_stdout*) on the execution machine. Second, the user-specified name is recovered on job completion and the output is transferred to the submission machine. As a result of the discovered bug, Condor failed to replace the user-specified name with the internal name on submission. The execution machine stripped the directory name and placed the output in *job1.out*. On job completion, Condor failed to map this name to the original name. Instead, it treated *job1.out* as one of the files created by the job and moved it to the current working directory on the submission machine.

We fixed the problem by changing the call to *InsertJobExpr* to treat the output file name as a per-job attribute. As a result of this change, *InsertJobExpr* no longer called $HashTable\langle MyString, int\rangle$*::lookup* for the output file name attribute, the name in the job data structure was correctly replaced with *_condor_stdout*, and the output file name was mapped back to the user-specified location. In Condor version 6.7.18, the developers had introduced a similar fix.

6.3.7 Incorporating Call Site Addresses into Analysis

After fixing the problem, we reran the test and discovered that the previously discussed path (*main* $\rightarrow$ *read_condor_file* $\rightarrow$ *queue* $\rightarrow$ *SetTransferFiles* $\rightarrow$ *InsertJobExpr* $\rightarrow$ *HashTable*$\langle MyString, int\rangle$*::lookup*) was still part of the difference between the first and the second flow. Although this path was no longer taken when constructing the output file attribute, our tracer recorded the same path for several unrelated per-cluster attributes. Function *SetTransferFiles* invoked *InsertJobExpr* from several call sites, but our prototype did not distinguish these invocations as different paths. Unlike the path for the output file attribute however, these paths corresponded to normal variations between flows and must be ignored.

Such variations did not prevent us from finding the cause of this problem. However, finding the causes of other problems may require analysis of paths that are distinguished by the call sites information. Since our agent already uses call site instrumentation, augmenting our approach to record the site address for each function call is straightforward. Our analysis techniques would be able to handle call paths of the form ($main \rightarrow site_1 \rightarrow A \rightarrow site_2 \rightarrow B$) without any modification.

6.4 Locating the Job-run-twice Problem in Condor

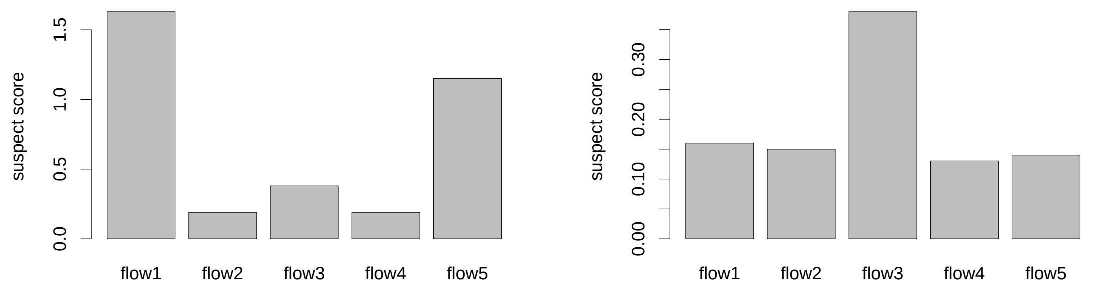
Techniques that allowed us to find the cause of the file-transfer problem also proved useful for finding another problem in the Condor environment. The problem was intermittent and was observed only at a customer site. The Condor *shadow* daemon crashed after reporting successful job completion. As a result, the *schedd* daemon restarted the *shadow* and the job was run for the second time. Re-running the job after reporting its successful completion caused a higher-level work-flow management component built on top of Condor to abort; reporting job completion twice also was confusing for the end user. Condor developers located and fixed the problem via numerous interactions with the customer and verbose logging of Condor events. Here, we describe how this problem could be found with the help of our automated techniques.

To reproduce the problem in our test environment, we inserted an intermittent fault in the Condor source code. If the fault occurs, it terminates the execution of the *shadow* daemon after it writes the job completion entry in the log; if the fault does not occur, the *shadow* completes successfully. Similar to the file-transfer problem, we then submitted a cluster of five identical jobs to Condor, obtained the system-wide trace that began at *condor_submit*, and separated the trace into flows. Next, we computed the suspect scores for each flow to identify the anomalous one. Finally, we applied our root cause analyses to help the analyst find the cause of the problem. Below, we present a summary of our findings.

6.4.1 Locating the Anomalous Flow

We used composite profiles to identify a flow that exhibits an unusual timing or communication behavior. Figure 6.16a shows the suspect scores for all flows computed without prior reference traces. Similar to Figure 6.14, Flows 1 and 5 have higher scores than the rest of the flows. Detailed examination of their differences from the common behavior showed that these differences corresponded to normal variations in activities performed only for the first and the last job in the cluster. Therefore, this approach is unable to locate the true anomaly. Time, communication, and coverage profiles performed similarly to the composite profiles and also could not locate the anomaly.

Unlike the file-transfer problem however, this problem was intermittent. As a



a: *Without reference traces (unsupervised)* **b:** *With known-normal traces (one-class)*

Figure 6.16: *Suspect scores for composite profiles of five Condor jobs computed using the unsupervised and the one-class methods*

result, we were able to obtain a set of known-correct traces, where the problem did not happen. Figure 6.16b shows the suspect scores for all flows computed using such known-correct traces as a reference. Flows 1 and 5 receive low suspect scores because similar flows were present in the normal run. In contrast, Flow 3 exhibits an anomalous behavior; it has not been observed in previous normal executions. By examining Condor job-completion log records, we confirmed that our automated approach identified the correct anomaly: Flow 3 corresponded to the job that was run twice.

6.4.2 Finding the cause of the anomaly

Similar to the file-transfer problem, we analyzed differences in call path coverage between Flow 3 in the anomalous run and Flow 3 in the known-normal previous run. The total number of paths present only in the anomalous flow or only in the normal flow was 964. After applying our techniques for eliminating call paths that are effects of earlier differences in the traces, we reduced the number of paths to inspect to 37, a factor of 26. The reduction factor was lower than that in the file-transfer problem since most differences corresponded to smaller functions with smaller call path subtrees. However, our elimination strategies still significantly reduced the number of paths and enabled us to examine the remaining ones manually.

We ranked all the paths by their time of first occurrence and started examining them from the earliest one. Similar to the file-transfer problem, several early paths did not represent the problem cause. They corresponded to noise in the trace due to minor variations in the workload. However, the 14^{th} path represented a symptom that was an immediate effect of the fault. In the anomalous flow, the *schedd* invoked function *DaemonCore::GetExceptionString* on path *main* → *DaemonCore::Driver* → *DaemonCore::HandleDC_SERVICEWAITPIDS* → *DaemonCore::HandleProcessExit* → *Scheduler::child_exit*. This function was not executed in the normal flow.

By examining the source code for *Scheduler::child_exit*, we determined that in normal executions, *Scheduler::child_exit* detects that the *shadow* terminated normally and marks the job as completed. However, when the *shadow* crashes or aborts, *Scheduler::child_exit* detects that the *shadow* terminated on receiving a signal, calls *DaemonCore::GetExceptionString*, writes a warning log record to the *schedd* log file, and reschedules the job. Due to the large size of the log file, and the large number of log files from other components on all hosts in the system, identifying this warning log record via manual examination of the logs was difficult. In contrast, our approach presented a small number of reports to the analyst and significantly simplified location of the failed component.

To evaluate the effectiveness of our second ranking strategy, we also ordered reported call paths by their length. This strategy placed the call path to *DaemonCore::GetExceptionString* in the first position: this path was the shortest among the differences. Here, ranking by length was more effective than ranking by time of occurrence. In contrast, in the file-transfer problem, the true cause was ranked 31^{st} by length and 15^{th} by time. For both problems, the hybrid ranking strategy that orders paths by length and then arranges the paths of the same length earliest-first performed the same as ranking by length. Determining which technique works better in most environments and designing alternative ranking techniques remains future work.

To find the location of the fault in the *shadow*, we examined the last functions called by the *shadow* in the anomalous Flow 3. The call path at the crash time pointed to the location where we previously inserted the intermittent fault. Therefore, our approach enabled us to correctly identify the cause of a non-trivial problem in Condor.

6.5 Summary of Experiments with Multi-process Flows

Our experience with locating the causes of two problems in Condor allows us to make four conclusions. First, our techniques for propagating across the process and host boundaries proved adequate for monitoring a complex distributed system. Condor contains multiple components that communicate via TCP sockets, UDP sockets, and UNIX pipes. Our prototype was able to propagate across such communications and obtain an accurate PDG for the end-to-end job scheduling process.

Second, our algorithm for transforming the PDG into disjoint flows proved useful for identifying activities that belong to separate jobs. The algorithm required little knowledge of system internals. In situations where such knowledge was necessary, user directives were a simple and effective mechanism for augmenting the automated algorithm. As a result, during manual investigation of call paths in Section 6.3.6, we have not seen a call path being attributed to a wrong flow. Automated comparison of profiles also demonstrated that similar activities correspond to similar flows.

Third, techniques for automated anomaly detection proved useful for identifying common and unusual flows. Time, communication, and composite profiles were effective for identifying the most visible differences between flows. The differences identified by the unsupervised algorithm represented normal variations in flows. Some actions had to be performed only for the first or the last job in the cluster and did not correspond to anomalies. In contrast, the one-class algorithm was able to identify the anomalous flow and ignore normal unusual flows.

Finally, coverage profiles proved useful for diagnosing both problems. The cause of each problem was manifested as a call path present in the anomalous flows and absent from the normal one. In addition to the cause of the problem however, the differences in coverage between normal and anomalous flows contained many normal variations. Our automated techniques were able to reduce the number of paths to inspect by a factor of 26–200. Our ranking techniques simplified manual path examination by ordering these paths by their length or time of occurrence.

Chapter 7

Conclusion

In this work, we identified key challenges that complicate problem diagnosis in production distributed systems. We have presented an approach that addresses these challenges, simplifies manual problem diagnosis, and locates many problems automatically. Below, we summarize our contributions and present directions for future work.

7.1 Contributions

We have identified two requirements to a diagnostic tool for production systems:

Work on a live system in a field environment. Since problems in production systems are often intermittent or environment-specific, reproducing them in the controlled development environment may be difficult. The diagnostic tool must work on the actual system in the field. Furthermore, since pausing a production system may not be possible, the tool must monitor system execution autonomously, with little interactive help from the user.

Automate the diagnosis to the extent possible. To diagnose a variety of problems, the tool must collect detailed run-time information. The volume of such information is often prohibitively large for manual examination. Thus, the tool must analyze the data automatically or significantly simplify manual examination.

To enable creation of tools that satisfy these requirements, we developed a three-step approach. Techniques used at each step form the main contribution of our work:

- **Self-propelled instrumentation technology.** This technology provides for autonomous, detailed, and low-overhead monitoring or control of system execution. The corner stone of this approach is a small fragment of code called an *agent* that is dynamically injected into a running process. The agent propagates through the system, inserting instrumentation ahead of the control flow in the traced process, and also across process and kernel boundaries. Within a process, self-propelled instrumentation is a low-overhead execution monitoring mechanism that can be rapidly enabled on-demand. In Section 3.2.4, we showed that it has substantially lower start-up overhead than dynamic binary translation.

  The unique feature of self-propelled instrumentation is the ability to propagate across process and kernel boundaries. The agent detects that the traced process attempts to communicate with another process, injects itself into the remote process, and starts following both flows of control. This mechanism is a foundation for system-wide on-demand problem diagnosis. On-demand deployment is crucial in production distributed systems as it allows us to introduce no overhead during normal execution, where tracing is not required.

- **Flow-separation algorithm.** Our current prototype agent obtains the distributed control flow trace of the execution. In modern systems, such traces often interleave events from multiple concurrent activities, possibly from different users. For manual examination, the presence of unlabeled events from interleaves activities is confusing; for automated analysis, such events increase trace variability and also complicate problem diagnosis.

To address these problems, we designed an algorithm that separates events from different activities. This algorithm constructs sets of events that we call flows. Each flow corresponds to a separate activity and can be constructed with little application-specific knowledge. In scenarios where our application-independent techniques are unable to construct accurate flows, we provided simple techniques for introducing program knowledge into the analysis.

- **Automated root cause identification algorithm.** We developed a collection of techniques based on a common framework for identifying the cause of some problems automatically. We focus on identification of anomalies rather than massive failures, as anomalies are often harder to investigate manually. By looking for anomalies in the distributed control flow, we locate an anomalous activity (e.g., an unusual request in an e-commerce system or a node of a parallel application), and help the analyst to identify the root cause of the anomaly (e.g., a function that is a likely location of the problem). Our techniques can work without prior examples of correct or failed executions, but can provide a more accurate diagnosis if examples of correct or failed executions are available.

To evaluate the effectiveness of our techniques, we applied our prototype tool to identification of several real-world problems in sequential and distributed applications. Self-propelled instrumentation proved effective for low-overhead on-demand data collection across the process and kernel boundaries. Our flow-separation algorithm constructed accurate flows in the Condor batch scheduling system. Manual trace examination enabled us to locate the causes of two performance problems in a multimedia and a GUI application, and a crash in the Linux kernel. Our automated analyses enabled us to find the causes of three problems in the SCore and Condor batch scheduling systems.

7.2 Future Work

We envision four major directions for future development of presented ideas:

- Self-propelled instrumentation can be extended to operate in more environments. We can support more message-passing communication mechanisms using steps outlined in Section 3.3. For each mechanism, we need to detect the initiation of communication, identify the name of the destination component, and detect the receipt of communication at the destination.

  Supporting communications in shared-memory systems is more difficult. First, defining the semantics of flows in such systems is an open research question. Similar to matching message-passing events, in shared-memory systems, we can introduce *data-dependent* and *synchronization-dependent* events [29, 90]. Unlike the communication-pair rule in the message-passing case however, neither data-dependent nor synchronization-dependent events must belong to the same flow.

  In a common scenario, threads servicing different requests or transactions may load, modify, and store a shared variable in a critical section. The load in thread u is data-dependent on the store in a thread t that previously accessed the critical section. However, the threads serve different requests so the load and the store must belong to different flows. Similarly, the event of thread u entering the critical section is synchronization-dependent on the event of thread t leaving the critical section. Despite the synchronization dependence, these events also belong to different requests.

Second, determining when a load event in thread u is data-dependent on a store event in thread t is difficult. We would need to detect the store by t, the load by u, and determine that the load happened after the store. Tracing loads and stores may introduce prohibitive overhead in production environments. Moreover, determining that the load happened after the store may not be possible if the memory accesses are not synchronized. Such accesses are common in environments that contain race conditions.

- For adoption of the tool in real-world multi-user distributed environments, we need to develop a security policy that determines whether host A is allowed to retrieve and examine traces collected on host B. To assemble per-host traces at a central location after the user stops the tracing, we currently access remote hosts and transfer their trace files over SSH [13]. In a real-world system however, the owner of the server may not let the clients examine the trace files or vice versa. System-wide control-flow traces might expose information about activities of other users on the server. To operate under such constraints, we envision an independent network service that is trusted by all parties to examine the traces and report the analysis results back.

- Our automated analyses may benefit from more detailed data collection mechanisms. More detailed data may enable us to locate a more accurate cause of a problem or locate problems with no differences in control-flow traces. For example, by recording the values of variables read and written, we could collect data-flow traces and extend our root cause analysis techniques to support them. Although the overhead of general data-flow tracing may be prohibitive, tracing only some variables may prove useful. For example, by recording function arguments and return values, we can obtain annotated control-flow traces and possibly improve the accuracy of our analyses.

- Self-propelled instrumentation can be applied to other domains. In addition to monitoring the execution of a system, the agent can also control it. Program shepherding uses dynamic binary translation to sandbox applications and prevent them from executing malicious code [55]. Self-propelled instrumentation might be used to extend this functionality to the entire distributed system.

References

[1] M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance Debugging for Distributed Systems of Black Boxes", *19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, New York, October 2003.

[2] A.V. Aho, R. Sethi, J.D. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986, ISBN 0-201-10088-6.

[3] M. Arnold and B.G. Ryder, "A framework for reducing the cost of instrumented code", *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Salt Lake City, Utah, June 2001.

[4] L. Albertsson, "Temporal Debugging and Profiling of Multimedia Applications", *Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2002.

[5] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl, "Continuous Profiling: Where Have All the Cycles Gone", *ACM Transactions on Computer Systems* **15**, 4, November 1997, pp. 357–390.

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

[7] A. Ayers, R. Schooler, A. Agarwal, C. Metcalf, J. Rhee, and E. Witchel, "TraceBack: First-Fault Diagnosis by Reconstruction of Distributed Control Flow", *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[8] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System", *Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, June 2000.

[9] T. Ball and J.R. Larus, "Optimally profiling and tracing programs", *ACM Symposium on Principles of Programming Languages (POPL)*, Albuquerque, NM, January 1992.

[10] R.M. Balzer, "EXDAMS — EXtendable Debugging and Monitoring System", *AFIPS Spring Joint Computer Conference* **34**, 1969, pp. 567–580.

[11] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for Request Extraction and Workload Modelling", *6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, San Francisco, CA, December 2004.

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

[13] D.J. Barrett, R.G. Byrnes, and R.E. Silverman, "SSH, the Secure Shell", O'Reilly, 2nd edition, 2005, ISBN 0-596-00895-3.

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

[15] A.R. Bernat and B.P. Miller, "Incremental Call-Path Profiling", *Concurrency: Practice and Experience*, to appear, 2006.

[16] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, "An Architecture for Differentiated Services", Request for Comments (Informational) RFC 2475, December 1998.

[17] D. Bruening, E. Duesterwald, and S. Amarasinghe, "Design and Implementation of a Dynamic Optimization Framework for Windows", *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, Austin, TX, December 2001.

[18] B. Buck and J.K. Hollingsworth, "An API for runtime code patching", *Journal of High Performance Computing Applications*, **14**, 4, pp. 317–329, Winter 2000.

[19] B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal, "Dynamic Instrumentation of Production Systems", *USENIX Annual Technical Conference*, Boston, June 2004

[20] R.C. Carrasco and J. Oncina, “Learning stochastic regular grammars by means of a state merging method”, *2nd International Colloquium on Grammatical Interference and Applications (ICGI)*, Alicante, Spain, September 1994.

[21] J. Cespedes, LTrace home page, `http://www.cespedes.org/software/ltrace/`

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

[23] A. Chan, W. Gropp, and E. Lusk, “User’s Guide for MPE: Extensions for MPI Programs”, Mathematics and Computer Science Division, Argonne National Laboratory, `http://www-unix.mcs.anl.gov/mpi/mpich1/docs/mpeman/mpeman.htm`

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

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

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

[27] T.Y. Chen and Y.Y. Cheung, “Dynamic Program Dicing”, *International Conference on Software Maintenance (ICSM)*, Montreal, Canada, September 1993.

[28] A. Chernoff and R. Hookway, “DIGITAL FX!32 Running 32-Bit x86 Applications on Alpha NT”, *USENIX Windows NT Workshop*, Seattle, Washington, August 1997.

[29] J.D. Choi, B.P. Miller, R.H.B. Netzer, “Techniques for Debugging Parallel Programs with Flowback Analysis”, *ACM Transactions on Programming Languages and Systems* **13**, 4, 1991, pp. 491–530.

[30] J.D. Choi and H. Srinivasan, “Deterministic Replay of Java Multithreaded Applications”, *SIGMETRICS Symposium on Parallel and Distributed Tools*, Welches, OR, August 1998.

[31] J.D. Choi and A. Zeller, ”Isolating Failure-Inducing Thread Schedules”, *International Symposium on Software Testing and Analysis*, Rome, Italy, July 2002.

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

[33] W.E. Cohen, “Multiple Architecture Characterization of the Linux Build Process with OProfile”, `http://people.redhat.com/wcohen/wwc2003/`.

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

[35] CyberLink Corporation, "CyberLink PowerDVD 5 System Requirements", `http://www.gocyberlink.com/english/products/powerdvd/system_requirements.jsp`

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

[37] E. Duesterwald, R. Gupta, and M.L. Soffa, "Distributed Slicing and Partial Re-execution for Distributed Programs", *5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, Connecticut, August 1992.

[38] Y. Endo and M. Seltzer, "Improving Interactive Systems Using TIPME", *ACM SIGMETRICS Performance Evaluation Review* **28**, 1, June 2000, pp. 240–251.

[39] Etnus TotalView, `http://www.etnus.com`

[40] H.H. Feng, O.M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information", *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

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

[42] E.R. Gansner and S.C. North, "An open graph visualization system and its applications to software engineering", *Software — Practice and Experience* **30**, 11, September 2000, pp. 1203–1233.

[43] S. Graham, P. Kessler, and M. McKusick, "gprof: A Call Graph Execution Profiler", *SIGPLAN'82 Symposium on Compiler Construction*, Boston, June 1982, pp. 120–126.

[44] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops", *20th IEEE/ACM international Conference on Automated Software Engineering (ASE)*, Long Beach, CA, November 2005.

[45] R.J. Hall, "Call Path Refinement Profiles", *IEEE Transactions on Software Engineering* **21**, 6, June 1995, pp. 481–496.

[46] L. Harris and B.P. Miller, "Practical Analysis of Stripped Binary Code", *Workshop on Binary Instrumentation and Applications (WBIA-05)*, St. Louis, Missouri, September 2005.

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

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

[49] V. Herrarte, E. Lusk, "Studying parallel program behaviour with upshot", *Technical Report ANL91/15*, Argonne National Lab, 1991.

[50] J.K. Hollingsworth, M. Altinel, “Dyner User’s Guide”, http://www.dyninst.org/docs/dynerGuide.v40.pdf

[51] Intel Corp., “Vampirtrace User’s and Installation-Guide”, http://www.pallas.com/e/products/pdf/VT20userguide.pdf

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

[53] J. Houston, “Kernel Trace Mechanism for KDB”, http://www.ussg.iu.edu/hypermail/linux/kernel/0201.3/0888.html

[54] “KerninstAPI Programmer’s Guide”, http://www.paradyn.org/kerninst/release-2.0/kapiProgGuide.html

[55] V. Kiriansky, D. Bruening, and S. Amarasinghe, “Secure execution via program shepherding”, *11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[56] B. Korel and R. Ferguson, “Dynamic Slicing of Distributed Programs”, *Applied Mathematics and Computer Science* **2**, 2, 1992, pp. 199–215.

[57] D. Jackson and E.J. Rollins, “Chopping: A generalisation of slicing”, Technical Report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994.

[58] Kernprof, http://oss.sgi.com/projects/kernprof/

[59] D. Knuth, "The Art of Computer Programming, Volume 3: Sorting and Searching", 3rd Edition, Addison-Wesley, 1997. ISBN 0-201-89685-0.

[60] S. Krempel, "Tracing Connections Between MPI Calls and Resulting PVFS2 Disk Operations", Bachelor's Thesis, March 2006, Ruprecht-Karls-Universität Heidelberg, Germany.

[61] L. Lamport, "Time, clocks and the ordering of events in a distributed system", *Communications of the ACM* **21**, 7, 1978, pp. 558–565.

[62] T. Lane and C.E. Brodie, "Temporal sequence learning and data reduction for anomaly detection", *5th ACM Conference on Computer and Communications Security*, San Francisco, CA, November 1998.

[63] J.R. Larus and E. Schnarr, "EEL: Machine-independent executable editing", *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, CA, June 1995.

[64] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging parallel programs with instant replay", *IEEE Transactions on Computers* **C-36**, 4, April 1987, pp. 471–482.

[65] A. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals", *Soviet Physics-Doklady* **10**, 8, 1966, pp. 707–710.

[66] J. Li, "Monitoring and Characterization of Component-Based Systems with Global Causality Capture", Technical Report HPL-2003-54, Imaging Systems Laboratory, HP Laboratories.

[67] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan, "Bug Isolation via Remote Program Sampling", *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2003.

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

[69] K.A. Lindlan, J. Cuny, A.D. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen, "Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates", *High Performance Networking and Computing Conference (SC 2000)*, Dallas, TX, November 2000.

[70] M. Linton, "The Evolution of Dbx", *USENIX Summer 1990*, Anaheim, CA, June 1990.

[71] Linux Manual Pages

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

[73] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[74] J.R. Lyle and M. Weiser, "Automatic Program Bug Location by Program Slicing", *2nd International Conference on Computers and Applications*, Beijing, China, June 1987.

[75] J. Maebe, M. Ronsse, K. De Bosschere, "DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications", *WBT-2002: Workshop on Binary Translation*, Charlottesville, Virginia, September 2002.

[76] P.S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrom, and B. Werner, "SimICS/sun4m: A Virtual Workstation", *USENIX Annual Technical Conference*, New Orleans, June 1998.

[77] M. McKusick, "Using gprof to Tune the 4.2BSD Kernel", *European UNIX Users Group Meeting*, April 1984.

[78] Microsoft Corp., "Event Tracing for Windows (ETW)", `http://msdn.microsoft.com/library/en-us/perfmon/base/event_tracing.asp`, 2002.

[79] Microsoft Knowledge Base, "Performance Issues When Loading Winmm.dll", `http://support.microsoft.com/default.aspx?scid=kb;en-us;266327`

[80] B.P. Miller, M. Clark, J.K. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Computers* **1**, 2, April 1990, pp. 206–217.

[81] B.P. Miller, "DPM: A Measurement System for Distributed Programs", *IEEE Transactions on Computers* **37**, 2, February 1988, pp. 243–247.

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

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

[84] A.V. Mirgorodskiy and B.P. Miller, "CrossWalk: A Tool for Performance Profiling Across the User-Kernel Boundary", *International Conference on Parallel Computing*, Dresden, Germany, September 2003.

[85] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller, "Problem Diagnosis in Large-Scale Computing Environments", *SC'06*, Tampa, FL, November 2006.

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

[87] B. Mohr, D. Brown, and A. Malony, "TAU: A portable parallel program analysis environment for pC++", *International Conference on Parallel and Vector Processing (CONPAR 94–VAPP VI)*, Linz, Austria, September 1994.

[88] *MPlayer home page*, `http://www.mplayerhq.hu/`

[89] N. Nethercote and J. Seward, "Valgrind: A program supervision framework", *3rd Workshop on Runtime Verification (RV03)*, Boulder, CO, July 2003.

[90] R.H.B. Netzer and B.P. Miller, “On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions”, *International Conference on Parallel Processing*, St. Charles, IL, August 1990.

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

[92] R.H.B. Netzer and M.H. Weaver, “Optimal tracing and incremental reexecution for debugging long-running programs”, *ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, June 1994.

[93] K. Nichols, S. Blake, F. Baker, D. Black, “Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers”, Request for Comments (Standards Track) RFC 2474, December 1998.

[94] Parallel Virtual File System, Version 2, `http://www.pvfs.org/pvfs2/index.html`

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

[96] “Playing DVD and videos”, `http://www.linuxquestions.org/questions/answers/28`

[97] M. Prasad and T. Chiueh, “A Binary Rewriting Defense against Stack-based Buffer Overflow Attacks”, *USENIX 2003 Annual Technical Conference*, San Antonio, TX, June 2003.

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

[99] P. Raymond and Y. Roux, "Describing non-deterministic reactive systems by means of regular expressions", *First Workshop on Synchronous Languages, Applications and Programming (SLAP)*, Grenoble, April 2002.

[100] "Red Hat Cluster Suite Configuring and Managing a Cluster", `http://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/cluster-suite/ap-hwinfo.html`

[101] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment", *Scalable Parallel Libraries Conference*, Los Alamitos, CA, October 1993, pp. 104–113.

[102] T. Reps and G. Rosay, "Precise interprocedural chopping", *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, DC, October 1995.

[103] Research Triangle Institute, "The Economic Impacts of Inadequate Infrastructure for Software Testing", *National Institute of Standards and Technology Planning Report 02-3*, May 2002.

[104] W. Robertson, itrace home page, `http://www.cs.ucsb.edu/~wkr/projects/itrace/`

[105] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, B. Bershad, H. Levy, and B. Chen, "Etch, an Instrumentation and Optimization tool for Win32 Programs", *USENIX Windows NT Workshop*, Seattle, WA, August 1997.

[106] M. Ronsse and K. De Bosschere, "RecPlay: A fully integrated practical record/replay system", *ACM Transactions on Computer Systems* **17**, 2, May 1999, pp. 133–152.

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

[108] M. Rosenblum, E. Bugnion, S. Devine, and S.A. Herrod, "Using the SimOS machine simulator to study complex computer systems", *ACM Transactions on Modeling and Computer Simulation* **7**, 1, January 1997, pp. 78–103.

[109] P.C. Roth and B.P. Miller, "On-line Automated Performance Diagnosis on Thousands of Processes", *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, New York City, March 2006.

[110] M. Russinovich, TCPView home page, `http://www.sysinternals.com/Utilities/TcpView.html`

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

[112] B. Schendel, "TraceTool: A Simple Dyninst Tracing Tool", `http://www.paradyn.org/tracetool.html`

[113] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati, "A fast automaton-based approach for detecting anomalous program behaviors", *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[114] S. Sistare, D. Allen, R. Bowker, K. Jourenais, J. Simons, and R. Title, "A Scalable Debugger for Massively Parallel Message-Passing Programs", *IEEE Parallel and Distributed Technology* **1**, 2, Summer 1994.

[115] M. Snyder and J. Blandy, "The Heisenberg Debugging Technology", *Embedded Systems Conference West*, September 1999.

[116] S.M. Srinivasan, S. Kandula, C.R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging", *USENIX Annual Technical Conference*, Boston, MA, June 2004.

[117] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools", *Conference on Programming Language Design and Implementation (PLDI)*, Orlando, Florida, June 1994.

[118] R.M. Stallman, R.H. Pesch, S. Shebs, et al., "Debugging with GDB, The GNU Source-Level Debugger", Free Software Foundation, January 2002.

[119] W.R. Stevens, "TCP/IP Illustrated, Volume 1: The Protocols", Addison Wesley Professional, 1994, ISBN 0-201-63346-9.

[120] W.R. Stevens, "UNIX Network Programming", Prentice Hall, vol. 1, 2nd edition, 1998, ISBN 0-13-490012-X.

[121] Sun Microsystems, "prof — display profile data", Solaris 2.6 Reference Manual Answer Book, man pages(1), `http://docs.sun.com`

[122] Sun Microsystems, "Tracing Program Execution With the TNF Utilities", Solaris 7 Software Developer Collection, `http://docs.sun.com`

[123] Sun Microsystems, "Truss — trace system calls and signals", Solaris 9 Reference Manual Collection, `http://docs.sun.com`

[124] A. Tamches and B.P. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels", *3rd Symposium on Operating Systems Design and Implementation*, New Orleans, February 1999.

[125] A. Tamches, Personal communication.

[126] D.M.J. Tax, "One-class classification", PhD thesis, Delft University of Technology, `http://www.ph.tn.tudelft.nl/~davidt/thesis.pdf`, June 2001.

[127] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience", *Concurrency and Computation: Practice and Experience* **17**, 2–4, February-April, 2005, pp. 323–356.

[128] R.S. Wallach, "Gemini Lite: A Non-intrusive Debugger for Windows NT", *4th USENIX Windows Systems Symposium*, Seattle, WA, August 2000.

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

[130] M. Weiser, "Programmers use slicing when debugging", *Communications of the ACM* **25**, 7, 1982, pp. 446–452.

[131] M. Weiser, "Program slicing", *IEEE Transactions on Software Engineering* **10**, 4, July 1984, pp. 352–357.

[132] R.F. Van der Wijngaart, "NAS Parallel Benchmarks Version 2.4", *NAS Technical Report NAS-02-007*, October 2002.

[133] Wine home page, `http://www.winehq.org`

[134] R. Wismuller, J. Trinitis, and T. Ludwig, "OCM — A Monitoring System for Interoperable Tools", In *SIGMETRICS Symposium on Parallel and Distributed Tools*, Welches, OR, August 1998.

[135] K. Yaghmour and M.R. Dagenais, "Measuring and Characterizing System Behavior Using Kernel-Level Event Logging", *USENIX Annual Conference*, June 2000.

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

[137] V. Zandy, "Force a Process to Load a Library", `http://www.cs.wisc.edu/~zandy/p/hijack.c`

[138] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs", *10th International Symposium on the Foundations of Software Engineering*, Charleston, SC, November 2002.