CPU Performance Counter-Based Problem Diagnosis for Software Systems

Keith A. Bare

CMU-CS-09-158

September 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Priya Narasimhan, Chair
Gregory R. Ganger

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This work has been partially supported by the National Science Foundation CAREER grant CCR-0238381 and CNS-0326453, and the Army Research Office grant number DAAD19-02-1-0389 (”Perpetually Available and Secure Information Systems”) to the Center for Computer and Communications Security at Carnegie Mellon University.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: black-box instrumentation, CPU performance counters, fingerpointing, problem diagnosis, support vector machines, Perfmon2, three-tier web-applications

This thesis is dedicated to everybody that has helped me make music such a wonderful part of my life. This includes, but is not limited to, Sandra Pease, Larry Wilhelm, Philip Simon, Mary Ulrey, Jan Duga, Paul Gerlach, Maria Sensi-Sellner, Susan Sands, and my loving parents Keith and Mary Lou Bare.

Abstract

Faults that occur in distributed software systems, such as e-commerce applications, are often costly in terms of lost revenue, but can be difficult to discover manually. Problem diagnosis tools attempt to detect, and often localize, faults that occur in distributed software systems. The goal is to detect problems soon after they occur, and rapidly notify an operator or automatically fix the issue as quickly possible.

Trade-offs are involved in designing such tools. A good tool should be accurate, and should have low overheads, to minimize adverse effects to the monitored system. Often there is a trade-off between these two goals. Application-level data can often lead to very accurate, fine-grained diagnoses, but at a high cost in terms of reduced system performance. Metrics collected from the operating system are less expensive to collect, but usually are only suitable for coarse fault localization, usually to a specific machine.

This thesis explores a data source that has only had limited use in problem diagnosis tools: CPU performance counters. Instrumentation based on these performance counters can be collected with very low overheads, and provides information with expressive power similar to data collected from the operating system. This data source is evaluated experimentally, in conjunction with a variety of simple analysis algorithms, via synthetic fault-injection experiments against a realistic 3-tier auction web-application.

Experimental results indicate that CPU performance counter-based approaches are able to consistently detect, and in some instances localize faults, even when only simple analyses are performed. Given the low cost to collect data from the counters, and the fact that this data need not be tied to a specific application or operating system, this work demonstrates the viability of such approaches to problem diagnosis.

Acknowledgments

I would like to acknowledge and thank the people that guided and supported me through my dizzying explorations of academic research.

First and foremost, I'd like to thank my adviser, Prof. Priya Narasimhan for her enthusiasm and support. She started and nurtured my interest in fingerpointing, and provided many helpful comments to me as I was writing this document.

I also would like to thank Prof. Greg Ganger, for expressing interest in my work, and agreeing to serve on my thesis committee.

The fingerpointing research group, consisting of Michael Kasick, Soila Kavulya, Eugene Marinelli, Xinghao Pan, and Jiaqi Tan, has been an invaluable group of friends and colleagues. Soila Kavulya provided me with code that served as basis for the fault-injector I used in this thesis. Michael Kasick was able to help me on several occasions when I faced difficulties. His helped ranged from discussion of the vagarities of the R language to suggestions on how to deal with overheating computers. He has been a very good friend through the duration of my Carnegie Mellon experience.

My parents, Keith and Mary Lou Bare deserve more thanks than I could ever give them. They played a huge role in making me the person I am today. Never more than a phone call away, they always gave me support when I needed it most.

I'd like to thank my former housemate, Michael Dille. He helped me maintain a reasonable perspective on life as a graduate student, and was also a great person to bounce ideas off of. Conversations with him led to the PCA-based metric selection heuristic, and the use of SVMs for data analysis.

Finally, Stéphane Eranian deserves acknowledgment, as the author and designer of Perfmon2. His freely-available software played a significant role in the research leading to this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's software systems are getting larger and increasingly complicated. Seemingly simple actions can involve processing on many different machines. As a result, it becomes increasingly likely that a given component of such a system might fail, and potentially cause a cascade of failures that affects the entire system. Studies [31, 36] demonstrate that a wide variety of these sorts of failures occur in e-commerce applications. Failures that cause service outages can result in large monetary losses. For example, an hour-long PayPal outage due to a failed piece of networking equipment on August 3, 2009 may have prevented up to $7.2 million in customer transactions [41]. Indirect costs in terms of dissatisfied customers are harder to quantify, but also present a serious concern.

Automated problem diagnosis techniques focus on alleviating the prohibitive cost of downtime by continuously monitoring important software systems and localizing/diagnosing the root-cause of failures when they occur. Knowledge of the root-cause of a failure is helpful to a system administrator, as it eliminates or reduces the time required to manually determine the cause of the failure, and allows the administrator to mitigate and/or repair the failure more quickly and effectively. This has the potential to reduce the duration and cost of any resulting downtime.

Current problem diagnosis techniques work at varying granularities. Techniques such as path-analysis [14, 19, 29, 39], latency analysis [8, 28] and performance-data analysis [4, 15, 16, 26, 27, 35, 45] all aim to automatically identify the components or nodes that are the source of the problem, thereby saving the system administrator the time and the manual effort in isolating the root cause of the problem.

Distributed software systems are comprised of multiple layers, each of which can be an effective source of instrumentation that then serves as input for problem diagnosis. Application-specific (white-box) instrumentation is the least general. It requires a specific understanding of the application's operation. Frequently it involves modifications to the application code in order to extract the relevant data. On the other hand, white-box instrumentation does have its advantages—inevitably, application-specific data is often required to trace a failure back to root-causes as specific as line of source-code. However, given the application-specific nature of white-box instrumentation, it needs to be redone for any new application and also requires an understanding of the semantics and functionality of the application, which reduces its value as a generic technique. Some problem diagnosis approaches have successfully used white-box approaches. For example, Pip [39] uses

explicit models of application behavior to determine whether or not annotated distributed systems are functioning properly. X-Trace [19] uses application-level instrumentation to determine causal paths in network protocols, which can be useful in determining which node returned an erroneous value, even when several are involved in handling requests.

Other types of instrumentation are completely application-agnostic (black-box) in that the instrumentation is transparent to the application, does not involve application-level modifications, and does not need an understanding of the application's semantics or structure. Black-box instrumentation generally takes the form of performance data collected by the operating system. This includes data such as context-switch rate, CPU usage, network-traffic rate, page-fault rate, etc. Black-box instrumentation is generic, and can be readily collected for any application in the same manner. Some problem diagnosis approaches have successfully used black-box approaches. For example, [15, 16] use performance metrics from the operating system to predict when service-level objectives (SLOs) are violated in web-applications. [35] uses peer-comparison of operating system performance metrics to identify nodes experiencing memory leaks, process hangs, and packet loss in two replicated systems.

Library/middleware instrumentation takes a middle ground. These gray-box approaches focus on instrumentation that can potentially be reused for an entire class of applications. For example, instrumentation that collects garbage-collection statistics at the Java Virtual Machine (JVM) level can be reused for any Java application. Other examples of gray-box instrumentation include path-tracing based on modified communications layers and management metrics provided by middleware. Some problem diagnosis approaches have successfully used gray-box approaches. For example, PinPoint [11, 14] and [29] use request tracing to diagnose Java exceptions, endless calls, and null calls in three-tier web-applications.

Due to its general, application-independent nature and the ease with which black-box data can be obtained, this thesis develops a black-box approach to problem diagnosis. However, it is not novel in its use of black-box instrumentation for problem diagnosis; many other black-box diagnosis techniques exist.

The novelty of this thesis lies in its use of a new black-box instrumentation source, specifically, CPU performance counters, for problem diagnosis. A CPU's performance monitoring unit (PMU) monitors the processor for the occurrence of specific performance-related events, incrementing performance counter registers when the events occur. Examples of events that can be monitored include cache misses, memory prefetches, branch mispredictions, and interrupts. These all provide low-level information about processor state as it executes code, which would otherwise only be obtainable by simulation. Traditionally, CPU performance counters are used for profiling and bottleneck analysis [5, 9, 32, 43] on a single processor. For example, tools like OProfile [32] can be used by application developers to determine which functions a program spends most of its execution time in. However, there has been little or no work done in correlated traces of data from these counters across the processors in a *distributed* system, with the aim of localizing a performance problem.

Thesis statement. The thesis statement is as follows:

> *CPU performance counters provide low-overhead, transparent instrumentation capable of supporting the black-box diagnosis of performance problems in three-tier distributed systems for e-commerce.*

Validation of thesis statement. The remainder of this thesis attempts to validate the thesis statement through experimental fault-injection studies on a well-known, open-source, three-tier e-commerce system, the Rice University Bidding System (RUBiS) [12], an auction site prototype modeled after eBay. The validation includes the following:

- Developing an instrumentation framework for extracting CPU performance counter data from an executing instance of RUBiS;
- Employing fault-injection in a methodical, repeatable manner, to induce various performance problems in an executing instance of RUBiS;
- Performing different kinds of analyses on the gathered fault-free and fault-induced traces of the gathered CPU performance counter data, with the aim of enabling offline problem localization;
- Understanding the impact, in terms of reduced throughput, caused by collecting CPU performance as RUBiS is executing.

Contributions. To the best of the author's knowledge, there is no previous work that leverages CPU performance counters for black-box problem diagnosis. This thesis' initial, empirical exploration of problem diagnosis using CPU performance counter data is a significant novel contribution in the area of dependability. Ancillary, but related, contributions include:

- A technique for determining which countable processor events are the most relevant/useful for problem diagnosis;
- Simple analysis techniques capable of anomaly detection and problem localization;
- Insights about the strengths and limitations of using CPU performance counter data for problem diagnosis.

Organization of thesis. This thesis is organized as follows: Chapter 2 provides the necessary background, by describing related work in both the context of CPU performance counters and the context of problem diagnosis. Chapter 3 provides a synopsis of the approach and covers the details of the target system, RUBiS. Chapter 4 describes the instrumentation (the CPU performance counters), the type of data collected, and the significance of the data. Chapter 5 provides the details of the experimental methods. Chapter 6 outlines the analytical techniques used to examine the data collected. Chapter 7 describes the results and the insights obtained from this new approach. Chapter 8 concludes with the lessons learned and directions for future work.

Chapter 2

Related Work

Related work falls into two main categories: applications of CPU performance counters, and techniques for problem diagnosis in three-tier web-applications. Performance counters are typically used for code profiling, which, in a sense, is a way of finding performance problems in code. However, there are differences. Problem diagnosis work is typically designed to apply to an entire distributed system, whereas most profiling techniques only apply to a single executable program on one machine. Additional constraints also sometimes apply to problem diagnosis tools. Some examples include low operating overheads, ability to function without access to the monitored system's source code, and limited presentation of data to avoid overwhelming administrators.

2.1 CPU Performance Counters

Nearly any software that makes use of CPU performance counters uses some sort library and/or kernel API to access and control the counters. This research uses Perfmon2 [34], as it is a well-featured, low-level interface that runs on the Linux systems that were available for experiments. There are other interfaces that have similar features. perfctr [33] is a performance counter driver for Linux some previous work has used, which has now for the most part been superceded by Perfmon2. PMAPI [37] is a library that runs on machines with POWERPC processors running AIX. In high performance computing, many tools use PAPI [17], which was designed to be portable across systems and processors. The work in this thesis could have used one of these alternative interfaces, assuming the interface supports event multiplexing and the abilities to program, start, read, and stop the processor's counters.

Most approaches that use a processor's performance monitoring features do some sort of statistical profiling. Anderson et al. [5] discuss a continuous profiling framework, one of the earliest examples. The main idea is to profile programs with low overheads. To do this, interrupts are generated after every n instances of a specific event occur. The interrupt handler examines the instruction that had been executing when the interrupt occurs, and attributes the event to that instruction. This allows a programmer to determine which instructions frequently cause certain events, and the runtime overhead can be decreased by increasing the value of n. Anderson uses the collected information to attribute pipeline stalls to specific instructions, and give a likely cause

for the stall. Other work that takes this approach includes OProfile [32], the sampling features of Perfmon2 [34], and Intel's VTune Performance Analyzer [24].

In [9], Azimi et al. explore difficulties that often arise when hardware performance counters are used. One difficulty given is that only a small number of counters are typically supported by processors. The solution provided is counter multiplexing, and it is empirically demonstrated that multiplexing several logical counters onto a single physical counter usually does not severely affect the accuracy of the data collected. The multiplexing approach is then used to collect data for attributing pipeline stalls to specific functional units. This thesis makes use of similar multiplexing, and uses the extra data collected to provide more features to the analysis algorithms used in performing problem diagnosis.

Virtual machines, such as the JVM, can affect code execution and performance in interesting, seemingly strange ways. Exploration of these phenomena yields work involving CPU performance counters that is in many ways similar to problem diagnosis. However, the focus is on understanding performance characteristics on a single processor, rather than diagnosing problems in distributed systems. Eeckhout et al. [18] explore how different JVM implementations and data sizes can affect the architectural events generated by the same programs. The main conclusion made is that many more things affect performance characteristics of Java programs than programs written in languages with simpler runtime environments like C and C++. The authors of [43] do, however, achieve success using hardware performance counters to explain certain behaviors in Java applications, included performance improvement over time due to JIT optimization, and decreased performance before garbage collection. The next year, Vertical Profiling [21] is introduced, and makes use of data collected by software in addition to the hardware counter data. The authors claim that hardware counters alone weren't expressive enough to explain some phenomena. However, software data often requires code modification to collect, which may be time consuming or impossible. For this reason, this thesis focuses on problem diagnosis using hardware counters alone.

2.2 Problem Diagnosis for Three-Tier Web-Applications

A variety of approaches have been explored to build problem diagnosis tools for three-tier web-applications. Approaches can be categorized by the source of data used to perform fault detection and localization. Typical sources of data include request-tracing, management metrics reported by server software, and black-box performance maintained by the OS. The author is not aware of any other approaches that used data collected CPU performance counters.

Request tracing requires information identifying which application components a given user-initiated request touch. If the monitored application does not provide this information, the application or its middleware must be modified. This modification may be difficult or impossible if the application is large, or its source is not available. CPU performance counter-based approaches, on the other hand, do not require information about the components user-initiated requests touch. Pinpoint [14] detects faults based on instrumented middleware that reports when Java exceptions occur. Records indicating which components were used for a request, and whether or not the request failed are fed into a clustering algorithm. Components in clusters associated with failed requests are reported as the likely cause of the failures. Candea et al. [11] extend Pinpoint to

proactively recover from failures using micro-reboot techniques. Khanna et al. [29] take a more complicated approach, using the path requests takes through application components. Probabilities are maintained estimating the reliability of components, the reliability of communications links between components, and how capable a component is it masking failures rather than propagating them. Based on the request path and the collected probabilities, the components most likely to have caused the fault are determined.

Management metrics are pieces of data relevant to performance maintained and reported by server software. These include things such as response times for different classes of requests, load-balancing weight assigned to certain machines, and load average. Many are specific to the server software used. Appleby et al. [8] use response time measurements for each component of the system. Transaction-level service level objectives (SLOs) are used to construct component SLOs, by dividing the transaction-level SLO into parts, based on dependency information (which can be estimated using management metrics via the approach in [3]) and typical response times for the components. When a fault occurs and a transaction-level SLO is violated, the fault can be localized by observing which components violated their component SLOs. In [4], the authors detect faults using change point techniques. Faults are detected based on rules specifying which management metrics should change as a result of the fault, and which should not. This approach is used to detect storm drain conditions in Trade 6. It is possible that the approach could be extended to other faults, however, a human with a priori knowledge of how the faults manifest would need to manually construct the rules. Jiang et al. present two approaches based on correlation. [27] explores additional regression methods to model correlations between metrics where a simple ordinary least squares (OLS) regression is insufficient. A fault is detected when a significant number of correlations appear to be violated, based on a Wilcoxon rank-sum test. [26] takes a similar approach, but uses clustering of normalized mutual information (NMI) values between metrics, rather than regression models. A fault is detected when changes in the entropy of a cluster are deemed significant by the Wilcoxon rank-sum test. Localization is attempted by scoring subsystems based on the Jaccard coefficient, potentially adjusting the scores to account for popular components.

Black-box instrumentation, typically in the form of performance metrics collected by the OS, has the benefit that it is not in any way tied to the software system being diagnosed. Cohen et al. explore this type of instrumentation. In [15], they use Tree-augmented Bayesian Networks (TANs) to predict when an SLO violation is occurring, based only on instrumentation data. Of course, this requires that SLOs be set, and that the SLOs are monitored when training the TAN model. The authors make an interesting observation: depending on the root cause of the SLO violations used for training, the TAN models select different metrics to predict future SLO violations. The work in [16] extends on this observation, developing a method for creating signatures for different types of faults. By comparing the current instrumentation values with historical signatures, Cohen et al. are able to identify faults that are likely to have caused a given SLO violation. While this thesis does make use of black-box instrumentation, it differs from the approaches discussed in three main ways: i) CPU performance counters are used as instrumentation, rather than OS-collected metrics, ii) there is no requirement to set SLOs, and iii) no attempt is made to identify the type of fault occurring.

Chapter 3

Approach

This chapter provides an overview of the approach taken in building a tool that uses CPU performance counter-based instrumentation to perform problem diagnosis. The first section clarifies the problem considered, by enumerating specific goals, non-goals, and assumptions. The second section presents the rationale behind CPU performance counter-based diagnosis. The third section provides a brief overview of the approach taken. Finally, the fourth section re-iterates the key points discussed in this chapter.

3.1 Problem Statement

Objectives. A tool for CPU performance counter-based problem diagnosis should achieve the following goals:

Low-overhead instrumentation. Instrumentation with high run-time overheads has the possibility of adversely affecting the target system-under-diagnosis. As a result, a good solution will provide CPU performance counter data with low overheads.

Support for online and offline data-analysis. In the field, it is desirable for a problem diagnosis tool to notify a system administrator of a potential problem as soon after a fault occurs as possible. In practice, this requires some type of online analysis that runs concurrently with the system-under-diagnosis. Thus, a good solution will perform data-analysis in an incremental manner, supporting offline analysis of execution traces for evaluation purposes while maintaining the possibility of an online implementation.

Tier-level localization. Fine-granularity diagnoses are likely to be the most useful to a system administrator—they minimize the amount of manual diagnosis that must be performed. In a three-tier e-commerce system, the tier from which an underlying fault originated is a natural choice of granularity. As a result, a good solution will localize performance problems to a faulty tier.

Scope. To limit this thesis to a tractable problem, the following explicit non-goals are adopted:

Novel or complex data-analysis algorithms. While necessary in the construction of an problem diagnosis tool, data-analysis algorithms are a secondary consideration. The focus is on validating

a novel source of instrumentation, so simple analysis techniques based on off-the-shelf tools are employed.

Fine-grained root-cause analysis. Some systems attempt to trace a problem back to a very fine-grained root-cause, such as a line of code or instruction. For example, [5] attributes pipeline stalls to specific instructions. The approaches in this thesis only attempt localization to a specific machine in a three-tier web-application.

Diagnosis of value faults. It may be possible for a system-under-diagnosis to fail in such a way that it still appears to be performing normally, while in actuality the system is returning incorrect or bogus values. A web-application meeting its SLO, but presenting the user with a page describing a Java exception would be an example of a value fault.

Combinations of multiple instrumentation sources. While it is possible to combine CPU performance counter-based instrumentation with other sources of data, that is not the focus of this thesis. Rather, this thesis explores the viability of CPU performance counters alone, as additional data sources could cloud the contribution CPU performance counters make.

Assumptions. This thesis assumes the following in the approach specified:

RUBiS is the only service running on the machines being diagnosed. CPU performance counter instrumentation data is generally collected as all processes in the system are running. As a result, other services that perform processing unrelated to RUBiS may cause what appears to be anomalous/faulty behavior, and lead to a false alarm.

Synchronized clocks. Timestamps are used to correlate measurements made on different machines temporally. The clocks in all the machines must be synchronized for this to work properly. Using the Network Time Protocol (NTP) is one solution for maintaining this synchronization.

Processor support for performance monitoring. It is obvious, but still worth noting, that this approach will only work on processors that have performance counters. However, performance monitoring features are ubiquitous, and are supported by all recent AMD and Intel processors.

3.2 Rationale

Why should CPU performance counter data be a useful source of data for problem diagnosis? The following insight hints at their viability. As a software system executes, and is functioning normally, it usually executes similar code paths for all requests of a given type. In most e-commerce systems under normal loads, it is likely that there is a stable request mixture. Since CPU performance counter events are based on activities that occur as code executes, it is assumed that *different blocks of code produce different proportions of events*. If faults are considered, this line of thought leads to the central assumption of this thesis: *When a fault occurs, different code executes, or the same code executes differently*. This phenomenon produces changes in the collected time-series of counter-value differences, a key observation that we exploit in the analysis techniques.

The assumption is that faults can be detected because they cause anomalous behavior in the CPU performance counters. However, other events may also cause anomalous behavior. One

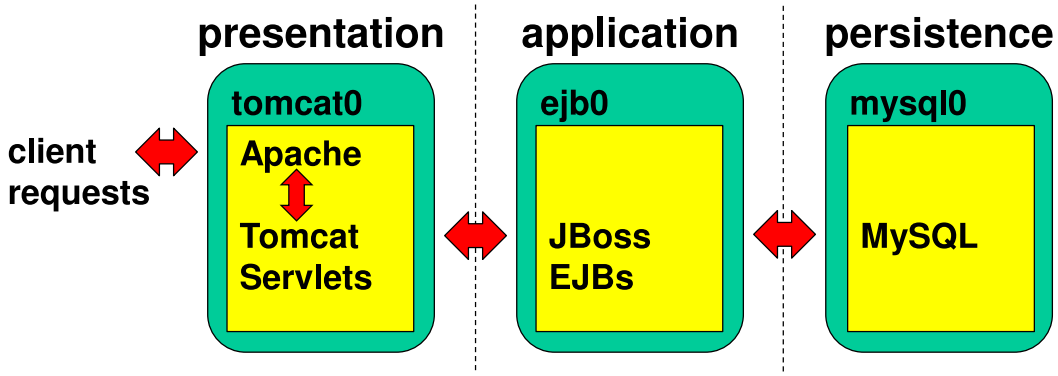

Figure 3.1: An overview of the three-tier EJB SessionBean version of RUBiS.

example of such an event is a change in the workload the system-under-diagnosis is experiencing. Differentiating between faults (truly anomalous behavior) and workload changes (different but correct behavior) is a challenge every problem diagnosis approach must face. Approaches that use application-level data can sometimes get around this difficulty by using an additional white-box metric that is known to be correlated with workload. Black-box approaches, such as the one developed in this thesis, often do not fare as well. One strategy is to introduce models of different modes of operation, one model per unique workload type. However, such an approach may introduce large numbers of missed fault-detections, as true faults may be mistaken for a workload change. For simplicity, this thesis acknowledges, but does not attempt to solve the problems associated with workload variation.

3.3 Synopsis of Approach

The approach taken to perform problem diagnosis involves i) periodically collecting accumulated event counts from CPU performance counter registers while a target system-under-diagnosis executes, ii) analyzing the collected data to determine whether a performance problem occurred, and iii) localizing the problem to the node on which the fault is occurring. The data-analysis stages, while currently only implemented as offline techniques for processing trace-data, were designed to operate incrementally, and thus support online operation in the future.

To validate the thesis statement in this diagnosis-centric study of hardware performance counter data, the author pursues an empirical approach through fault injection. RUBiS [12], an open-source, multi-tier e-commerce system, is selected as the target system-under-diagnosis. Specifically, the three-tier EJB SessionBean version was used. Figure 3.1 provides a high-level overview of the RUBiS setup. Faults to inject are selected from a set of performance problems, based both on resource contention and realistic problems (i.e., network communications problems and program bugs). These faults are injected to a specific tier of the system-under-diagnosis, as CPU performance counter data is collected. An analysis technique then attempts to localize the fault to the appropriate tier using the collected traces of performance counter data.

While this approach is validated on a single system-under-diagnosis (RUBiS) in this thesis,

many other systems could have been studied. The data-collection and data-analysis techniques are in no way specific to RUBiS, and are applicable to other target systems. Many of the faults injected are also generic, so large portions of the fault-injection framework will apply to other systems. Of course, it cannot be assumed that the experimental conclusions reached with RUBiS necessarily generalize to other systems.

3.4 Summary of Chapter

This chapter attempted to formulate the problem of performing CPU performance counter-based problem diagnosis with increased clarity. In that interest, the author enumerated goals, non-goals, and assumptions. The rationale, and key assumptions leading to the believed expressivity of CPU performance counter instrumentation were then covered. Finally, the target system and an overview of the approach to solving the problem were presented. More detail on the CPU performance counters, and techniques in which the data they provide is captured will be presented in the next chapter.

Chapter 4

Instrumentation

Three main components are involved in providing hardware performance counter-based instrumentation: the processor's performance monitoring unit (PMU), the kernel performance monitoring subsystem, and the `pfmon` program. The PMU does most of the real work; it increments special counter registers when certain architectural and micro-architectural events occur. Unfortunately, configuring the PMU typically requires the execution of privileged instructions. As a result, the kernel performance monitoring API is introduced as an intermediary, granting non-privileged programs limited access to the PMU's features. Finally, the `pfmon` program is used to collect and record the data generated.

4.1 CPU Support for Performance Monitoring

Modern processors all support some sort of performance monitoring capability. Designed to support profiling to optimize the performance of code execution, a processor's PMU supports a wide variety of functions. This thesis uses one of the simplest, most common performance monitoring features: the ability to count the occurrence of architectural and micro-architectural events via programmable counter registers.

The following steps are taken in using these performance counters. First, values are written to control registers. These values precisely specify which events should be counted, and include details, including but not limited to: i) privilege modes in which events are counted, ii) functional units to be monitored, and iii) level- or edge-triggered mode of operation. Initial counter values also must also be loaded into the counter registers themselves. Next, a bit is set to enable the counters. At this point, the processor increments the appropriate counter every time an event of interest occurs. If any counter overflows, the processor may generate an interrupt if configured to do so. Finally, the data is collected, either by periodically reading the counters' values, or by observing overflow interrupts.

The number of counters available for use, and the set of countable events is processor dependent. For example, the AMD Opteron family 0Fh processors used in this research support four programmable counters [2], while processors based on Intel's Core micro-architecture support three fixed-function counters, and two programmable counters [23]. While the events available

for counting vary between processor makes and models, available metrics are usually similar. For example, nearly every processor will support counting clock cycles, instruction retirement, and cache-related events. Moreover, processor manufacturers typically provide support for monitoring events that can lead to poor performance. This is so developers employing profiling techniques based on performance counters can discover and fix code that executes poorly on the processor.

Additional information on the performance monitoring features of processors is typically found in the processor vendor's documentation. For AMD processors, general information is available in section 13.3 of [1], with specific details in chapter 11 of [2]. Intel provides the information for its processors in chapter 18 and appendix A of [23].

4.2 Kernel Support for Performance Monitoring

Writing to control registers is a privileged operation, so programming the CPU's performance counters must be mediated by the operating system. Perfmon2 [34], a set of patches to the Linux kernel, provides access to the performance counters through a set of system calls. A wide variety of processor types and models are supported. As a consequence, the approach taken by this thesis is not necessarily restricted to AMD Opteron processors; it may be possible to obtain similar results on a different architecture or processor.

Several steps must be taken when using the Perfmon2 kernel API. First, a performance monitoring context, represented by a file descriptor, must be created. This can be associated with a processor, the current thread, or another thread that is being traced via `ptrace()`. If the context is associated with a thread, rather than a processor, Perfmon2 will virtualize the counters to the specified thread by saving and restoring the counter values when context switches occur. Next, the performance monitoring context can be configured. This is done using the `pfm_write_pmcs()` and `pfm_write_pmds()` system calls, which write to the PMU control and counter registers associated with the context, respectively. Once configured, the counters can then be enabled and disabled via the `pfm_start()` and `pfm_stop()` system calls. Finally, whenever it is desired, the current counter values can be obtained using the `pfm_read_pmds()` system call.

Perfmon2 is slightly more than a thin wrapper around PMU hardware. It supports some functionality not available from the processor alone. The most useful feature is the ability to multiplex multiple event sets onto a single physical set of counter registers. When multiplexing is enabled, Perfmon2 will change which logical counters are mapped to physical counters at a regular interval, and keep track of how long each logical counter is activated. Using this information, full counts for all the events can be extrapolated. [9] describes a similar technique for multiplexing, and evaluates accuracy of the resulting estimates. Given the limited number of counters available on most processors, multiplexing is essential. A second feature is per-thread virtualization of the performance counter registers. This makes it appear as a given thread is the only program using the CPU; only events that occur as the thread is executed get counted. Internally, this is implemented by saving and restoring the performance counters when context switches occur. This virtualization feature was not used for most experiments; generally data is collected in system-wide mode, where performance events that occur in any process contribute to the measured event counts.

4.3 The pfmon Program

The Perfmon2 project includes the `pfmon` program, a sample, well-featured program that uses the Perfmon2 API to program and control system PMUs. `pfmon` supports collecting the number of events accrued, either on a given CPU, on all CPUs, or for a specific thread. It records these counts to standard output or to file. In the thread-specific mode of operation, it has the ability to track lifecycle events (i.e., calls to `fork()`, `exec()`, and `clone()`), and also monitor newly created forks and threads.

This program served as a good starting point for data collection for offline analysis. However, two small changes had to be made to facilitate using the collected system-wide data for problem diagnosis. The first modification added timestamps to collected counts. There was pre-existing support for dumping system-wide event counts accumulated at regular intervals, but this data was not annotated with time stamps. The second change made `pfmon` more suitable for background data collection. The program was designed for interactive use, so it collects data until the user presses return. This causes problems when `pfmon` is run without an input source, so the `getchar()` call used to wait for input was replaced with a call to `pause()`, to wait for a signal instead.

Unless otherwise noted, to collect data for experiments, `pfmon` is started in system-wide mode 10 seconds after the services on each of the machines providing RUBiS' three tiers. It is configured so that the CPU performance counters only count events that occur in user mode. `pfmon`'s output, the counter value difference observed every second, is written to two files, one for each processor core. To ensure the timestamps from each machine are comparable, `ntpdate` is executed prior to each experiment. After an experiment completes, the output files are copied to a central location and merged.

4.4 Summary of Chapter

This chapter presented and described the components used for CPU performance counter-based instrumentation. These components include physical hardware in the CPU, kernel services, and a user program. The net result is a stream of data that can be fed into a variety of analysis algorithms. The author claims that this data is useful, and will proceed to evaluate the data in the context of synthetic fault-injection experiments.

Chapter 5

Experimental Methods

The goal of this thesis is to validate the hypothesis that CPU performance counters provide data that can be used to detect problems that occur in software systems. This chapter will explain common features of the experimental methods used to empirically evaluate the hypothesis. RUBiS, an eBay-like three-tier auction web-application, is the targeted software system. During each experiment, the workload generator included with RUBiS emulates many concurrent clients while faults are optionally injected into one of RUBiS' tiers. Measurements from the CPU performance counters are taken every second as the experiment runs and the collected data is then fed into an analysis algorithm.

5.1 System Setup

The Rice University Bidding System (RUBiS), described in [12], is a three-tier a web-application that is believed to be similar in architecture and design to real e-commerce applications. It provides an auction web site modeled off of eBay. While there are many versions of RUBiS (it was originally written to compare the performance of different application servers and implementation techniques), this thesis only used the Enterprise Java Bean (EJB) SessionBean version of RUBiS. The PHP and Servlet versions were not used because they do not separate presentation and application logic, and hence only have two tiers. The initial decision to use the SessionBean version over the other EJB versions of RUBiS was arbitrary, although [12] demonstrates that using the SessionBean version is a good choice in practice; the SessionBean RUBiS implementation achieves better performance than any of the other EJB-based implementations.

Three-tier designs separate distinct concerns of application design. They consist of a front-end or presentation tier, an application tier, and a persistence or database tier. The first tier is responsible for interacting with users, and formatting information. The second tier handles application logic, and the third tier is responsible for retaining and storing data. In the case of the EJB SessionBean version of RUBiS, the presentation tier is handled by a web server, a servlet container, and servlets. The application tier is handled by an EJB container and EJB Session Beans. Finally, the persistence tier is handled by a Relational Database Management System (RDBMS).

5.2 System Configuration

For all experiments, a set of four identical machines was used. Three machines were dedicated to RUBiS' three tiers. The fourth was used to simulate clients. The machines each have a dual-core 2.8GHz AMD Opteron processor and 4GB of RAM. All were connected to a gigabit ethernet network via a Broadcom NetExtreme ethernet interface. Each system ran Debian GNU/Linux with a 2.6.26 kernel, which was patched to support the Perfmon2 API.

Version 1.4.3 of RUBiS was used for all experiments. The machine hosting the front-end tier was configured with Apache Server [6] version 2.2.9 and Tomcat [7] 5.5.26. Apache Server was responsible for serving static content, and proxying requests for dynamic content to Tomcat via `mod_jk` using AJP, Tomcat's custom protocol. The application tier was hosted by JBoss Application Server [25] version 4.2.3.GA's EJB container. Finally, MySQL [30] 5.0.51a was used to provide the database tier. All Java was run under Sun Java 1.5.0.

Very few changes were made to the default configurations of the software used. Configuration changes were mainly limited to providing the components the ability to communicate with each other. For example, Apache Server had to know to proxy certain requests to Tomcat. The RUBiS servlets running in Tomcat had to be able to communicate with the application server. JBoss had to be configured to use MySQL for persistence, and MySQL was adjusted to support remote connections. The Java heap size for Tomcat was increased to 512MB, while JBoss ran with the defaults set by its startup script.

There are a few interesting things to note about this experimental setup. One is that there is no replication or redundancy. This is not necessarily realistic; most important services would use some form of redundancy to handle higher loads, better be able to cope with problems, and reduce downtime. However, it was convenient to use a configuration without replication or redundancy, due to resource constraints. Such configurations actually make it more difficult to localize problems. When a service is replicated three or more times, peer-comparison approaches (such as [35]) provide an easy mechanism to determine which machine is experiencing a fault; it's probably the one that's behaving differently, since all of the replicas should ostensibly be acting the same. Even if services are just duplicated to handle higher loads, peer comparison approaches may be possible if requests get distributed to the machines such that each machine is handling a similar workload. The second point of interest is that all the machines are running different software, and performing different functions. Despite the fact that three machines are being used, peer-comparison between them is not meaningful.

5.3 Event Selection

The AMD Opteron processors used for the experiments define a large number (nearly 90) of countable events. These are architectural and micro-architectural events, including things like memory and cache accesses, branch instructions, and pipeline stalls. AMD documents the complete list in chapter 11 of [2]. Given such a long list of potential metrics, the task of selecting metrics most useful for problem diagnosis is somewhat daunting. Three steps were taken in selecting the events that were most useful for diagnosis purposes. First, a large candidate set of events that seemed

potentially relevant was constructed. Second, plots were made, showing how the counter values vary over time under a emulated workload and with three simple fault types injected. The plots were used to manually identify a second candidate set of event counts that visibly reacted to the faults and workload. Finally, metrics from the second candidate set were ranked using a method based on Principle Component Analysis (PCA) [42].

AMD's documentation gives brief descriptions of each event countable by the CPU. Many seemed completely irrelevant to problem diagnosis in a 3-tier web-application. For example, some involved floating point operations. Those events were eliminated from the list, since web-applications generally only use integer math. On the other hand, if one was interested in performing problem diagnosis on scientific computations, use of the floating point events should probably be re-considered. The second category of events eliminated involved the memory controller built-in to the Opteron processor. Counting these events may have provided useful information but using them would have caused implementation difficulties, due to constraints on their use. One constraint is that events can only be counted when the processor is not halted. To count such events accurately, Linux's idle thread would need to be adjusted to busy wait, rather than use the HLT instruction. Such events can also only be counted by a single processor core. This conflicts with the requirement that other events all be counted on both cores, to ensure that no events are missed.

Selecting events by their descriptions still resulted in a large set of metrics, 47 of the 90 supported by the processor. This is still too many, given that only four can actually be collected at a time on the Opteron processors. Multiplexing is possible, but it seemed likely that 12-way multiplexing would significantly reduce the accuracy of the counts. To evaluate which metrics seemed to convey the most useful information in practice, several executions of the emulated workload described in section 5.4 were performed, both without fault-injection and with injected cpuhogs, diskhogs, and memoryhogs (see section 5.5 for more details). Every metric was then plotted with respect to time. The resulting graphs were manually examined to distinguish between metrics that were definitely affected by workload changes and faults, and those that are noisy or completely unaffected. The result was a shorter list of 24 event types.

Three examples of graphs that were produced have been included. In all graphs, dotted vertical lines indicate changes in the workload generated by the RUBiS client emulator, and solid vertical lines indicate the start and end of the fault-injection period. The "ejb0" label refers to the application tier, the "mysql0" label refers to the persistence tier, and the "tomcat0" label refers to the presentation tier. Figure 5.1 demonstrates how a cpuhog fault causes a significant decrease in the RETIRED_BRANCH_INSTRUCTIONS, as different code with different branching characteristics begins executing. Figure 5.2 demonstrates how a memoryhog fault causes a momentary spike in cache misses, due to the memoryhog allocating and zeroing a large amount of memory, trashing the cache. Finally, figure 5.3 demonstrates how some event counts just report noise that is completely unrelated to the workload and fault-injection.

This reduction is much better than the approximately 90 events supported by the Opteron processors, or even the original reduction to 47 seemingly relevant events. However, it would be ideal to have a small set of 4 to 12 metrics, so only a small amount of multiplexing is necessary. However, all the metrics at this point were well-affected by faults. As a result, the goal was to determine which of events were most affected by the faults. That is, to find subsets that feature the "best" performance counter events.

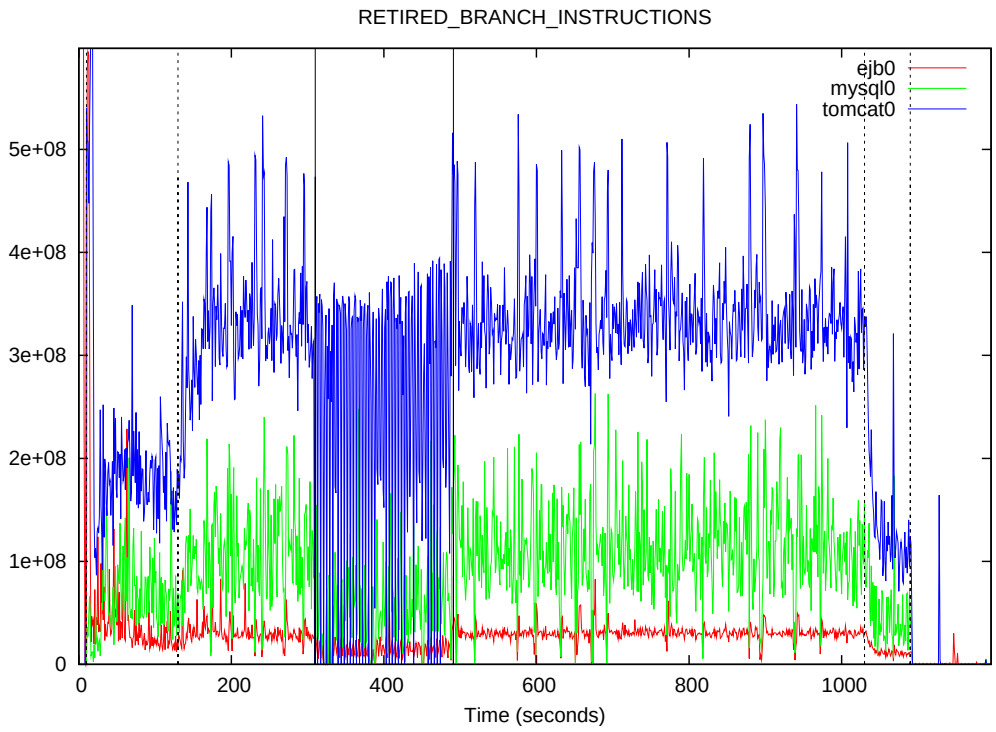

Figure 5.1: Plot of a cpuhog injected on tomcat0.

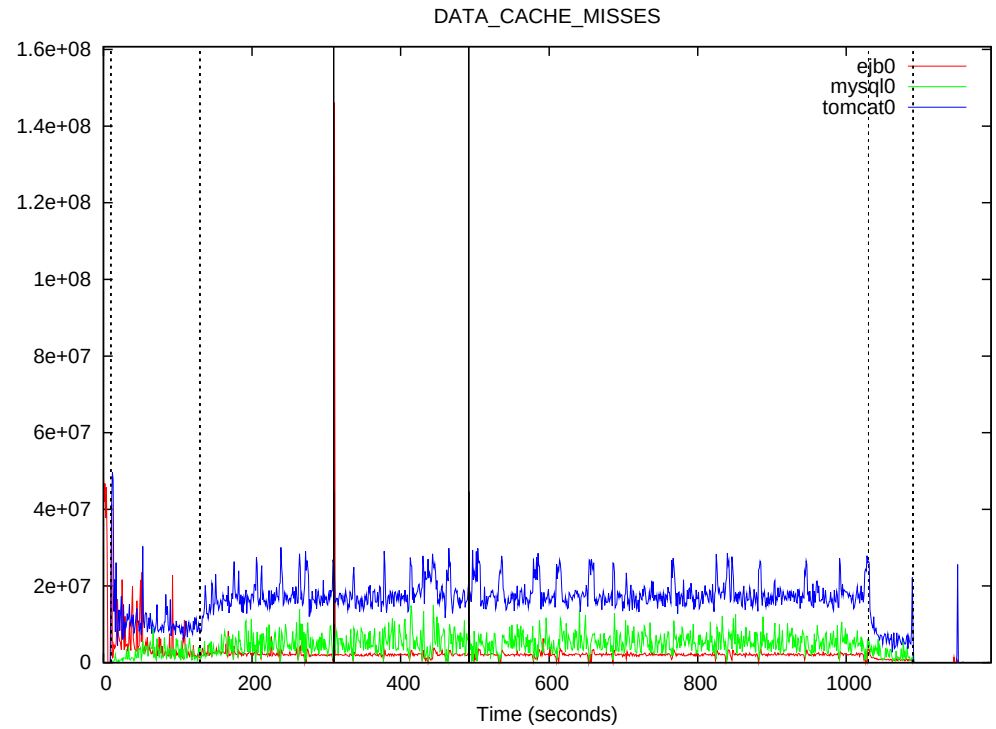

Figure 5.2: Plot of a memoryhog injected on ejb0.

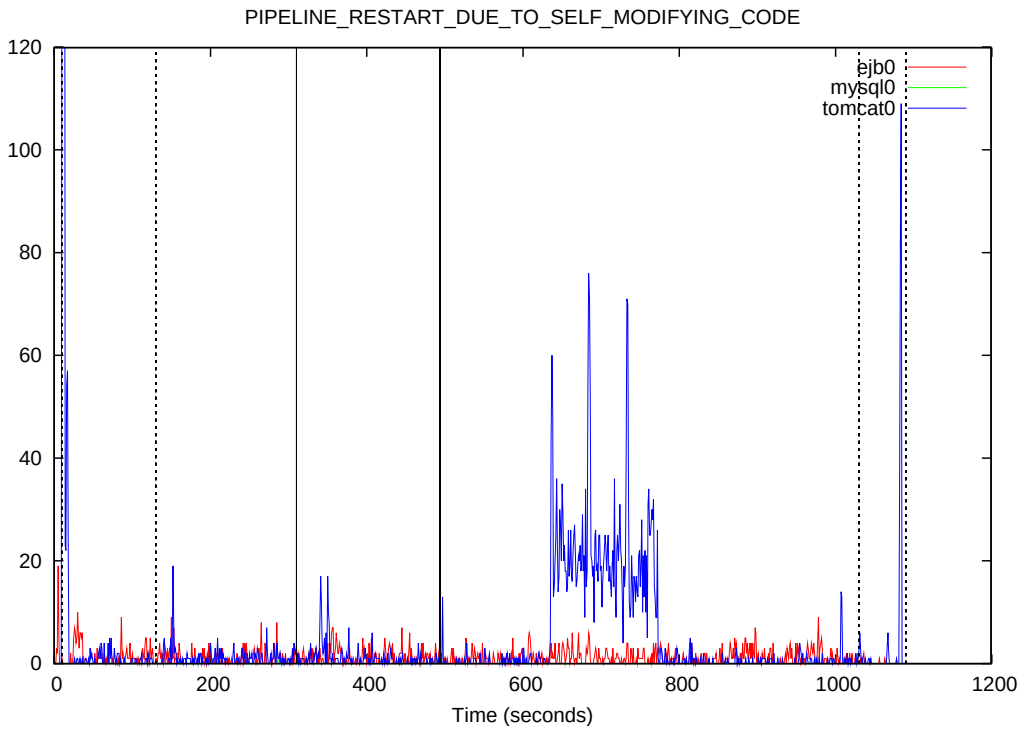

Figure 5.3: Plot of a diskhog injected on mysql0.

Principal Component Analysis (PCA) is a technique for analyzing data with high dimensionality. One typical use is to reduce the dimensionality of data. Given m vectors of length n, PCA attempts to find a transformation of the data to a new orthonormal basis with the property that the variance of the first dimension in the new orthonormal basis is maximized, and each subsequent dimension maximizes its variance, given the constraint that it must be orthogonal to the previous dimensions. Thus, taking the first k dimensions in the transformed space maximizes the variance retained when reducing the original data to k orthogonal dimensions. Additional conceptual information on PCA, and discussion on its computation is given in [42].

Note that the typical use of PCA, dimensionality reduction, is not particularly helpful in this thesis. The goal is to collect fewer metrics at the beginning, not to transform the data into a lower-dimensional space. However, the transformation matrix PCA produces does contain information that is helpful. Each dimension in the new basis found by PCA is a linear combination of the original dimensions, counter events in this case. As a result, the weight of a counter event in the PCA-generated transformation is indicative of the counter event's importance. Moreover, selecting counter events with large contributions to the first several dimensions of the PCA-generated basis retains more of the original data's variance. A heuristic algorithm for ordering counter events based on these observations is given in figure 5.4.

The algorithm was applied to two sets of data, all of which was collected without using multiplexing. Since multiplexing was not used, a set of experiments is defined to be enough experiment runs to collect data for each of 24 counter events. The first set of data, referred to as standard, consisted of data collected from a set of fault-injection experiments for each possible fault, as well

```
1: procedure RANKMETRICS(names, T)
2:     ▷ names_i is the name of the ith original metric
3:     ▷ T is the transformation matrix generated by PCA
4:     while NROWS(T) > 1 do
5:         j ← argmax_{i=1}^{NROWS(T)} |T_{i,1}|
6:         OUTPUT(names_j)
7:         T ← DELETEROW(T, j)
8:         names ← DELETE(names, j)
9:         for all i ∈ {1, ..., NROWS(T)} do
10:            if |T_{i,1}| > |T_{i,2}| then
11:                T_{i,2} ← T_{i,1}
12:            end if
13:        end for
14:        T ← DELETECOLUMN(T, 1).
15:    end while
16:    OUTPUT(names_1)
17: end procedure
```

Figure 5.4: Heuristic algorithm for ranking metrics based on PCA.

as a single set of fault free experiments. All experiments used 1000 simulated clients, and a 15 minute steady-state period. The second set, referred to as w/ extra faultfree used the same data as standard but additionally used two sets of fault free experiments with 90 minute steady-state periods. Both sets of data were examined, because it was not initially clear whether using extra fault-free training data would help or hinder performance when detecting and localizing faults.

The fact that data collected from all instrumented machines is fed into the selection algorithm introduces the caveat that the resulting list of metrics distinguishes between the same counter event collected on different machines. To remove the distinction between machines, the counter events are ranked based on their first occurrence in the selection algorithm's output. The resulting ordered counter event lists for both the standard and w/ extra faultfree sets of data are displayed in table 5.1.

5.4 Emulated Workload

RUBiS includes a client emulator, a Java program that simulates the workloads that human clients might generate. This program simulates clients by having many threads perform interactions with the RUBiS deployment. A thread will perform an interaction, wait for some amount of think time, and then choose another interaction to perform based on the previous interaction and a probability matrix. Session and think times are based on those from the TPC-W benchmark [44]. Some additional discussion of the RUBiS client emulator is given in [12].

Unless otherwise noted, the same RUBiS client emulator configuration was used for all experiments. In this configuration, a single dedicated machine emulates 1000 clients. This number of clients was chosen as it resulted in approximately 70% load on the front-end, which was the bottleneck machine. Client transitions were made based on the default browse/buy transition matrix.

rank	standard	w/ extra faultfree
1	L1_ITLB_MISS_AND_L2_ITLB_MISS	L1_ITLB_MISS_AND_L2_ITLB_MISS
2	RETIRED_BRANCH_INSTRUCTIONS	RETIRED_BRANCH_INSTRUCTIONS
3	RETIRED_TAKEN_BRANCH_INSTRUCTIONS	RETIRED_TAKEN_BRANCH_INSTRUTIONS
4	RETIRED_MISPREDICTED_BRANCH_INSTRUCTIONS	RETIRED_MISPREDICTED_BRANCH_INSTRUCTIONS
5	REQUESTS_TO_L2.ALL	REQUESTS_TO_L2.ALL
6	INSTRUCTION_FETCH_STALL	DATA_CACHE_REFILLS_FROM_L2
7	DATA_CACHE_REFILLS_FORM_L2	DATA_PREFETCHES.ATTEMPTED
8	DATA_CACHE_REFILLS_FROM_SYSTEM.ALL	DISPATCH_STALL_FOR_BRANCH_ABORT
9	DISPATCH_STALL_FOR_BRANCH_ABORT	DISPATCH_STALL_WAITING_FOR_ALL_QUIET
10	RETIRED_BRANCH_RESYNCS	RETIRED_BRANCH_RESYNCS
11	DISPATCH_STALL_WAITING_FOR_ALL_QUIET	RETIRED_FAR_CONTROL_TRANSFERS
12	DATA_PREFETCHES.ATTEMPTED	DATA_CACHE_REFILLS_FROM_SYSTEM.ALL
13	DISPATCH_STALL_FOR_SEGMENT_LOAD	DISPATCH_STALL_FOR_SEGMENT_LOAD
14	RETIRED_FAR_CONTROL_TRANSFERS	INSTRUCTION_FETCH_STALL
15	DATA_CACHE_ACCESSES	DISPATCH_STALL_FOR_RESERVATION_STATION_FULL
16	DISPATCH_STALLS	DATA_CACHE_ACCESSES
17	INSTRUCTION_CACHE_FETCHES	DISPATCH_STALLS
18	DISPATCH_STALL_FOR_LS_FULL	DISPATCH_STALL_FOR_LS_FULL
19	DISPATCH_STALL_FOR_RESERVATION_STATION_FULL	INSTRUCTION_CACHE_FETCHES
20	RETIRED_UOPS	CPU_CLK_UNHALTED
21	D1_DTLB_MISS_AND_L2_DTLB_MISS	L1_DTLB_AND_L2_DTLB_MISS
22	RETIRED_INSTRUCTIONS	RETIRED_UOPS
23	DATA_CACHE_MISSES	RETIRED_INSTRUCTIONS
24	CPU_CLK_UNHALTED	DATA_CACHE_MISSES

Table 5.1: Counter event lists, as ordered by the PCA-based algorithm.

When the RUBiS client emulator runs, it features three distinct phases of operation: a warm-up phase, a steady-state phase, and a cool-down phase. Experiments typically used a 2 minute warm-up, a 6 minute steady-state, and a 1 minute cool-down.

5.5 Fault Injection

Three broad categories of faults were injected: resource contention caused by an external program, internal faults that could potentially be caused by program bugs, and packet loss that reduces network performance. Injected faults are activated 4 minutes after the emulated workload starts, which is 2 minutes into the workload emulator's steady-state period. If the effects of the fault can be reversed, the fault is de-activated 90 seconds later. The resource contention and packet loss faults can be injected into any of the 3 tiers, while the internal faults are specific to a certain service. Each injected fault is described in table 5.2.

5.6 Summary of Chapter

To reiterate, CPU performance counter data is periodically collected as RUBiS, a three tier web-application runs under an emulated workload and simulated faults. Three machines are involved, a front-end running Apache server and Tomcat, an application server running JBoss, and a database server running MySQL. The collected data is saved to disk for future analysis.

resource contention	
cpuhog	An external program consumes 50% of all CPU time by performing matrix multiplication under a real-time scheduler.
diskhog	A script uses up I/O bandwidth by repeatedly writing a 1GB file in 102MB blocks to disk with `dd`.
memoryhog	An external program allocates, zeros, and locks 2GB of RAM.
internal	
dblock	A script locks the `items` table in the RUBiS database, and holds the lock for the duration of the fault-injection period.
jbas994	Simulation of JBAS-994 [40], a real JBoss bug, that caused a serious connection leak.
packet loss	
pktloss_both	`iptables` is used to drop 40% of all incoming network packets, as well as 40% of all outgoing network packets.
pktloss_in	`iptables` is used to drop 40% of all incoming network packets.
pktloss_out	`iptables` is used to drop 40% of all outgoing network packets.

Table 5.2: Descriptions of injected faults.

Chapter 6

Analysis Techniques

Analysis algorithms were implemented to demonstrate that CPU performance counter data is suitable for detecting problems in software systems. This thesis does not focus on designing novel or complex data-analysis algorithms, but rather attempts to demonstrate the utility of applying hardware performance counters to problem diagnosis. Simple techniques based on libsvm [13], an off-the-shelf machine learning package implementing Support Vector Machines (SVMs) [20], are applied to data from a novel source. Two approaches are taken: one that performs classification based on performance counter data collected from all machines, and one that performs local classification based on data from a single machine and then merges the data to perform fault localization. For each of method, multiple ways of labeling the training data are considered.

6.1 Support Vector Machines

The SVM is a popular supervised classification method. The underlying principle is that an SVM takes as input a set of n-dimensional data points with binary labels, and outputs a hyperplane. If possible, the hyperplane separates the data points, such that all points on one side of the hyperplane have one label, and all points on the other side have the other label. Moreover, the hyperplane is chosen to maximize the margin, the orthogonal distance between the hyperplane and any point, in the hope that this margin provides a maximal safety cushion for generalization to yet-unseen data. If there is no such hyperplane that separates the data points as described (the data is not linearly separable), a cost is associated with points that end up lying on the wrong side of the hyperplane, and a parameter C controls the trade-off between this cost, and the size of the margin produced. Classification of new data points can be performed by determining which side of the hyperplane the point lies on.

SVMs can be extended in a couple useful ways to improve their usability or predictive performance. One extension is the ability to classify into more than two groups. Multiple SVMs can be used to allow classification into an arbitrary number of sets. A second extension that the author does not explore, is the use of a kernel function to transform non-linear data into a linear feature space.

A concise, but more detailed discussion of SVMs can be found in [20]. Given their wide use

and numerous public implementations, SVMs will be treated as black boxes for the remainder of this thesis.

One issue with SVMs and other supervised-learning algorithms, is that they require labeled training data for each possible classification. This causes difficulty when real faults are involved. Unless faults can accurately be simulated or a previous occurrence of a real fault was used for training, such algorithms may have difficulty identifying real faults. As a result, the analysis techniques discussed may not be ideal for real-world problem diagnosis applications. However, positive results using SVMs for analysis still will demonstrate that the data from CPU performance counters is expressive enough to indicate when a fault is occurring. Thus, success with SVMs may indicate that approaches that do not require training data for faults, (e.g., clustering or peer comparison-based approaches) may also work.

6.2 Data Labeling

Since SVMs require labeled training data, one issue that arises is how to label the data. Many possibilities exist. Three specific schemes were evaluated.

Detection. This labeling scheme simply distinguishes between instances in which a fault is a occurring, and instances where no fault is occurring. All data points collected when the fault injector is active are labeled "fault." Every other data point is labeled "nofault."

Localization. This labeling scheme uses the hostname of the machine on which a fault is occurring. For example, data points collected when the fault injector is active on the machine hosting the database would be labeled "mysql0." If the fault injector was instead active on the application server, the data points would be labeled "ejb0." Data points collected when fault injection is not occurring are labeled "nofault." In a three-tier system, this results in a total of four possible labels.

Limited. This labeling scheme is essentially a hybrid of detection and localization. Data points collected while faults that only significantly affect a single machine are injected are labeled with the faulty machine's hostname, as in localization. However, some faults significantly affect all machines, given the lack of redundancy in the experimental setup. For example, the dblock fault causes drops in many metrics on all machines, since the front-end is waiting on the application server, which is waiting on the database. Such faults are labeled "fault." The dblock, jbas994, and pktloss faults are given this special treatment. As in the other approaches, data points that occur when no fault is being injected are labeled "nofault." In a three-tier system, this results in a total of five possible labels.

To clarify the labeling schemes, table 6.1 shows several faults together with the corresponding labeling for each of the three labeling schemes.

example fault	labeling scheme		
	detection	localization	limited
none	nofault	nofault	nofault
diskhog injected into persistence tier	fault	mysql0	mysql0
cpuhog injected into application tier	fault	ejb0	ejb0
dblock injected into persistence tier	fault	mysql0	fault
jbas994 injected into application tier	fault	ejb0	fault

Table 6.1: Examples of the three labeling schemes.

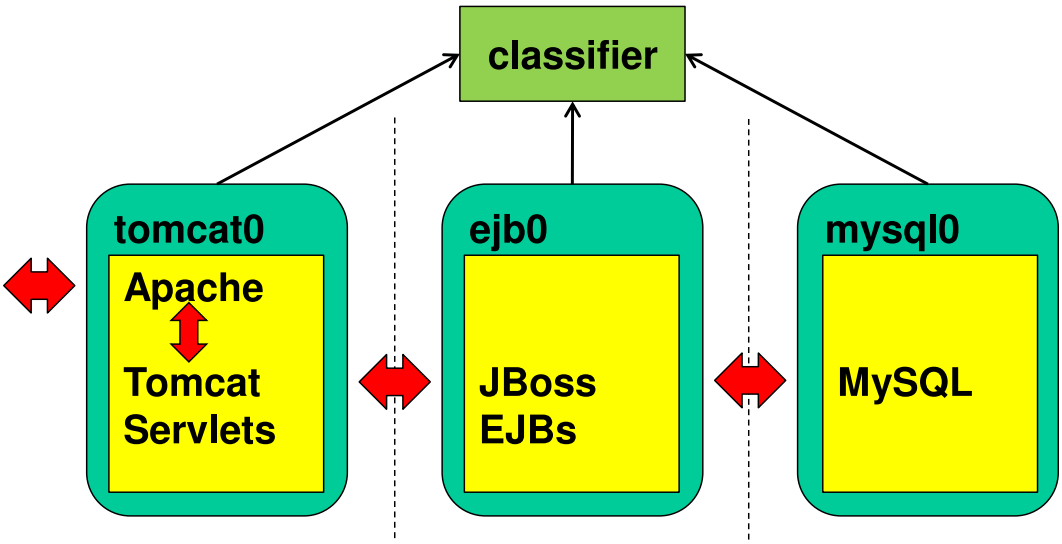

Figure 6.1: Overview of SVM analysis Technique 1.

6.3 Implemented Techniques

Two analysis techniques using SVMs were implemented. The first uses CPU performance counters from every monitored machine as inputs to a single classifier. This classifier's output is the final result of the analysis. The second uses a separate classifier for each machine, and each classifier is only aware of the data from that one machine. The data from each classifier is then analyzed in a second stage before generating a final result.

Technique 1. The first technique is conceptually simple. There is a single analysis agent, aware of all data in the system. Localization can occur if the SVM was trained with data labeled using the localization or limited scheme. Moreover, this technique has the advantage that it can conceivably detect faults that only cause anomalous-looking behavior when metrics from multiple machines are examined simultaneously. For example, consider a cpuhog fault. Looking just at data from the machine on which the cpuhog is running, the cpuhog could potentially be mistaken as increased workload. However, if data from the other machines is examined, their lack of an apparent workload increase would lead to the conclusion that a fault truly is occurring. Unfortunately, there is also a significant disadvantage. The trained model created for problem diagnosis becomes invalid

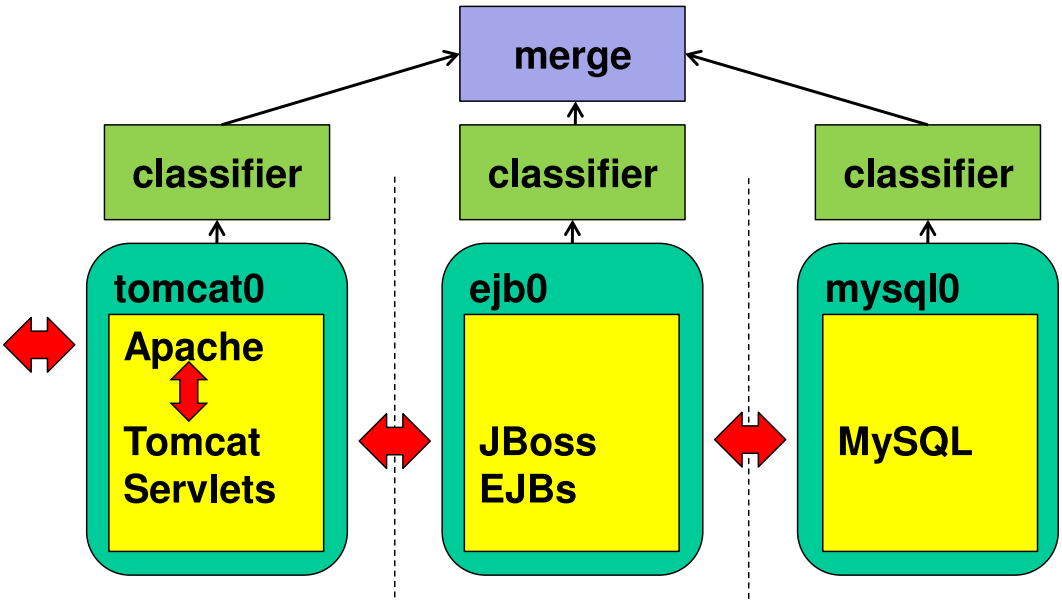

Figure 6.2: Overview of SVM analysis Technique 2.

whenever the number or make-up of machines to monitor changes. Figure 6.1 gives a high-level view of this analysis technique.

Technique 2. The second technique trains a separate SVM classifier for each type of machine (i.e., front-end, application server, or database) based solely on the CPU performance counter data collected from a single machine of the given type. As a result, if a new machine of an existing type is added, and the software system balances load such that its workload is similar to that for existing machines, no new training is necessary. Data labeling for training is also modified slightly; when a fault occurs, only the data points used by the classifiers on machines from which the fault is apparent get labeled to indicate a fault is occurring. For example, if the detection labeling scheme is used, and a diskhog fault is being injected on the database server, the data point for the database server is labeled "fault," but the data points for the other machines are labeled "nofault." On the other hand, if a dblock, jbas994, or pktloss fault is being injected, data points for all machines will be labeled "fault." Localization can then proceed in two ways: by an explicit prediction that a specific machine is faulty (only applicable if the localization or limited labeling scheme is used) or by extrapolating that faults only predicted by the classifier on a single machine are in fact occurring on that specific machine. Figure 6.2 gives a high-level view of this analysis technique.

In all cases, training data was pre-processed to avoid bias due to differing magnitudes for different processor event counts. As suggested by [22], linear scaling was applied to make the range of each feature $[0, 1]$. The scaling transformations used were saved, so the same transformations can be applied to test data analyzed in the future.

6.4 Summary of Chapter

This chapter discussed the analysis techniques implemented. The first is based on a single SVM classifier, and the second uses one SVM classifier for each machine, together with logic for merging the intermediate per-machine results. Since performing a prediction with an SVM is essentially just a polynomial evaluation, both techniques have low run-time computational requirements.

More sophisticated analysis could, and would likely result in better performance. However, as the experimental results will indicate, even these simple analysis approaches achieve good true- and false-positive rates, and can accurately localize certain types of faults.

Chapter 7

Results and Evaluation

This chapter presents results obtained from experiments performed using synthetic fault-injection. First, the metrics used to measure the performance characteristics of analysis algorithms are presented, so the results are well-defined. Next, the tunable parameters of the analysis techniques are presented and discussed. This discussion includes the rationale for choosing specific values, together with data used to make those decisions. Finally, results for combinations of analysis and labeling techniques are presented, followed by a section examining the overhead of performance counter-based data collection.

7.1 Metrics for Evaluation

Evaluation of problem diagnosis performance is always carried out on data collected during the RUBiS client emulator's steady-state phase, to avoid the effects of workload change. To account for hysteresis, a fault is considered to be occurring from 10 seconds before fault-injection, until 10 seconds after fault-injection terminates.

The true-positive rate for fault detection (*tp*). This is the number of experiments that experienced a true-positive (the analysis detected a fault, and a fault was actually occurring) divided by the number of experiments where a fault was injected. A perfect algorithm would have a *tp* value of 1. Note that this does consider not whether or not a fault was correctly localized. For example, if a fault is occurring on the database, and the analysis believes a fault is occurring on the application server, a true-positive is recorded.

The false-positive rate for fault detection (*fp*). This is the number of experiments that experienced a false-positive (the analysis detected a fault, but no fault was actually occurring) divided by the total number of experiments. A perfect algorithm would have a *fp* value of 0. For presentation purposes, most graphs show $1 - fp$, so all plots have the property that higher values are better.

The accuracy rate for fault localization (*accuracy*). The result of a single experiment is said to be accurate if the analysis algorithm indicated the machine in which the fault injected was faulty. The accuracy rate is the number of accurate experiments divided by the number of experiments where a fault was injected. A perfect algorithm would have an *accuracy* value of 1.

The average precision of fault localization (*precision*). The precision of a single experiment is the number of machines correctly identified as faulty by the analysis algorithm divided by all machines identified as faulty by the analysis algorithm. When no machines are determined to be faulty by the analysis algorithm, and the denominator of the ratio is 0, the precision is reported as 1. A perfect algorithm would have a *precision* value of 1.

The *tp* and *accuracy* metrics provide information on how well the analysis techniques cover the detection and localization, respectively, of the faults explored. High values indicate the analysis techniques work well for many types of faults. *fp* and *precision* are complimentary in some sense. *fp* must be low, and *precision* must be high for a problem diagnosis tool to be useful. Otherwise, a large amount of the system administrator's time will be wasted trying to fix problems that don't exist or are on a different machine from the one being examined.

One important thing to remember, is that due to the definitions of *precision* and *accuracy*, an analysis technique that always predicts all machines are faulty will have an *accuracy* value of 1 and a *precision* value of 0.33. When those values appear on result graphs, it is important to remember that performance is poor in reality.

7.2 Parameters for Analysis

The analysis techniques presented, like most analysis techniques, have several parameters that can affect their performance. Some are due to the use of SVMs. These include the choice of kernel, kernel parameters, and cost parameter C. Others involve the choice of CPU performance events counted. Due to the limited number of counter registers available, only 4 events can be counted simultaneously with the current experimental setup. Multiplexing several logical counters onto a single physical counter is a strategy to overcome this limitation, but may start introducing inaccuracies into the collected data. Lastly, a detection window was introduced to Technique 2, to avoid excessive false-positives. The size of this window must be selected to provide a good trade-off between *tp*, *fp*, *accuracy*, *precision*, and latency to diagnosis.

7.2.1 SVM Parameters

Only limited exploration of SVM parameters was carried out. For the most part, the guidelines in [22] were followed. While the RBF kernel is generally recommended, it is noted that when there are large numbers of features (e.g., as with Technique 1), the linear kernel may be a better choice. For consistency and performance reasons, the linear kernel was used with both techniques. A good value for the cost parameter C can be found by running v-fold cross-validation on the training data

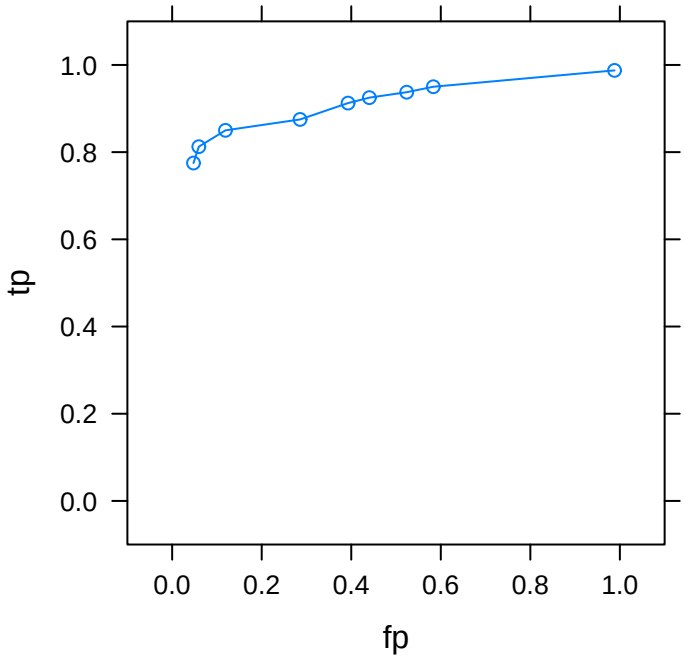

Figure 7.1: ROC curve for Technique 2 fault detection with detection labeling.

with varied values of C. A few rough tests indicated that higher C values typically result in higher cross validation accuracy. However, the default C value of 1 was only slightly worse (0.3-2.0% less accurate for 5-fold cross-validation), and caused training to run much more quickly. As a result, a C value of 1 was used in all experiments.

7.2.2 Window Size

Initial results using analysis Technique 2 displayed very high false-positive rates. Examination of these results indicated that the high false-positive rates were due to a small number of isolated fault predictions when no fault was occurring. Given that most faults are expected to have longer-term effects, it makes sense to ignored isolated predictions. A windowing approach achieves this goal.

The windowing approach works as follows: Only report fault predictions when the prediction lies in some N-sample window containing at least two fault predictions. This is successful at reducing the false-positive rate, but may introduce up to N seconds of additional latency in reporting a fault. To observe empirically the affects of different values of N on the evaluation metrics, N is varied to construct Receiver Operating Characteristic (ROC) curves [38].

Data from experiments using the detection labeling method for Technique 2 was used to generate the ROC curves in figures 7.1 and 7.2. These curves were generated from experiments where the first 8 counter events selected by PCA-based metric selection with standard training were collected. The first curve, which provides a visualization of the analysis technique's fault detection

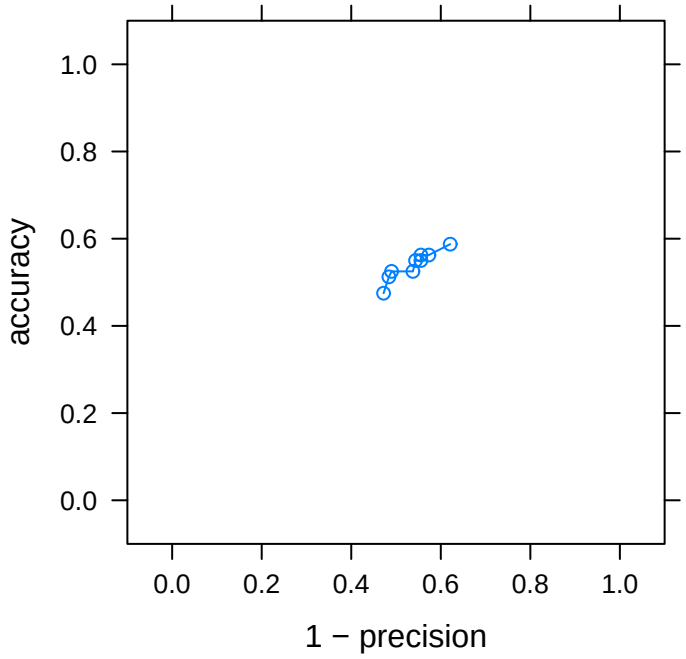

Figure 7.2: ROC curve for Technique 2 fault localization with detection labeling.

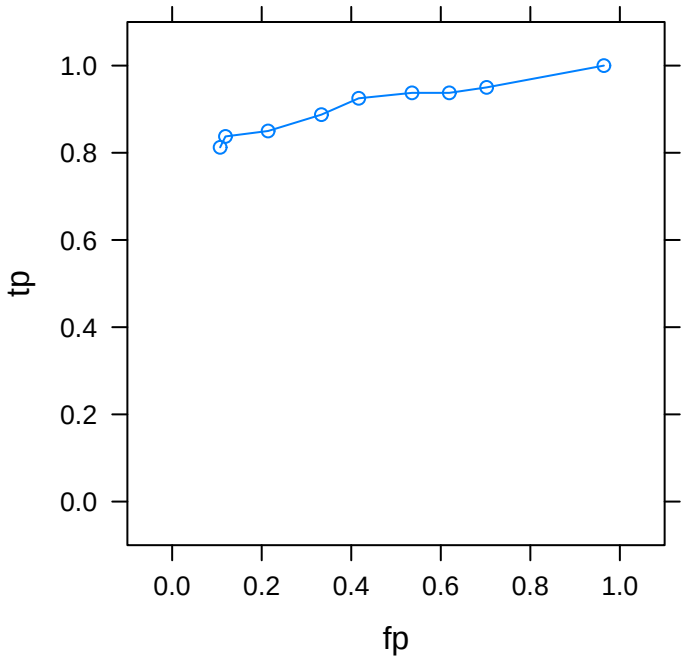

Figure 7.3: ROC curve for Technique 2 fault detection with limited labeling.

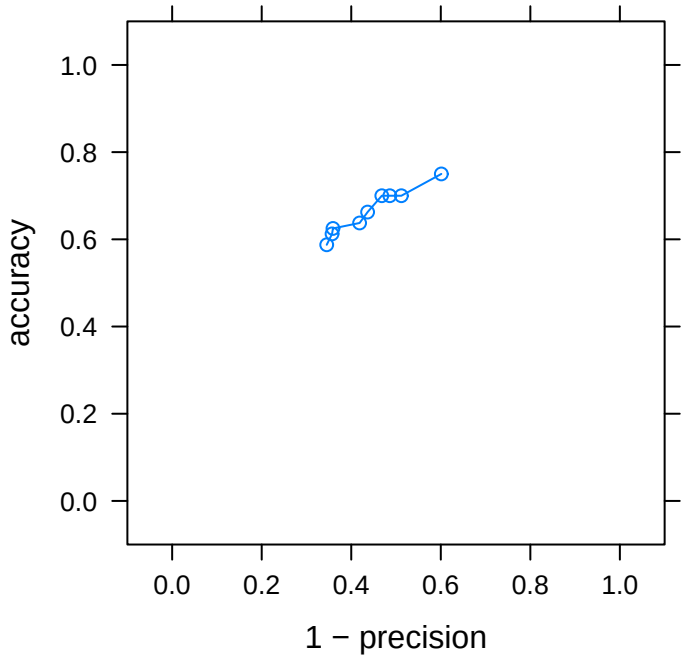

Figure 7.4: ROC curve for Technique 2 fault localization with limited labeling.

performance, has a long, flat, high curve. This is a good characteristic, and means it is possible to obtain a high true-positive rate, with a low false-positive rate. The ideal point on an ROC curve is the upper left-hand corner of the plot, where the true-positive rate is 1 and the false-positive rate is 0. Data points near that location correspond to small window sizes, such as 10. The second curve illustrates fault-localization performance. It is not nearly as good; all of the points are near the center of the grid, meaning all window sizes experiences moderate accuracy and precision.

Equivalent data from experiments using the limited labeling method are shown in figures 7.3 and 7.4. These curves were generated from experiments where the first 4 counter events selected by PCA-based metric selection under either training data set were collected. The curves are very similar to those for the detection case, and similar conclusions can be made.

7.2.3 Dataset Selection

Section 5.3 described a technique based on PCA for determining which CPU performance counter were most likely to convey useful information. Two lists of events, ordered based on the expressive power of the events were the result. One, standard, was constructed using data from standard experiments, while the other, w/ extra faultfree, used additional data from an extended faultfree run of the system under an emulated workload. Results are presented using the top-4, 8, 12, and 16 events from each of the two lists.

Graphs useful in evaluating the choice of dataset size for analysis Technique 1 are shown in

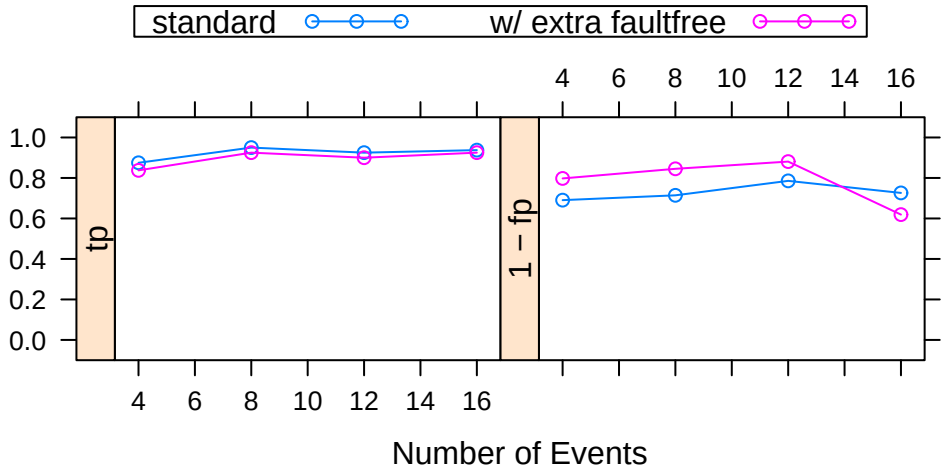

Figure 7.5: Dataset comparison graphs for SVM Technique 1 with detection labeling.

figures 7.5, 7.6, and 7.7. In all graphs, the results are based on data from all experiments performed that were not used for SVM training. Specifically, the graphs show results for four runs of each fault type, and four fault-free runs.

Figure 7.5 shows that, for Technique 1 with detection labeling, *tp* values range from about 0.8 to 0.9. Predictions based on the counter event sets constructed from the standard training set always fare slightly better than those constructed from the w/ extra faultfree training set, and subsets with 8 events work the best. For *fp*, additional counters help for up to 12 event types, but then performance decreases with 16. The event sets based on w/ extra faultfree training typically do better than those based on standard. Based on the graphs, it appears that the 8 events constructed from the w/ extra faultfree training data perform the best overall.

Figure 7.6 shows performance information for Technique 1 with localization labeling. In this case, an increased number of events definitely helps *tp* and *accuracy*. Only at that point does performance become at all reasonable. However, at that same point *precision* takes a dramatic decrease. Based on the graphs, it is probably most reasonable to use 12 counter events, based on the standard PCA-selected metrics.

Figure 7.7 shows performance for different datasets when Technique 1 is used with limited labeling. This labeling technique was motivated by the poor *tp* values seen using the localization labeling technique. As expected, limited labeling greatly improves *tp*, but at the cost of *accuracy*; *accuracy* values never make it above 0.2. Performance does not improve significantly after 8 counter events, so 8 events based on the standard PCA-selected metrics work best for this technique and labeling combination.

Graphs useful in evaluating the choice of dataset size for analysis Technique 2 are shown in figure 7.8 and 7.9. As with the Technique 1 graphs, data was based on four runs of each fault type, and four fault free runs, none of which were used for SVM training. Windowing was configured to require 2 fault predictions within 10 seconds in order for a fault to be reported. As shown in section7.2.2, this size greatly decreases the false-positive rate, without also causing a large decrease

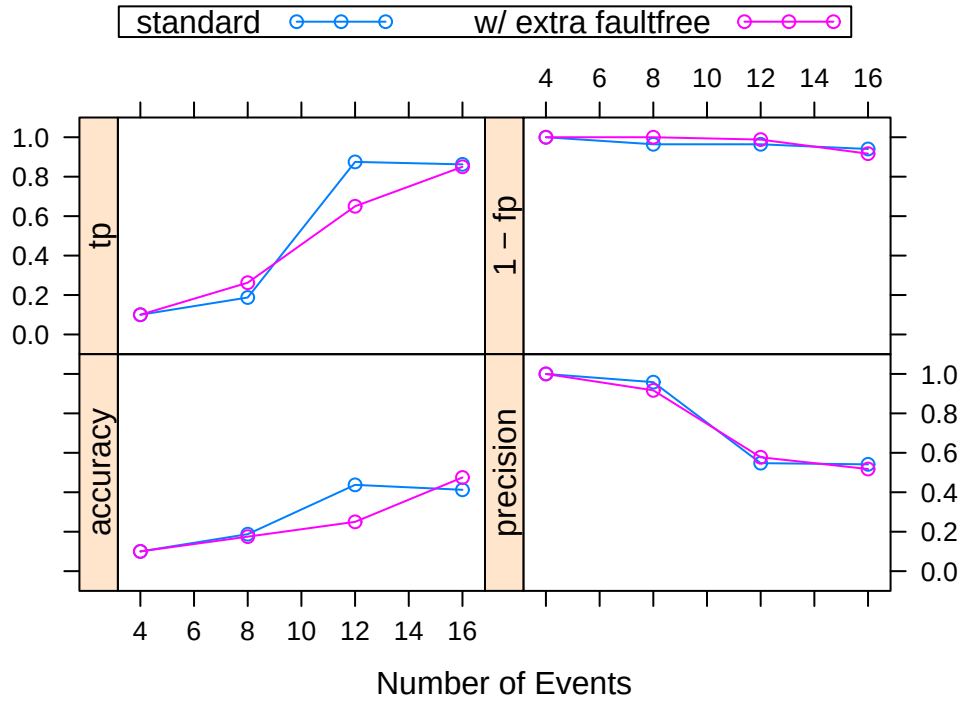

Figure 7.6: Dataset comparison graphs for SVM Technique 1 with localization labeling.

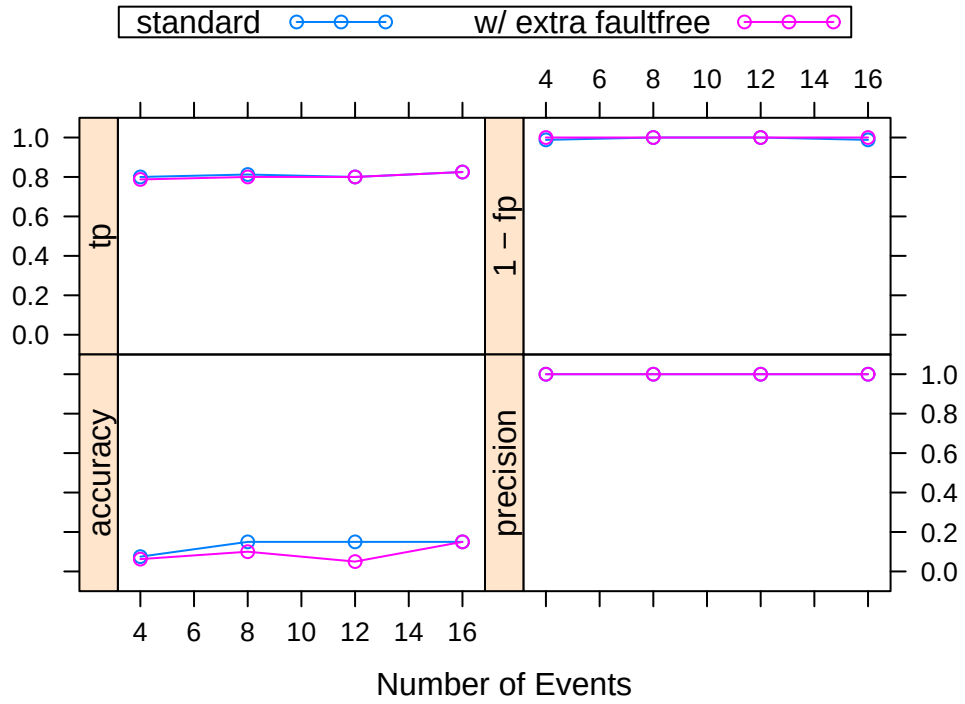

Figure 7.7: Dataset comparison graphs for SVM Technique 1 with limited labeling.

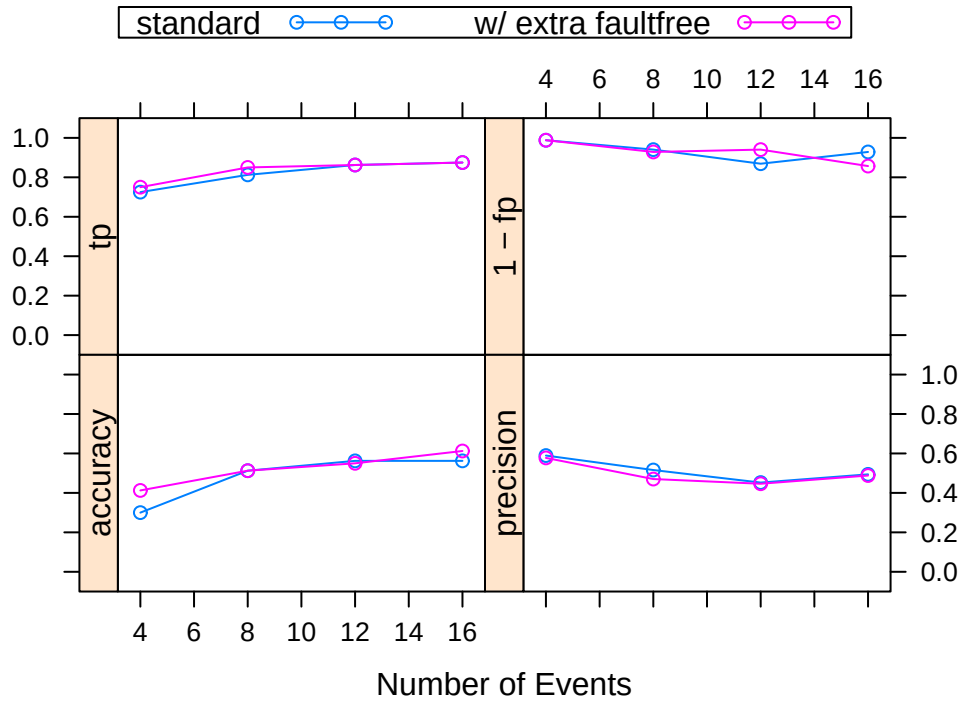

Figure 7.8: Dataset comparison graphs for SVM Technique 2 with detection labeling.

in the true-positive rate.

Figure 7.8 shows performance for different datasets when Technique 2 is used with detection labeling. In general, more counters yield better *tp* and *accuracy* values, at the expense of *fp* and *precision*. Fault detection works well, with *tp* values above 0.8 when 8 or more events are counted, with *fp* values below 0.2. Localization, on the other hand does not perform as well. Both *accuracy* and *precision* values hover around 0.5. Based on the graphs, it appears either of the 8 counter event sets would be good a choice.

Figure 7.9 shows performance for different datasets when Technique 2 is used with limited labeling. All sets of events have similar performance characteristics. However, it appears that on average using the 4 counter events selected by the PCA from the standard PCA fair slightly better. In reality the difference between the two 4 counter event sets is entirely due to experimental variance; the PCA-based selection algorithm chooses the same top-4 events when either the standard or the w/ extra faultfree training data is used. *accuracy* and *precision* are slightly better than with detection labeling, but still leave something to be desired, having values around 0.6.

7.3 Overall Results

This section presents overall results based on system-wide CPU performance counter instrumentation. All data was collected using the settings and parameters given in section 5.4. Data was

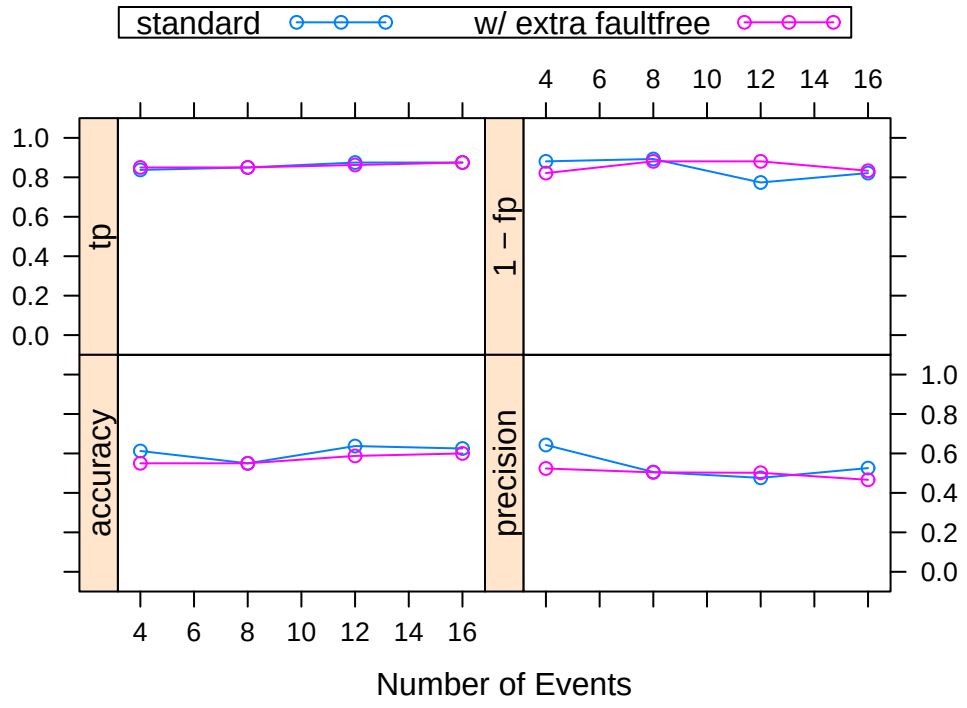

Figure 7.9: Dataset comparison graphs for SVM Technique 2 with limited labeling.

collected from four experimental runs under each fault, and four fault-free runs. This data is separate from the data collected and used to train the SVMs for analysis.

7.3.1 Technique 1 with Detection Labeling

Based on the evaluation in section 7.2.3, the combination of Technique 1 and detection labeling works best with the top-8 metrics selected by the PCA when the w/ extra faultfree training is used. Results, itemized by injected fault type are displayed in figure 7.10. The graph shows that all faults other than memoryhog can be consistently detected. This fact is not necessarily surprising; the momentary spike a memoryhog fault produces in measurements from CPU performance counters seems like it should harder to detect than the sustained behavior changes the other faults cause. Overall, the combination of this analysis technique and detection labeling resulted in true-positive rate of 0.925, and a false-positive rate of 0.155.

7.3.2 Technique 1 with Localization Labeling

Using localization labeling, Technique 1 works best with the top-12 counter events selected via PCA-based selection using the standard training data. Results are displayed in figure 7.11. For fault detection, memoryhog faults again cause the greatest difficulty. Localization works best for the cpuhog faults. Poor performance seems largely due to the fact that the dblock, jbas994,

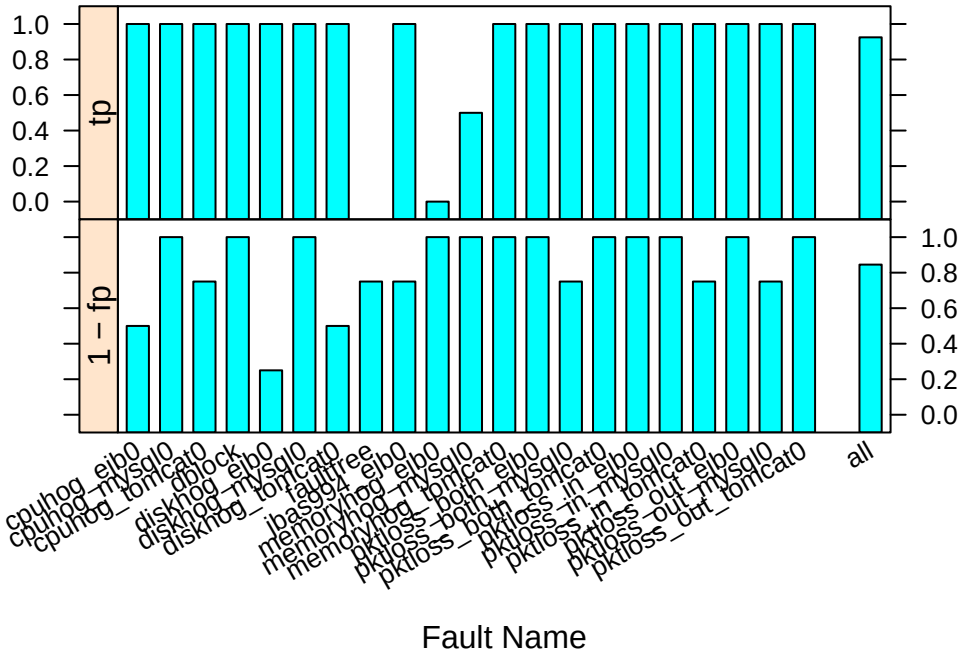

Figure 7.10: Results for Technique 1 with detection labeling.

and pktloss faults all cause similar manifestations to each machine hosting RUBiS. Overall, this combination of analysis and labeling techniques resulted in a true-positive rate of 0.875, a false-positive rate of 0.0357, an accuracy of 0.438, and an average precision of 0.548.

7.3.3 Technique 1 with Limited Labeling

Limited labeling was introduced to try and solve some of the issues with localization labeling. The top-8 events selected by PCA-based event selection on the standard data set were measured. Result graphs showing the detection and localization statistics for each fault are given in figure 7.12. The bars labeled "all *" are the evaluation metrics as calculated when only considering cpuhog, diskhog, and memoryhog faults. Given the labeling used to train the SVM, those are the only faults where the analysis is expected to be able to perform localization. The results demonstrate that Technique 1 with limited labeling does not do a good job localizing faults, only obtaining an accuracy of 0.1. On the other hand, detection works well, obtaining a true-positive rate of 0.8 and no false-positives.

7.3.4 Technique 2 with Detection Labeling

Figure 7.13 shows the performance of analysis Technique 2 with detection labeling. As discussed in section 7.2.2, a fault is only reported if two faults are predicted by the underlying SVMs within

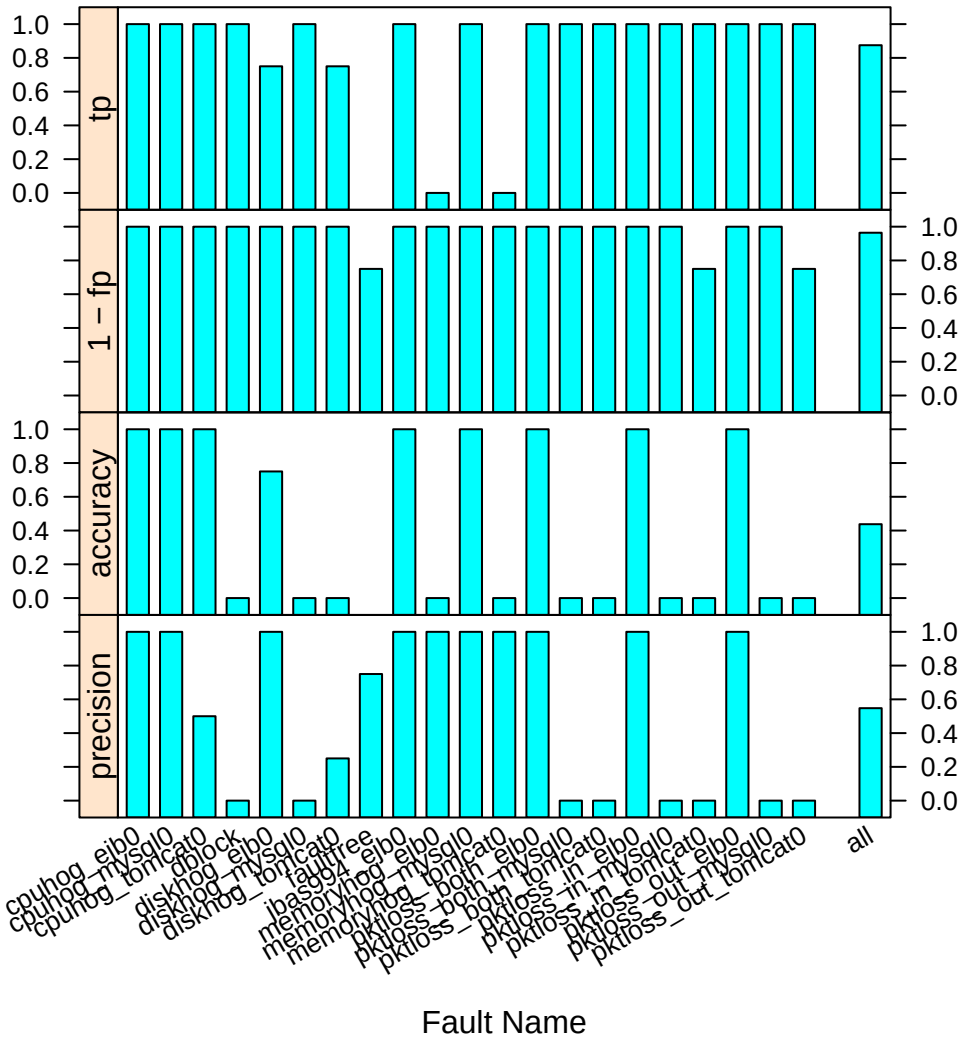

Figure 7.11: Results for Technique 1 with localization labeling.

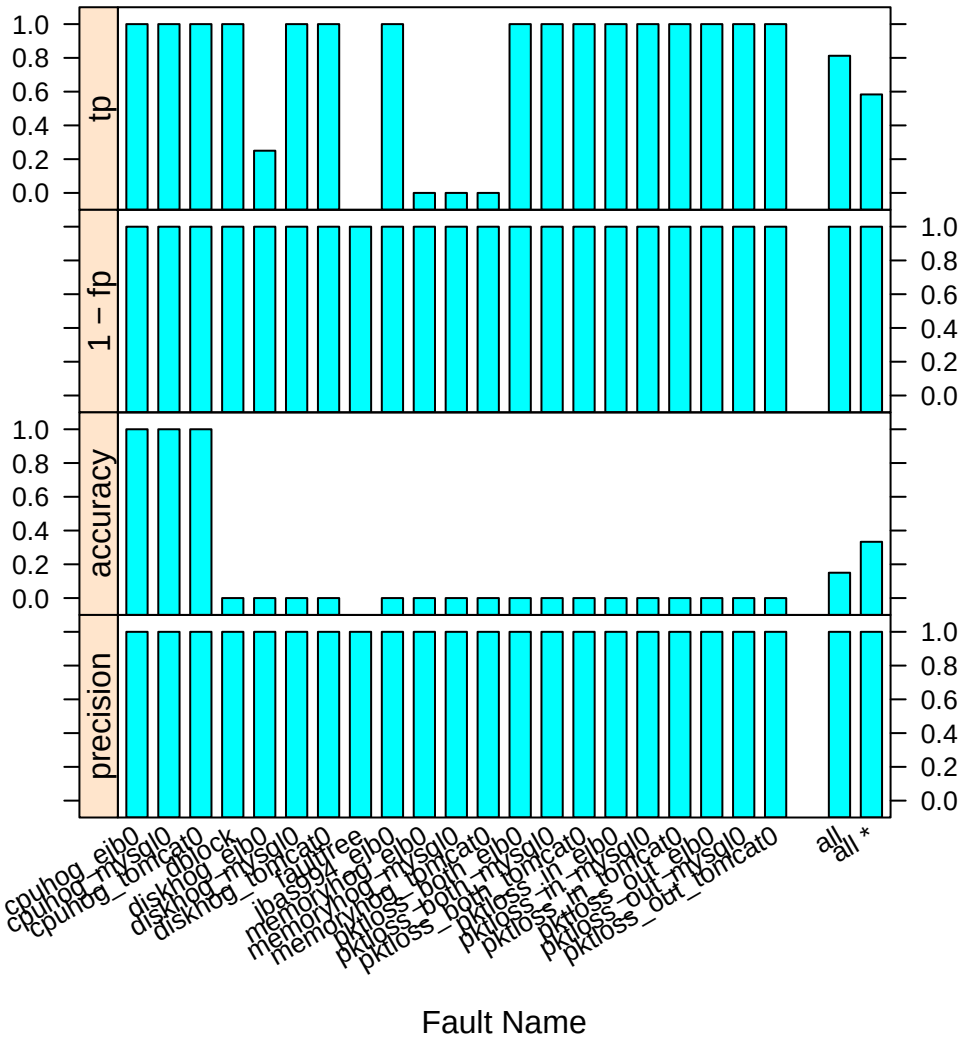

Figure 7.12: Results for Technique 1 with limited labeling.

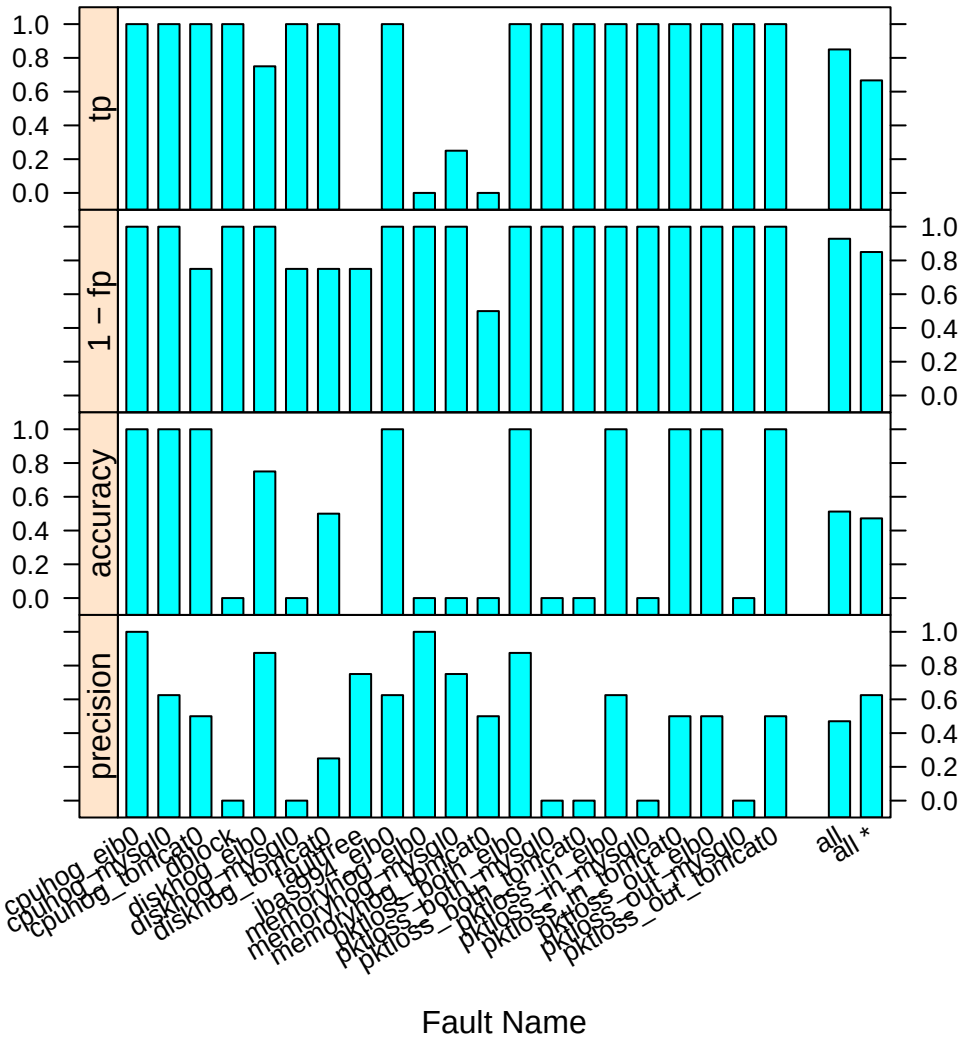

Figure 7.13: Results for Technique 2 with detection labeling.

some 10 second window. For this data, the top-8 counter events selected via PCA-based selection on the w/ extra faultfree dataset were used. All non-memoryhog faults are consistently detected. Localization is better than that experienced using Technique 1, but still not good enough to be particularly useful. Overall, this approach yields a true-positive rate of 0.850, a false-positive rate of 0.071, an accuracy of 0.513, and an average precision of 0.470.

7.3.5 Technique 2 with Limited Labeling

Figure 7.14 shows the performance of analysis Technique 2 with limited labeling. As before, the window size for fault reporting was set to 10. Plots show the top-4 metrics chosen by PCA-based selection on the standard training data. Results are similar to those using detection labeling from the previous section, but with slightly worse detection performance and slightly better localization performance. Overall, this approach yields a true-positive rate of 0.838, a false-positive rate of 0.119, an accuracy of 0.613, and an average precision of 0.643.

7.4 Per-Task Operation

As mentioned, Perfmon2 supports a per-task mode of operation where CPU performance counters are virtualized per-thread. It seems like this mode of operation might work better than the system-wide mode that has been explored so far, as the per-task CPU performance counters will not pick up noise introduced by other, unrelated programs. Initial exploration, however, runs contrary to these expectations. Analysis results indicates that the system-wide mode generally performs better.

A seperate metric selection/ranking step was not performed on per-task data. As a consequence, fault detection and localization performance in this initial exploration of per-task instrumentation is probably worse than would be expected if the datasets were more carefully tuned.

Figure 7.15 shows the performance of analysis Technique 2 with limited labeling, when per-task data is used. As with the system-wide data, the window size was set to 10. The results displayed are based on the 8-event set generated by running the PCA-ranking heuristic on w/ extra faultfree. Fault detection is reasonable, but localization is very poor.

No other analysis technique and labeling combination obtained good results on the per-task data. Some had *tp* values of 1 with *fp* values of 1. Others had *tp* values of 0 with *fp* values of 0.

7.5 Overheads

This thesis hypothesizes not only that data from CPU performance counters is suitable for use in problem diagnosis, but that the data also can be collected with low runtime overhead. This section evaluates the overheads introduced experimentally, by observing how much throughput (as measured by the RUBiS client emulator) decreases when instrumentation is enabled.

The average overheads when a single set of counters is used (i.e., there is no multiplexing), of instrumentation are given in table 7.1. This data was collected from experiments where no

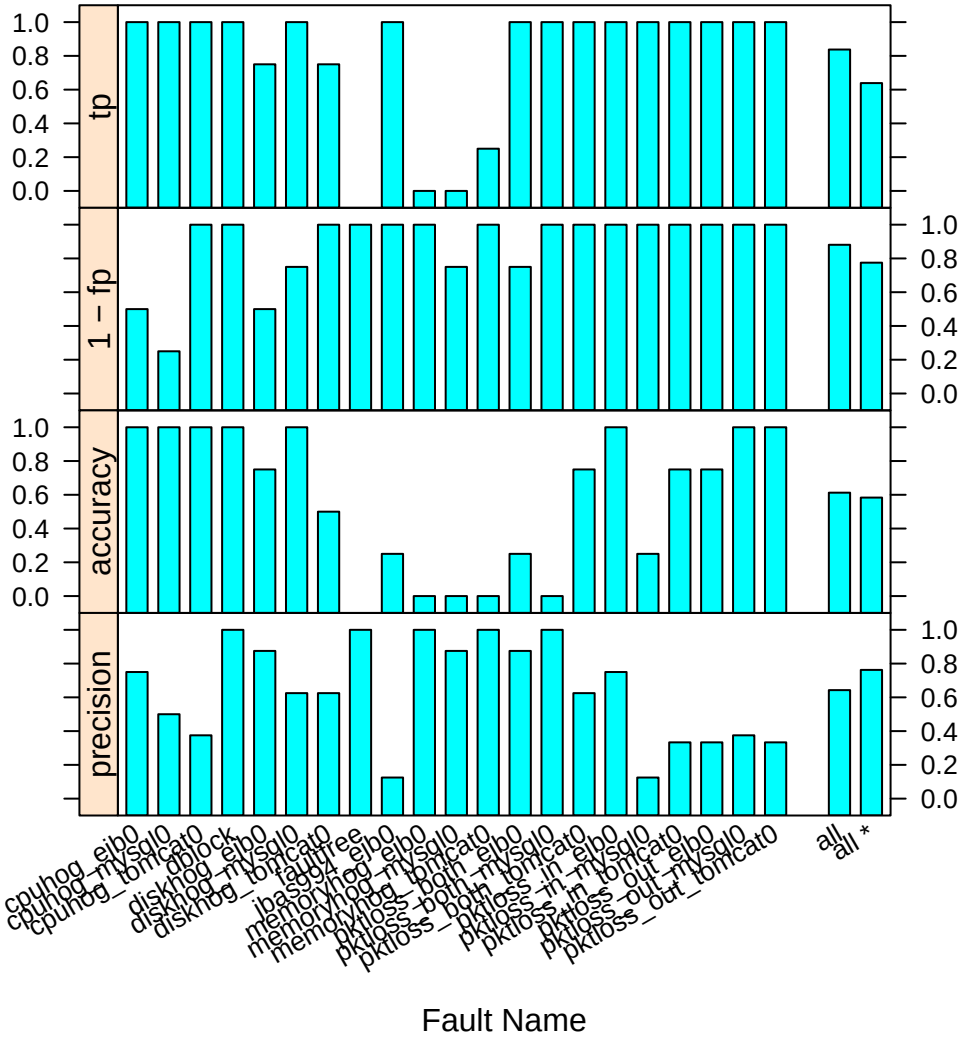

Figure 7.14: Results for Technique 2 with limited labeling.

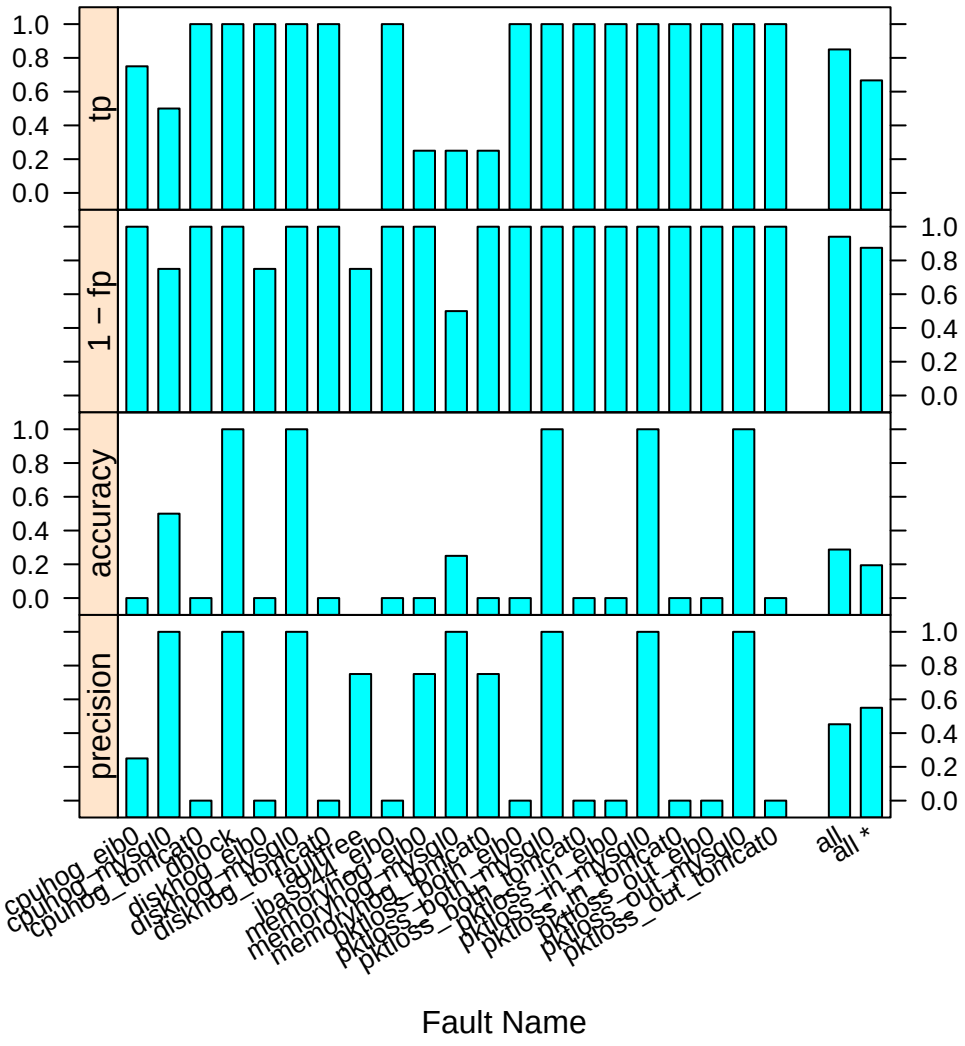

Figure 7.15: Results for Technique 2 with limited labeling, when applied to per-task data.

	avg	stdev	min	max
baseline	161.33	0.4924	161	162
instrumented	160.67	0.5164	160	161
overhead	0.413%		0.000%	1.235%

Table 7.1: Overheads introduced by instrumentation: no multiplexing.

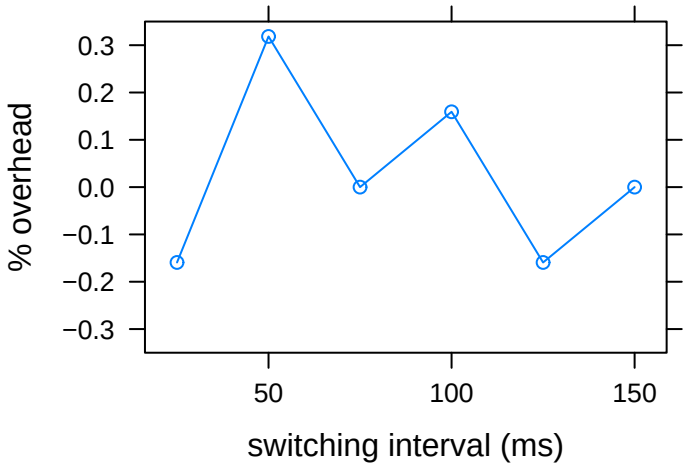

Figure 7.16: Overheads introduced by instrumentation: with multiplexing.

fault was injected, 1000 clients were emulated, and the steady-state period lasted 15 minutes. 18 experiments were performed in total: 12 without instrumentation, in order to obtain a baseline for expected throughput, and 6 with instrumentation. The data demonstrates that CPU performance counter-based instrumentation truly is low-overhead; on average, throughput decreases by less than 0.5%.

To implement multiplexing functionality, Perfmon2 has a timer generate interrupts at a regular period to trigger event set switching. Thus, when multiplexing is used, one would expect higher overheads, since event set switching uses CPU time and potentially hurts cache locality. Moreover, shorter switching intervals should result in higher overheads, since more event set switching must occur. To evaluate the effects of multiplexing, experiments were run varying the switching interval from 25 to 150 milliseconds in 25 millisecond steps. Four fault-free experiments were performed using each switching interval, in addition to four experiments performed without any instrumentation. All experiments used 1000 emulated clients and a steady-state period lasting 6 minutes. The average overheads experienced are graphed in figure 7.16. Varying the switching interval for multiplexing does not appear to affect overheads. Moreover, in all cases, the overheads produced are very small, less than 0.4%. This seems to indicate the overheads are similar in magnitude to variance in the experimental results.

7.6 Summary of Chapter

This chapter first presented four metrics for measuring the performance of problem diagnosis tools: *tp*, *fp*, *accuracy*, and *precision*. These metrics provide a solid foundation on which experimental data can be discussed. Second, tunable parameters of the analysis techniques were explored. Rationale, based on empirical data when possible, is given for the parameter values ultimately chosen. Third, performance was evaluated for combinations of analysis techniques and labeling methods. Fault detection performance is usually rather good, but localization only succeeds for a limited subset of the faults explored. Finally, the instrumentation overheads were examined. Further reflection will consider the strengths and weaknesses of this CPU performance-counter based problem diagnosis approach, and consider potential improvements and extensions.

Chapter 8

Conclusion

This thesis presented a technique for performing problem diagnosis based on black-box data collected from CPU performance counters. Methods of data collection and analysis were designed, and then evaluated empirically via a series of experiments with synthetic fault-injection. Through these experiments, the author was able to determine good values for parameters controlling various aspects of the analysis, and obtain concrete results on the effectiveness of a CPU performance counter-based approach to problem diagnosis.

8.1 Lessons Learned

The main lesson learned from this research is that CPU performance counters can provide information that is very suitable for detecting faults. One of the analysis techniques was able to obtain a true-positive rate of 0.875 and a false-positive rate of only 0.0357. However, at least with the algorithms explored, localization does not work particularly well. The analysis technique that worked best for localization only obtained an accuracy of 0.613 and a precision of 0.643.

Of course, it is possible that more sophisticated analysis techniques can do a better job localizing faults. Given the low instrumentation overheads, which were less than 1% on average, additional exploration seems worthwhile.

It is also constructive to consider the types of faults where localization was most effective. These faults involved external code running on one of RUBiS' tiers, which was directly measurable through the CPU performance counters. The faults that could not be effectively localized typically were not directly measurable by CPU performance counters: instead, the fault caused a bottleneck in RUBiS' request processing, which caused indirect effects to the CPU performance counters on all of the tiers. Since these indirect effects occurred on all the tiers, localization was not possible. As a result, it seems like localization is only effective when a fault's direct effects are greater than its indirect effects seen on all machines.

8.2 Future Work

This research has focused on one use of CPU performance counters for problem diagnosis. Additionally, only simple analysis techniques were explored, and problem diagnosis was only performed offline, and in the context of a single system-under-diagnosis, RUBiS.

As a part of this work, the original intent was to implement an online version of CPU performance counter-based diagnosis in the context of ASDF [10]. In fact, a functioning data-collection module based on `pfmon` was written. However, no analysis techniques were ultimately implemented as ASDF modules. It would be instructive to actually implement an analysis technique in ASDF, to empirically measure the overheads of online analysis.

The current method for collecting performance counter data is black-box. As a result, it is general, but lacks some context information that gray- and white-box approaches are able to obtain. An interesting thing to note, is that data collection from CPU performance counters need not be black-box. It would be very interesting to instrument an RPC or middleware library or application function, to collect the number of performance counter events that occur during the call. Then, a profiling approach similar to the one specified in [28] could be used to determine if a given call may be anomalous.

Another area of interesting future work would be to explore additional algorithms for fault detection and localization. The techniques presented serve as a good proof of concept, but more advanced techniques would likely be able to achieve better results.

Three-tier e-commerce applications form an important class of deployed, well-used software systems. However, the exploration of different types of systems might demonstrate the increased applicability of this hardware performance counter-based approach, or introduce additional difficulties. It would also be instructive to try different system configurations. Presumably, peer-comparison analysis approaches would become applicable if one or more tiers were replicated or duplicated for increased capacity.

Finally, more faults could be explored. The set of faults used in those work mainly consist of external problems that were induced outside of the target system-under-diagnosis. In some ways, this is very well-suited to the global event-collection data mode discussed. The effects of many of the faults studied could be observed directly. However, there are many more types of faults. For example, some faults may cause specific requests to hang, or fail and complete prematurely. Some may not be easily diagnosable with the methods presented, but getting a better idea of the limitations of an approach is helpful in understanding what combinations of problem diagnosis techniques would provide sufficient coverage.

CPU performance counter-based diagnosis certainly is interesting and has value, as it presents a novel instrumentation source for problem diagnosis. In many ways, this thesis just explores a small subset of the potential uses of this instrumentation source. Future work is bound to reach many more exciting conclusions.

References

[1] Advanced Micro Devices, Inc. *AMD64 Architecture Programming Manual*, volume 2: System Programming. July 2007. 4.1

[2] Advanced Micro Devices, Inc. *BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors*. July 2007. 4.1, 5.3

[3] M. K. Agarwal, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Mining activity data for dynamic dependency discovery in e-business systems. *IEEE Transactions on Network and Service Management*, 1(2):49–58, Dec. 2004. 2.2

[4] M. K. Agarwal, M. Gupta, V. Mann, N. Sachindran, N. Anerousis, and L. B. Mummert. Problem determination in enterprise middleware systems using change point correlation of time series data. In *NOMS '06: 10th Annual IEEE/IFIP Network Operations and Management Symposium*, pages 471–482, Vancouver, BC, Canada, 2006. 1, 2.2

[5] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. 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):357–390, 1997. 1, 2.1, 3.1

[6] The Apache HTTP Server Project, Aug. 2009. `http://httpd.apache.org/`. 5.2

[7] Apache Tomcat, Aug. 2009. `http://tomcat.apache.org`. 5.2

[8] K. Appleby, J. Faik, G. Kar, A. Sailer, M. Agarwal, and A. Neogi. Threshold management for problem determination in transaction based e-commerce systems. In *IM '05: 9th IFIP/IEEE International symposium on Integrated Network Management*, pages 733–746, Nice, France, May 2005. 1, 2.2

[9] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing*, pages 101–110, Cambridge, MA, USA, 2005. 1, 2.1, 4.2

[10] K. Bare, M. P. Kasick, S. Kavulya, E. Marinelli, X. Pan, J. Tan, and R. Gandhi. Asdf: Automated, online fingerpointing for hadoop. Technical Report CMU-PDL-08-104, Parallel Data Lab, Carnegie Mellon University, Pittsburgh, PA, USA, May 2008. 8.2

[11] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, 9(2):175–190, 2006. 1, 2.2

[12] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of ejb applications. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 246–261, Seattle, Washington, USA, 2002. 1, 3.3, 5.1, 5.4

[13] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`. 6

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *DSN '02: Proceedings of the International conference on Dependable Systems and Networks*, pages 595–604, Bethesda, MD, USA, 2002. 1, 2.2

[15] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th symposium on Operating Systems Design and Implementation*, pages 231–244, San Francisco, CA, USA, 2004. 1, 2.2

[16] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *SOSP '05: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 105–118, Brighton, United Kingdom, 2005. 1, 2.2

[17] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 289, Washington, DC, USA, 2003. 2.1

[18] L. Eeckhout, A. Georges, and K. De Bosschere. How java programs interact with virtual machines at the microarchitectural level. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 169–186, Anaheim, California, USA, 2003. 2.1

[19] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: a pervasive network tracing framework. In *NSDI '07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, pages 271–284, Cambridge, MA, USA, Apr. 2007. 1

[20] D. Fradkin and I. Muchnik. Support vector machines for classification. In J. Abello and G. Cormode, editors, *Discrete Methods in Epidemiology*, volume 70 of *DIMACS Series in Discrete Mathematics*, pages 13–20. AMS, Providence, RI, USA, 2006. 6, 6.1

[21] M. Hauswirth, A. Diwan, P. F. Sweeney, and M. Hind. Vertical profiling: understanding the behavior of object-oriented applications. In *OOPSLA '04: Proceedings of the 19th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, Canada, 2004. ACM. 2.1

[22] C.-W. Hsu, C.-C. Chang, and C.-J. Lin. *A Practical Guide to Support Vector Classification*. Taipei, Taiwan, May 2009. 6.3, 7.2.1

[23] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3B: System Programming Guide, Part 2. Nov. 2008. 4.1

[24] Intel VTune Performance Analyzer, 2009. `http://software.intel.com/en-us/intel-vtune/`. 2.1

[25] JBoss Application Server, Aug. 2009. `http://jboss.org/jbossas/`. 5.2

[26] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward. Automatic fault detection and diagnosis in complex software systems by information-theoretic monitoring. In *DSN DCCS '09: 39th IEEE/IFIP International Conference on Dependenable Systems and Networks Dependable Computing and Communications Symposium*, Estoril, Portugal, June 2009. 1, 2.2

[27] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward. System monitoring with metric-correlation models: Problems and solutions. In *ICAC '09: 6th IEEE International Conference on Autonomic Computing and Communications*, pages 13–22, Barcelona, Spain, June 2009. 1, 2.2

[28] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating system profiling via latency analysis. In *OSDI '06: Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 89–102, Seattle, WA, USA, 2006. 1, 8.2

[29] G. Khanna, I. Laguna, F. A. Arshad, and S. Bagchi. Distributed diagnosis of failures in a three tier e-commerce system. In *SRDS '07: Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 185–198, Beijing, China, 2007. 1, 2.2

[30] MySQL: the world's most popular open source database, Aug. 2009. `http://www.mysql.com/`. 5.2

[31] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USITS '03: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, USA, 2003. 1

[32] OProfile, Aug. 2009. `http://oprofile.sourceforge.net/`. 1, 2.1

[33] Linux performance counters driver, Aug. 2009. `http://perfctr.sourceforge.net/`. 2.1

[34] Perfmon2, May 2009. `http://perfmon2.sourceforge.net/`. 2.1, 4.2

[35] S. Pertet, R. Gandhi, and P. Narasimhan. Fingerpointing correlated failures in replicated systems. In *2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning (SysML)*, Cambridge, MA, USA, Apr. 2007. 1, 5.2

[36] S. Pertet and P. Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, USA, Dec. 2005. 1

[37] PMAPI, Aug. 2009. http://www.alphaworks.ibm.com/tech/pmapi/. 2.1

[38] Receiver operating characteristic. *Wikipedia, The Free Encyclopedia*, Aug. 2009. http://en.wikipedia.org/w/index.php?title=Receiver_operating_characteristic\&oldid=310144687. 7.2.2

[39] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI '06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 115–128, San Jose, CA, USA, 2006. 1

[40] Serious connection leak in connection pool—JBoss JIRA, Sept. 2009. https://jira.jboss.org/jira/browse/JBAS-994. 5.5

[41] S. Shankland. Paypal suffers from e-commerce outage. *CNet News*, Aug. 3, 2009. http://news.cnet.com/8301-1023_3-10302072-93.html. 1

[42] J. Shlens. *A Tutorial on Principal Component Analysis*, Apr. 2009. 5.3, 5.3

[43] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of java applications. In *VM '04: Proceedings of the 3rd USENIX Virtual Machine Research and Technology Symposium*, pages 57–72, San Jose, CA, USA, 2004. 1, 2.1

[44] Transaction Processing Performance Council. *TPC Benchmark W (Web Commerce)*, Feb. 2002. http://www.tpc.org/tpcw/. 5.4

[45] A. W. Williams, S. M. Pertet, and P. Narasimhan. Tiresias: Black-box failure prediction in distributed systems. In *IPDPS '07: 21st IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, Long Beach, CA, USA, Mar. 2007. 1