PROBABILISTIC ERROR DETECTION AND DIAGNOSIS IN LARGE-SCALE DISTRIBUTED APPLICATIONS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Ignacio Laguna Peralta

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2012

Purdue University

West Lafayette, Indiana

To my parents, Marisela and Ignacio, who sacrificed themselves to provide the best education for me, and thought me the value of hard work and persistence.
To my wife, Sonia Ivonne, whose love has been unconditional throughout this journey; this work has only been possible because of her support.

ACKNOWLEDGMENTS

Many people contributed to this dissertation in countless ways, and I am grateful to all of them.

First, I would like to thank my advisor, Prof. Saurabh Bagchi, for his patience and encouragement during my graduate studies. He gave me constructive criticisms as well as enough freedom to pursue my own ideas. He always motivated me and trained me to improve the quality of my research. I am grateful for all the doors and windows of opportunity that he opened to me along this journey.

I would like to thank my committee members, Martin Schulz, Prof. Samuel Midkiff and Prof. Charlie Hu, for their encouragement, insightful comments, and hard questions.

I also want to acknowledge the great advice that I received from my mentors at the Lawrence Livermore National Laboratory (LLNL): Bronis de Supinski, Martin Schulz, Todd Gamblin, Greg Bronevetsky and Dong H. Ahn. They introduced me to the world of high-performance computing, taught me how to frame research ideas, helped me in writing this dissertation, and provided professional advice to grow as an independent researcher. I was privileged to work with so many talented individuals at LLNL.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CC	Correlation coefficient
CCV	Correlation coefficient vector
HPC	High-performance computing
LP	Least progress
LPT	Least-progress task
MM	Markov model
MPI	Message passing interface
NN	Nearest neighbor
SMM	Semi-Markov model
PDG	Progress-dependence graph

ABSTRACT

Laguna, Ignacio Ph.D., Purdue University, December 2012. Probabilistic Error Detection and Diagnosis in Large-Scale Distributed Applications. Major Professor: Saurabh Bagchi.

As today's distributed applications increase in complexity, it becomes increasingly difficult to detect errors and performance anomalies in these applications. In addition, some faults only manifest when the application is deployed at large scale. Most of the existing debugging tools scale poorly and do not automate the process of finding the origin of failures. Although it is desirable to automatically predict impending failures, most of the existing error detection approaches do not predict failures.

This dissertation proposes scalable techniques for error detection, problem localization, and failure prediction for distributed applications. First, an error detection and diagnosis technique for scientific applications is presented. The technique summarizes historic control-flow and timing information of MPI tasks using semi-Markov models. When a failure occurs, it leverages the models to determine the parallel task(s) and code region(s) where a fault is first manifested. The isolation of a difficult-to-catch bug in a large scale molecular dynamics simulation code and fault injections demonstrate the effectiveness of the technique. Second, frameworks for problem-localization and failure-prediction for commercial distributed applications are proposed. The frameworks learn application's normal behavior by monitoring multiple performance metrics. They then infer normal correlations between the metrics to pinpoint the suspicious metric(s) and code region(s) where faults are manifested. Using time-series models, the frameworks can predict impending failures with up to 15–51 minutes in advance. The frameworks are demonstrated with bug cases in Apache Hadoop, HBase, Android OS, and a campus-wide Java EE application.

1. INTRODUCTION

1.1 Motivation

Distributed systems and applications are becoming increasingly pervasive in today's world providing the core infrastructures for the largest commercial and scientific applications. The complexity and scale of these applications increase continuously as they span a larger number of software components, parallel tasks[1] and computing nodes. For example, large-scale applications running in today's data centers and supercomputers span thousands of computing nodes with multiple cores per node. With this increasing trend in complexity and scale, it also becomes increasingly difficult to detect errors, performance anomalies, and unexpected behavior in these applications. System administrators need efficient techniques and practical tools for error detection without significantly slowing down the main applications and that scale to the size of the largest systems. Error-detection techniques need to operate online—as the application runs—and to detect errors and anomalies with a small delay—the time between the error manifestation and its detection has to be short.

Debugging, an important step in the software developing process, also becomes increasingly challenging as the number of concurrent tasks increases in large-scale distributed applications. With millions of cores in the largest supercomputers, fixing bugs in high-performance computing (HPC) applications is a non-trivial task. Most of the existing debugging techniques, as implemented in tools like gdb [1], TotalView [2] and DDT [3], do not automate the debugging process—developers must manually locate the root-case of problems by backtracking through interactions across processes. Also the majority of the debugging tools perform poorly at large scale—the over-head

[1]In this work we refer to a "task" and a "process" interchangeably.

of collecting large amounts of runtime information and an absence of scalable error detection algorithms generally cause poor scalability.

Many of the large-scale distributed applications require continuous availability to ensure business continuity. To reduce the application downtime, it is desirable to automatically localize the origin of failures, and, whenever it is possible, to predict impending failures based on observed symptoms in the system. Problem localization (or failure diagnosis) is a critical aspect of a fault-tolerant distributed application—to recover from a failure, one has to first localize the application component that originated the failure so it can be replaced. The granularity of the problem localization step can be a compute node, a set of abnormal tasks, a code region or even a line of code. The finer the granularity the faster the recovery phase.

When failures can be anticipated with sufficient time, mitigation techniques can be triggered in advance, such as software rejuvenation [4], microrebooting [5], redirection of further requests to a healthy server, or simply starting a backup service for the data. Many software bugs and performance anomalies develop at runtime a temporal pattern that ends up in end-user failures such as unresponsiveness, resource exhaustion or crashes. Metrics of the operating system, middleware and application can be measured to detect errors as previous research has demonstrated [4, 6–10]. Unfortunately, the majority of the existing approaches only consider a restricted set of metrics, do not analyze measurements (and relations between them) in a temporal manner, and most importantly, do not do any failure prediction.

The number of hardware and software architectures that are used today for distributed computing, along with multiple programming models, also increases the difficulty of building error-detection and diagnosis tools because of the diversity of features that need to be considered. As a consequence, it is challenging to build tools that work in general for all types of distributed architectures and programming models—features that are useful in detecting problems in a particular architecture or programming model may not necessarily be the best features to analyze to detect problems in another one.

1.2 Summary of Contributions

In this dissertation we make the following contributions:

- We design scalable techniques for error detection and problem localization in large-scale distributed applications. We evaluate my techniques in distributed applications with two common architectures:

  Commercial applications: (i) client-server multi-tier applications in which the presentation, the application processing, and the data management are logically separate processes. Example of these architectures are the Java Enterprise Edition (Java EE) standard [11]; (ii) MapReduce programming model [12] for processing large data sets. MapReduce is typically used to do distributed computing on clusters of computers.

  HPC applications: scientific and engineering applications that run in large clusters of machines with parallel tasks that communicate with the message-passing interface (MPI) [13].

  In multi-tier applications the features that we analyze are metrics from the system-to-application stack, whereas in the HPC applications we analyze per-parallel-process information (about their control-flow and execution time of code blocks).

- We introduce an error-detection and problem-localization technique, called AutomaDeD, that helps application developers find the period of time, parallel task and code region where a fault is first manifested in a HPC application. Parallel tasks' behavior is modeled statistically using semi-Markov models which are analyzed using scalable clustering and nearest-neighbor methods to pinpoint the location of errors. AutomaDeD builds a model that captures the control-flow and timing characteristics of MPI parallel tasks in an efficient way by gathering stateful information at MPI calls. AutomaDeD is able to pinpoint the origin of correctness problems, such as an application's hang, by creating a progress-dependence graph of all the MPI tasks and by finding probabilistically the least-progress task (that often originates the hang).

- We introduce a failure-prediction and error-detection technique, called Augury, that keeps track of temporal and spatial correlations between metrics in commercial applications. Metrics are gathered from different levels of the application stack—the OS-, middleware- and application-level—and a statistical model of "normal" behavior is built up in an offline phase. Examples of the metrics that are monitored are CPU- and memory-utilization, number of threads, and user-request counts. At runtime, Augury looks for deviations of "normal" relations between the metrics to predict impending failures.

- We introduce a problem-localization technique, called Orion, that uses the same statistical models of Augury to pinpoint regions of code where software bugs are manifested on. Orion uses metric-correlation deviations to find the abnormal window of time, abnormal metrics and abnormal code regions when a fault is manifested.

- All my techniques are evaluated with real-world bug cases and with fault-injection methods at large scale. We show the efficacy of AutomaDeD in isolating the origin of a bug—an application's hang at 8,000 MPI tasks or more—that occurred in the development of a molecular-dynamics simulation code at the Lawrence Livermore National Laboratory. We evaluate Augury and Orion with real-world bug cases from a multi-tier application that runs on Purdue's campus, with the Android OS, and with bug cases of BigData applications such as Hadoop and HBase.

1.3 Published Work

Individual parts of the work presented in this dissertation have appeared in the following publications:

- Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Saurabh Bagchi, Todd Gamblin: Probabilistic Diagnosis of Performance Faults in Large-Scale Parallel Applications, International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, Sep, 2012.

- Ignacio Laguna, Todd Gamblin, Bronis R. de Supinski, Saurabh Bagchi, Greg Bronevetsky, Dong H. Ahn, Martin Schulz, Barry Rountree: Large Scale Debugging of Parallel Tasks with AutomaDeD, ACM/IEEE Conference on Supercomputing 2011 (SC), Seattle, WA, Nov, 2011.

- Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski, Dong H. Ahn, and Martin Schulz: Statistical Fault Detection for Parallel Applications with AutomaDeD, 6th IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE), Stanford, CA, Mar 23-24, 2010.

- Greg Bronevetsky*, Ignacio Laguna*, Surabh Bagchi, Bronis R. de Supinski, Dong H. Ahn, Martin Schulz: AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Chicago Illinois, Jun-Jul, 2010. (* co-first authors).

During my work on this dissertation, I have contributed to other published work that is closely related to error detection and diagnosis of large-scale distributed applications but that is somewhat outside scope for inclusion:

- Greg Bronevetsky, Ignacio Laguna, Saurabh Bagchi, Bronis R. de Supinski: Automatic Fault Characterization via Abnormality-Enhanced Classification, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Boston, Massachusetts, Jun, 2012.

- Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Greg L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz: Scalable Temporal Order Analysis for

Large Scale Debugging, ACM/IEEE Conference on Supercomputing 2009 (SC 2009), Portland, OR, Nov 2009.

- Ignacio Laguna, Fahad A. Arshad, David M. Grothe, Saurabh Bagchi: How To Keep Your Head Above Water While Detecting Errors, ACM/IFIP/USENIX 10th International Middleware Conference (Middleware 2009), UIUC Illinois, Nov-Dec 2009.

My work on large-scale debugging of HPC applications presented in this dissertation has been recognized with the ACM & IEEE *George Michael Memorial HPC Fellowship* for 2011, which was announced in the Supercomputing (SC) conference in November, 2011. This award honors exceptional PhD students throughout the world whose research focus area is HPC.

1.4 Outline

This dissertation is structured as follows. We begin by surveying the related work in error detection, problem localization and failure prediction in Chapter 2. We also present in this chapter the assumed fault model for the techniques that we propose and a general view of their design goals. In Chapter 3, we present an error detection technique for scientific applications named AutomaDeD (Automata-based debugging for dissimilar parallel tasks). In Chapter 4, we focus on extending AutomaDeD's framework to large-scale systems by using scalable machine-learning methods and efficient compression algorithms. In Chapter 5, we present a diagnosis technique for scientific applications based on AutomaDeD's statistical models. We present a failure diagnosis technique, named Orion, for commercial applications in Chapter 6. In Chapter 7, we present Augury, a failure prediction and error detection framework for commercial applications. Finally, I conclude in Chapter 8 and present future work in Chapter 9.

Notice that, as described by Chapters 3, 4 and 5, AutomaDeD is a framework that is composed of multiple techniques for error detection and diagnosis in HPC applications. Although the design choices for each technique are different (as it is explained in each chapter), we term each technique as AutomaDeD throughout these chapters.

2. BACKGROUND

In this chapter, first, we give an overview of the fault model that is used in the dissertation. Second, we summarize related work in debugging, error detection and failure prediction in distributed applications in the past years. Finally, we describe the design goals for the techniques that are presented in the following chapters.

2.1 Fault Model

We follow the following definitions for failure, error and fault:

- **Fault:** is a defect, incorrect step, process or data definition in a computer program. Faults are the hypothesized cause of an error. In most cases, faults remain dormant for some time and once they become active, they cause an incorrect system state, which is an error. An example of a fault is a software bug (i.e., a programmer's mistake).

- **Error:** is the situation when "things go wrong" in the system because a fault is manifested. Formally, an error happens when the system's states deviates from the correct state. As a consequence, an error is the part of the system's state that may lead to its subsequent service failure. An example of an error is when a software bug is activated when the buggy code is executed.

- **Failure:** is defined as "an event that occurs when the delivered service deviates from correct service". The key idea is that it can be observed by the application user. An example of a failure is when an application hangs because of an error.

Some of the techniques presented in this dissertation are executed online—when the application runs—to detect and diagnose problems in the application. To differentiate from related work where problem detection is performed offline—for example when software bugs are found using static source-code analysis—we use the term *error detection* instead of *fault detection*. We consider that the errors that are detected could be caused due to multiple reasons such as incorrect system configurations or deployments, software bugs, or unexpected performance problems.

We term the process of identifying the nature and cause of a failure (or error) as "*diagnosis*". We use the terms "*diagnosis*", "*problem localization*" and "*root-cause analysis*" interchangeably. We consider "*debugging*" as a diagnosis process that is done offline by a human being rather than online by a computer program.

2.2 Summary of Related Work

2.2.1 Debugging Parallel Applications

Traditional debugging techniques, including sequential debuggers such as gdb [1] and "printf debugging," require that users manually trace the origins of their coding error. Traditional parallel debuggers, such as DDT [3] and TotalView [2], extend these techniques to allow tracing of multiple processes. They provide convenient interfaces to the state of these processes, but the main procedure of identifying errors remains manual. Overall, the traditional techniques require a significant amount of user experience, intuition and time, and are ultimately unreliable for debugging of large, complex parallel applications.

Several debugging tools detect bugs in large-scale applications without relying on much manual effort. These typically focus on detecting violations of deterministic and statistical properties of the applications. Deterministic tools can validate certain properties at runtime; any violation of these properties during an execution is reported as an anomaly. For example, FlowChecker [14] focuses on communication-related bugs in MPI libraries. It extracts information on the application's intentions of message passing (e.g., by matching MPI Sends with MPI Receives) and at runtime checks whether the data movement conforms to these intentions. Bug localization follows directly: the data movement function that caused a discrepancy is the location of the bug.

Statistical tools [15, 16] detect bugs by deriving the application's normal behavior and looking for deviations from it. For example, if the behavior of a process is similar to the aggregate behavior of a large number of other processes, then it is considered correct and different behaviors are considered incorrect. Mirgorodskiy et al. [16] monitors the application's timing behaviors and focus the developer on tasks and code regions that exhibit unusual behaviors. This focuses on function call traces to identify the trace that is most different from other traces. DMTracker [15] uses data movement related invariants, tracking the frequency of data movement and the chain of processes through which data moves.

While the above tools are effective in their own domains, their primary weakness is that their designs do not consider scalability. Typically, these tools collect trace data during the application's execution and write it to a central location. They then process the data to detect potential problems. In contrast, the techniques presented in this dissertation,analyzes the application's behavior online, without any central bottlenecks. In this respect, the closest prior work to ours is STAT [17–19], which provides scalable detection of task equivalence classes based on the functions that the processes execute. STAT uses MRNet [20], a tree-based overlay network, to gather and to merge stack traces across tasks and presents the traces in a call-graph prefix tree that identifies task equivalence classes. STAT removes problems associated with a central bottleneck by reducing the trace data as part of a computation being performed within the overlay network through a custom reduction plug-in. STAT focuses primarily on the state of the application once an error manifests itself, whereas we focus on scalable analysis of the entire application execution.

2.2.2 Error Detection via Metrics Analysis

In troubleshooting computer systems, a conventional approach widely used in practice, is to measure values of various metrics and compare these values to (manually set or automatically deduced) thresholds. More subtle techniques compare not instantaneous values, but trends in values to thresholds.

In the research literature, there is a volume of work in identifying errors (latent or already manifested) through analysis of metric values, typically through variants of machine learning algorithms [7–10, 21]. The canonical structure for this work is to create models of expected behavior based on labeled (typically) or unlabeled measurements of metrics, and then during runtime collect measurements of the same metrics, use the same algorithm to generate a model and identify if the generated model is significantly different from the prior-learned models, which would indicate an anomaly. Most of these approaches go further in trying to localize the identified problem to a small part of the source code, where the localization is predicated on identifying which parts of the code cause the model generated through runtime observations to differ from the prior models. There is a specific line of work within localization, which seeks to automatically generate signatures of problems that have been observed in the system and fixed through manual means [10, 22]. These signatures comprise values of multiple metrics over time. Later when a problem arises and its signature is deduced, if the signature matches that of a prior problem, then the fix from the prior problem can be applied, possibly with some modification.

2.2.3 Model Checking

Model checkers are useful for checking small applications against specifications [23–26]. Due to the exhaustive nature of model checking, it is not feasible for most real-world applications. In addition, it is more appropriate for checking against specifications. Liu *et al* [27] proposes a technique that does live model checking and provides execution replay. The programmer writes a predicate that is invariant throughout the execution, and this predicate is checked as the application runs. When the predicate is violated, the system states leading to the violating state are given as output. While this approach works well for specifications, as the instruction that changes the system state from conforming to violation is usually the root cause of the problem, for other types of problems such as performance problems, the errors may have accumulated from different regions of the code before the specification is violated. My techniques do not assume that the instruction that causes the system state to be a violation is the root cause of the problem, and are thus more applicable to a wider range of problems.

2.2.4 Problem Localization

There is a volume of recent work that utilizes statistical methods to detect and localize problems. Some of the work analyzes application logs [28–31]. However, there is often a one-to-many mapping between the log record corresponding to the problem and the actual code regions that could be the source of the problem. [32–34] analyze request flows to diagnose problems in request-processing applications. Other work analyzes metric values, typically using machine learning algorithms [35–38]. In [35] and [36], the signature of the current problem is compared to a database of known problems. If there is a match, the diagnosis and fixes used previously can be reused again. This approach is suitable for problems that are not easy to fix even if the root cause is known (e.g., overloaded servers), problems due to the environment, or hardware problems. In other situations, once a problem is diagnosed and fixed, it will not occur again, limiting the usability of the tool. The overall approach of [37] and [38] is similar to the approaches presented here in that machine learning models are trained based on training data, and the models are then used to classify current state of the system. If the system is in an abnormal state, the metrics that are most abnormal are reported to help localize the failure's origin. In addition to this, when this is not enough to pinpoint the location of the fault, my techniques go a step further and provide a ranking of most suspicious code regions to reduce the programmer's effort needed to fix the problem.

2.2.5 Failure Prediction

Failure prediction methods have been proposed for hardware [39–42] and software [9, 43, 44] failures. The majority of the proposed techniques keep track of events in the monitored system to build up symptoms that correlate to previously observed failures. Examples of events that are monitored are hard drives features and system logs for hardware failures, and system performance metrics such as cpu- and memory-usage for software failures. The main objective of failure prediction is to anticipate a failure so that mitigation actions can be taken. A common approach—used in both hardware and software failures—is to keep track of sequences of errors, i.e., fault manifestations that have not been resulted in an end-user failure, to anticipate failures when a particular sequence is found in a fixed period of time [40, 43], for example when five errors occur in a short time window. Another approach, when metrics of the system are monitored, is to model measurements of metrics with regression and time-series models and to forecast future values of the metrics [4]. If forecasted values go beyond a threshold, a problem is said to be anticipated.

The existing approaches for failure prediction suffer from one or more of the following problems: (a) a limited number of metrics or events of the system are considered—if a fault is manifested in a metric or event that is discarded a priori, this fault will not be detected which reduces the possibility of predicting a failure; (b) because of the complexity of the statistical models that are used, many techniques do not scale to a large number of metrics and computing nodes in large-scale distributed application. Therefore these techniques cannot be used online to anticipate failures because they cannot keep up with the distributed application demands [9]; (c) the failure prediction technique does not do any failure detection, or vice versa, often because the two types of techniques are decoupled. This limits their usability in real-world scenarios where, if a failure cannot be predicted, at least it could have been detected with a short delay so that system administrators can take recovery actions in a short time from the problem manifestation.

2.3 General Design Goals

Here we describe the set of desired characteristics or goals for the techniques that are presented in this dissertation:

- **Online operation:** we want our techniques to operate in an online manner, i.e., to be executed as the application runs. Many faults are only manifested at runtime, so online techniques are needed to detect these classes of faults. Online operation also permits to detect errors close to the manifestation time so that recovery actions can be taken before they become failures and visible to end users.

- **Minimum performance degradation of the monitored application:** I aim at designing techniques that degrade minimally the performance of the monitored applications, or to avoid affecting them at all if it is possible. Examples of mechanisms that can affect the performance of the monitored applications are (i) software instrumentation (e.g., by using library wrappers or binary instrumentation), and (ii) system-to-application-stack metrics collection. We aim at using instrumentation that gives sufficient granularity for the error-detection and problem-localization but that does not slow down the execution time of the application substantially. The typical slowdown for our techniques is between 1.01x and 1.67x.

- **Scalability:** we want our techniques to be scalable to the size of the largest distributed applications and machines of today. Our techniques should handle growing amounts of work in a graceful manner, where the amount of work can grow by various ways such as a large number of computing nodes, parallel tasks or software components.

- **Problem localization:** We aim at techniques that can help developers and system administrators locate the source of a problem quickly. This could be in the form of pinpointing the source code block where an error is first manifested or propagated to, or by finding metrics from the system-to-application stack that are highly related to a problem. The idea is that, after an error or a failure is detected, the detection report should also provide useful information for developers and system administrators that facilitate locating and fixing the original fault.

- **Low detection delay and look-ahead time:** Two type of time-based measures are important when designing our techniques: (i) *error-detection delay*: the elapsed time between a fault manifestation (i.e., an error) and its detection; (ii) *look-ahead time*: the time between a failure is anticipated and its actual occurrence (if it occurs). We aim at reducing these times as much as possible. The first one helps in reducing the downtime of a system while the second one allows the activation of mitigation techniques in advance.

3. ERROR DETECTION IN SCIENTIFIC APPLICATIONS

In this chapter we present *AutomaDeD* (Automata-Based Debugging for Dissimilar Parallel Tasks), a framework for detecting errors in HPC applications. We first give background information about debugging and error-detection in clusters of MPI applications. Next, we describe the methodology that we use to model the behavior of MPI tasks and that we use to isolate problems. Finally, we show an experimental evaluation of the framework with fault injections and a bug in the MVAPICH library (a portable implementation of the message-passing interface).

3.1 Introduction

The number of cores used in large scale systems already exceeds a million cores. As a result, the challenge of developing correct, high performance applications is also growing. When an application does not complete or completes with incorrect results, the developer must identify the offending MPI task and then the portion of the code in that task that caused the error. Traditional parallel debugging tools [2, 45–47] often perform poorly at large task counts. We focus on developing a detection tool set that identifies the offending task and, to a customizable granularity, the relevant portion of code within the task.

We present *AutomaDeD*, a tool set that achieves this goal of focusing debugging efforts to improve developer efficiency. It performs runtime monitoring of a parallel application to build a statistical model of the application's typical timing and control flow behavior. The typical use case for *AutomaDeD* is that a user suspects a run of an application is erroneous and would like to get some guidance to what parts of the application code to focus on for debugging. *AutomaDeD* achieves this by identifying the period in time, the task(s), and the *error site*, the region of code, where a fault first manifests itself. Thus, *AutomaDeD* provides the basis for eventual root cause diagnosis including identification of the exact erroneous line of source code.

This work makes technical contributions in *two broad areas*. First, we describe *a model to characterize the behavior of parallel applications*. Second, we present methods that *compare the behavior of tasks in a parallel application in time and in space* to identify the error site. *AutomaDeD* models the the control flow and timing behavior of application tasks as *Semi-Markov Models (SMMs)* and detects faults that affect these behaviors. States of these SMMs represent regions of application code and edges represent execution progress from one region to another. SMMs capture the probability of transitioning from one region to another and the distribution of times spent in each region. We delimit code regions by MPI calls and use MPI calls (along with call stack information) and the computation interleaved between them as two different kinds of states in SMMs.

AutomaDeD examines how each task's SMM changes over time and relates to the SMMs of other tasks to identify the task and code region where a given fault is first manifested. First, *AutomaDeD* detects which time period in the execution of the application is likely erroneous. *AutomaDeD* then clusters task SMMs of that period, and performs *cluster isolation*, which uses a novel similarity measure to identify the task(s) suffering from the fault. Finally, *transition isolation* detects the transitions that were affected by the fault more strongly or earlier than others, thus identifying the code region where the fault is first manifested. In addition to focusing the developer on the root cause of their bug, *AutomaDeD* also enables the use of traditional debuggers, such as gdb [1], at previously infeasible scales by focusing them on the time period, tasks and code regions that are most likely to have a bug.

Our evaluation injects synthetic errors into six applications from the NAS Parallel Benchmark (NPB) suite [48] at random time points and in randomly chosen tasks. The errors include delays, hangs in application tasks, interference due to execution of an extra CPU- or memory-intensive thread on an application compute node and

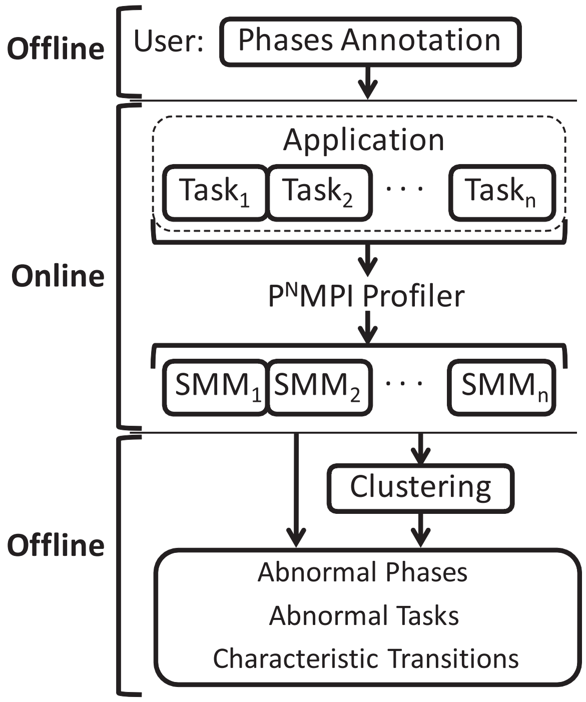


Fig. 3.1.: Design of *AutomaDeD*

message drops and duplication. Our results demonstrate that *AutomaDeD* correctly identifies the time period that is likely erroneous in 90% of our trials for delays, hangs and message faults and in 70% of our trials for interference faults. Given the correct time period, *AutomaDeD*'s cluster isolation achieves over 80% accuracy for delays and hangs, 40% for message faults and 70% accuracy for interference faults. Given the correct cluster, it isolates the injected transition with 90% accuracy for delays and hangs and 50% accuracy for interference faults.

3.2 Approach

As Figure 3.1 shows, this version of *AutomaDeD* consists of both on-line and off-line mechanisms. An on-line mechanism gathers data about executions into an SMM

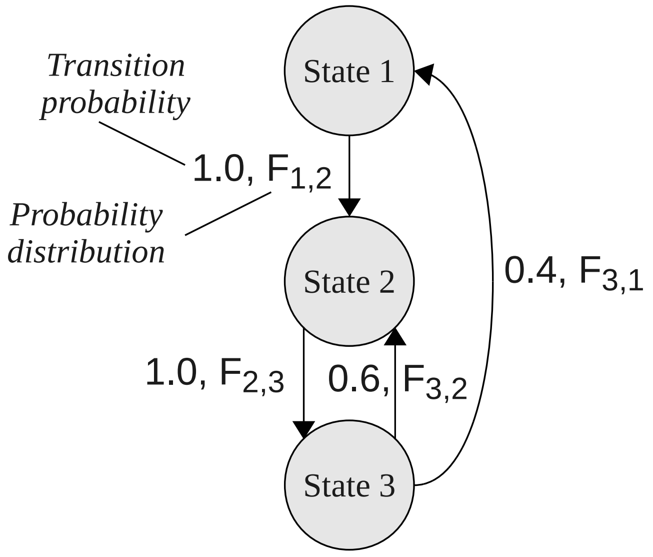


Fig. 3.2.: Example of a Semi-Markov Model

database. *AutomaDeD*'s off-line mechanisms then use this data to derive a deeper understanding of the application behavior, particularly when bugs are manifested.

3.2.1 Semi-Markov Models

We model the control flow and timing properties of application tasks in order to debug common anomalies. We track control flow as a sequence of application states, defined as MPI calls (including their arguments and call stack) or the computation interleaved between them. We maintain the amount of time each task spends in each state to capture temporal aspects of the states. Given the expense of maintaining full traces, we model task behavior as a *Semi-Markov Model* (SMM), a finite automaton of task states and transitions where the task spends a random amount of time in each state and randomly selects its next transition with no dependence on its history.

Figure 3.2 shows a sample SMM with edges labeled by the probability of transitioning from one state to another and the probability distribution of the time preceding the transition. In the above SMM, tasks in state S_3 transition to state S_1 40% of the time and to S_2 the other 60%, with the times that precede the transitions sampled from distributions $F_{3,1}$ and $F_{3,2}$, respectively. Probability distribution $F_{i,j}$ explains the amount of time the application spends in state i before transitioning to state j. We compute the SMM states, transitions and probability distributions from program traces captured on-line by a P^{n}MPI-based wrapper library [49] that intercepts all calls to MPI functions. We use the observed normalized frequency of each transition as its transition probability. Section 3.3.1 explains how we derive time distributions.

3.2.2 Overview of Analysis

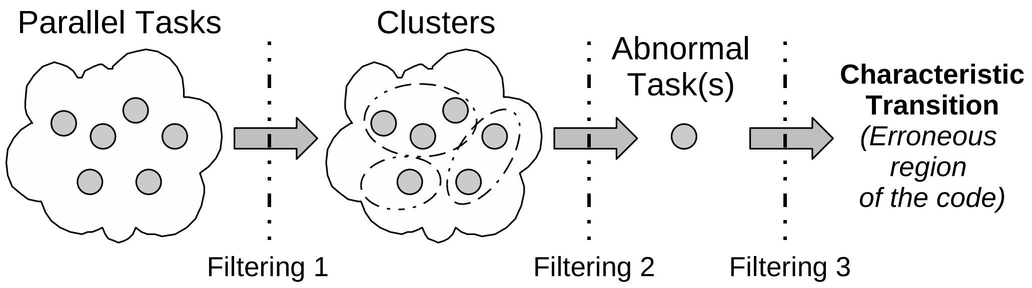


Fig. 3.3.: Problem-size reduction with *AutomaDeD*

The SMM abstraction couples the dynamic execution of an application with distinct regions of its code. Thus, we can focus the developer's attention on the tasks and regions of code that are behaving anomalously. Figure 3.3 shows the several stages in which we accomplish this goal. First, we divide the application's execution into a series of time periods called *phases*. Naturally, applications behave according to a repetitive pattern for periods of time and then their behavior changes, to a different repetitive pattern or some random pattern. We divide the period of time of repetitive behavior into smaller time periods, which we call phases. Thus, across the phases within one repetitive pattern boundary, we expect the application behavior to be statistically identical. *AutomaDeD* then computes an SMM for each task within each phase and then clusters similar SMMs for each phase. This clustering may partition the tasks based on correct differences between them, such as with master-slave applications that have two correct partitions. Alternatively, it may identify behavioral differences due to a bug. *AutomaDeD* either compares a task's SMMs from different phases or the task clustering from different phases to determine the phase during which a bug is first manifested. It can also compare SMMs or their clusterings to those from prior, correct executions. If no sample runs are available, *AutomaDeD* calibrates its bug detection algorithms based on the first phase, which works well in practice because we target rare, hard to find bugs, which manifest themselves after a few iterations of the main processing loop.

Once *AutomaDeD* identifies a faulty phase, it proceeds to identify the task cluster or individual task in which the bug is first manifested. *AutomaDeD* compares SMMs or clusters across phases to identify the SMM or cluster that has changed the most from the normal phases. *AutomaDeD* can again use SMMs or clusters from prior, correct executions or earlier phases of the same execution. *AutomaDeD* also compares the individual state transitions within the faulty phase to find the first unusual transition, which may identify the error site. Alternatively, the most unusual SMM transition of the faulty task may identify it.

Thus, *AutomaDeD* focuses the developer's debugging efforts through multiple steps. First, it identifies the faulty execution phase. Then it finds the faulty task or group of tasks. Finally, it locates the error site. The granularity of this identification is a state in the SMM. Thus, *AutomaDeDdoes not* identify the root cause of the error and cannot identify the manifestation to a very fine granularity, such as line of code. However, it does significantly reduce the amount of information that must be considered when performing a root cause analysis.

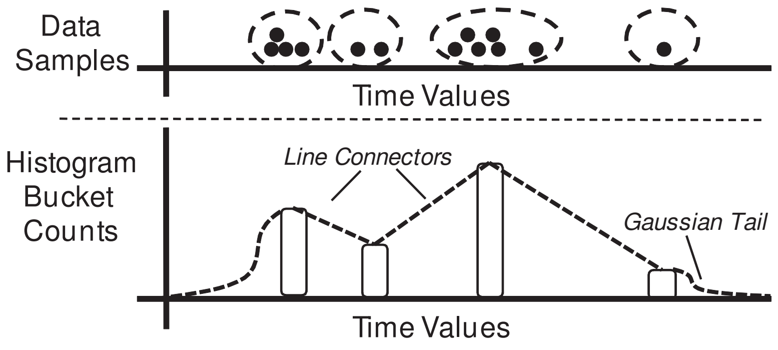


Fig. 3.4.: Example of histogram construction

3.3 SMM Mechanisms

3.3.1 Creating Time Distributions

We consider two methods for deriving the time probability distributions that explain the time spent by a task in the SMM states. In one, we assume that the time values follow a Gaussian distribution. In the other, we compute a histogram of ranges of the observed time values, instead of assuming a particular distribution.

Assuming Gaussian distribution has several advantages. First, we can easily calculate the parameters of a Gaussian distribution given enough sample points. By using maximum-likelihood estimation, we only need to calculate the mean and standard deviation of the data points. Second, it is a well-known distribution with a rich theory. However, a Gaussian distribution is not appropriate for state transitions that have multi-modal or asymmetric behavior. The former can occur when different code within a compute region is executed at different times and the latter occurs when the time that precedes a transition is consistent except for spikes due to system or network interference.

Histograms provide a more detailed fit to the observed data. The basic approach, which Figure 3.4 shows, divides the observed data points into a number of equal-sized buckets. The probability of a particular bucket is the fraction of data points within it. Since timing data may have outliers orders or magnitude above the median, equal-sized buckets can aggregate most data points into a single bucket, providing poor resolution. We therefore used variable-sized buckets via an online clustering algorithm. We assign each new data point to its own bucket. If the resulting number of buckets rises above a threshold, we merge the two buckets with the closest means. We derive a continuous probability distribution from the discrete histogram by linearly connecting adjoining bucket counts and modeling the regions beyond the smallest and largest buckets using the lower and upper halves of Gaussian distributions, which models the probability of observing new extreme values.

The basic tradeoff between these two distributions is that Gaussians are cheaper (in terms of computation and memory cost to create and to query) and more constrained, while histograms are more expensive but very flexible. Evaluating both options provides significant information about the basic tradeoffs of this design parameter, thus illuminating the potential of other statistical models such as mixed-Gaussian distributions and Kernel Density Methods [50].

3.3.2 Comparing Task SMMs

AutomaDeD detects faulty phases and tasks and performs task clustering by comparing SMMs to each other. We define an SMM distance metric that reflects the differences between the control flow and timing behaviors of their respective tasks. The difference between two SMMs is the sum of the differences in their transition probabilities and transition time distributions.

Given two SMMs A and B, let S_A and S_B be their sets of states, and T_A and T_B be their sets of transitions. A transition between two models are the same only if their source and destination states are the same. Also let $d_{s,i}$ be the state transition probability distribution for state $s \in S_i$, and let $d_{t,i}$ be the time probability distribution for transition $t \in T_i$. The difference between A and B is:

$$Diff(A,B) = \sum_{s \in \mathbf{S}} D(d_{s,A}, d_{s,B}) + \frac{1}{\nu} \sum_{t \in \mathbf{T}} D(d_{t,A}, d_{t,B}) \tag{3.1}$$

where $\mathbf{S} = S_A \cup S_B$, $\mathbf{T} = T_A \cup T_B$, $D(d_{r,A}, d_{r,B})$ is the difference between a pair of probability distributions $d_{r,A}$ and $d_{r,B}$, where r is a state or transition. ν corresponds to a weighting factor defined in Section 3.3.3 that weighs differences on transitions with consistent timing behavior above those with poor information content. We define the metric $D(d_{r,A}, d_{r,B})$ as:

$$D(d_{r,A}, d_{r,B}) = \begin{cases} L_2(d_{r,A}, d_{r,B}) * \alpha & \text{if } r \in A \text{ and } r \in B \\ 10 & \text{otherwise} \end{cases} \tag{3.2}$$

$L_2(d_{r,A}, d_{r,B})$ is the L_2 norm between the probability distributions [51], $L_2(d_{r,A}, d_{r,B}) = \int_{-\infty}^{\infty} |d_{r,A}(j) - d_{r,B}(j)|^2 dj$. The integral is over the space of possible events (state transitions or transition times).The parameter α gives greater weight to differences in time distribution with distant means, μ_d and $\mu_{d'}$. For time distributions it is equal to

$$\alpha = 1 + \frac{|\mu_{d_{r,A}} - \mu_{d_{r,B}}|^2}{(\mu_{d_{r,A}} + \mu_{d_{r,B}})/2} \tag{3.3}$$

and $\alpha = 1$ for state transition distributions.

In most cases $D(d_{r,A}, d_{r,B})$ is below 10 for transitions and states r that appear in both A and B. As such, if r appears in one but not the other, $D(d_{r,A}, d_{r,B})$ was set to 10 to make differences in application control flow more significant than differences in its timing behavior.

3.3.3 Normalized SMM Comparison

Different SMM transitions will have very different timing properties, with a variety of means, standard deviations and distribution shapes. Differences between SMMs on a transition that has consistent timing and a tightly focused distribution can be very informative. In contrast, if the transition is noisy, the differences are most likely due to system interference. *AutomaDeD* focuses on the critical differences between two SMMs by looking at the "normal" difference between the SSMs of a sample set and weighting $D(d_{t,A}, d_{t,B})$ accordingly. Thus, given a transition t and a set M of sample SMMs, we define the weighting factor ν as the root-mean-square of D on this transition among the members of M:

$$\nu = \sqrt{\frac{\sum_{A,B \in M, A \neq B} D(d_{t,A}, d_{t,B})^2}{|pairs(t, M)|}} \tag{3.4}$$

where $|pairs(t, M)|$ is the number of SMM pairs in M that both have transition t. In the absence of sample runs, ν for a given transition in a given phase of the faulty run is computed by summing over SMMs in the run's other phases.

This weighting scheme overcomes a commonly observed effect where certain transitions have multi-modal timing characteristics—very consistent timing behavior within each mode and sudden shifts to a different mode either within a given run or across multiple runs. This may be caused for example by a computation that executes the same set of instructions but takes very different times depending on whether the data is cached or not. For such behavior, the value of ν will be high, thereby weighing down the difference metric D.

3.3.4 Clustering Tasks' Models

AutomaDeD detects behavioral clusters by using Hierarchical Agglomerative Clustering (HAC) [52] on the SMMs of all application tasks. HAC initially sets each task to be in its own cluster. During each iteration, HAC merges the two most similar clusters into a single cluster, so that it has one cluster less after that iteration. Cluster difference is defined as the smallest difference between any member of one cluster to any member of the other cluster. These steps are repeated until the minimum difference between any pair of clusters is above a given threshold (i.e., no two clusters are similar enough to merge).

HAC requires a threshold that defines the normal difference of similar tasks. *AutomaDeD* chooses this threshold by having the developer provide the number of clusters that accurately describe the application's expected behavior. For example, a relaxation algorithm with non-periodic boundaries operating on a 2-dimensional grid is best described by a 9 clusters (one for the interior, and one for each side and each corner region). However, it should have a single cluster if the boundaries are periodic. *AutomaDeD* applies HAC on SMMs of a set of training phases (assumed to have few bugs), identifying the average threshold that produces the desired number of clusters. We use this threshold for subsequent clustering. If sample runs of the application are provided, *AutomaDeD* trains on phases in these runs. Otherwise, it trains on the given run's first phase, which we assumed is fault-free.

The resulting clustering organizes tasks into behavioral groups that reflect the effect of the bug on the application's normal behavior. This helps to identify the time and the location when the fault was first manifested.

3.4 Error Detection Procedure

We describe the procedure that a user employs to isolate a bug using *AutomaDeD*. Figure 3.1 shows the complete sequence of steps. *On-line* steps occur when the program executes, while *off-line* steps occur after execution. The next sections describe each step.

3.4.1 Phases and Epochs

AutomaDeD models the behavior of discrete regions of application execution that the developer identifies via source code markers. The term *phase* denotes a region of execution, such as a time step, that repeats multiple times. Phases are grouped into `phase sets`, where all phases in a set are assumed to behave similarly to each other. For example, adaptive mesh refinement applications periodically re-partition their work and meshes. Thus, individual iterations may be identified as phases while iterations between adjacent re-partitionings may be grouped into a set. Developers annotate phases and phase sets in their code by adding calls to `MPI_Pcontrol`, a special function call that is intercepted by our wrapper library.

3.4.2 Faulty Phase Detection

AutomaDeD detects the phase during which a fault was first manifested using one of two algorithms, depending on how it effects application behavior. Currently the user of *AutomaDeD* must try both faulty phase selection mechanisms. We leave automation of this selection to future work.

If the effects are temporary (e.g., temporary delay due to unusual erroneous control flow), *AutomaDeD* searches for the phase that differs from all other phases. If *AutomaDeD* has a set of sample runs, it compares each phase to its counterparts in those runs. It can either compare each task's SMM directly to its sample counterpart or it may compare each phase's clustering to the clustering of its counterpart phase. For the former we use the SMM difference metric from Section 3.3.2, with the difference between two phases defined as the squared sum of the differences between their respective task SMMs. For the latter we use the Mirkin difference metric [53], which is the fraction of task pairs that are grouped differently in the two clusterings, (i.e., tasks T_1 and T_2 are in the same cluster in one clustering and not in the same cluster in the second, or vice-versa). Then for each phase we compute a "deviation score", which is the sum of the squared distances from this phase in the faulty run to the same phase in the sample runs. We identify the phase with the highest deviation score as faulty. If no sample runs are provided, *AutomaDeD* compares each phase to all others within the faulty run using either of the above metrics to compute each phase's deviation score. We identify the phase that differs most from the others as faulty. When sample runs are provided, ν weighting terms are computed from the SMMs of these runs. When they are not provided, the ν used for each phase's comparisons is computed from the other phases in the faulty run.

If the effects are permanent (e.g., a runaway thread that interferes with the application), *AutomaDeD* identifies the phase when application behavior shifted. If *AutomaDeD* has sample runs, it computes deviation scores as above but then uses k-Means Clustering [52] to divide the phases into two clusters: those that are similar to the sample runs (low deviation) and those that are different (high deviation). We identify the earliest phase in the high deviation cluster as faulty. Without sample runs, *AutomaDeD* identifies the pair of adjacent phases that are most different according the SMM or clustering difference metrics. The later phase in this pair is judged to be faulty.

3.4.3 Pinpointing Faulty Task(s) and Error Sites Using SMM Analysis

AutomaDeD provides two complementary mechanisms to identify the faulty task(s) and the error site. We describe the first mechanism, which compares SMMs and clusterings, here. We discuss the second, which is based on individual transitions, in Section 3.4.4. Successful identification of the faulty cluster greatly simplifies determining the root cause. Cluster isolation is particularly helpful when the manifestation of a bug results in a cluster with a single task.

AutomaDeD clusters the tasks of the faulty phase and then identifies the most unusual cluster by computing its deviation from the other clusters. If we have a set of sample runs, we compare each cluster to them using the SMM or clustering difference metric (comparison is focused on the phase identified as faulty). The SMM cluster difference is simply the sum of the squared SMM differences between the SMMs of member tasks in the faulty phase and their SMMs in the same phase of a given sample run, divided by the number of tasks. The clustering difference metric is a variant of the Mirkin difference where the deviation from sample phase clustering $C' = \{c'_1, ...c'_n\}$ of test cluster c is the fraction of its member task pairs that appear in different clusters in C'. Each cluster's overall deviation score is then the squared sum of its differences with respect all the sample runs. If no sample runs are provided each cluster is compared as above but to the other phases of the faulty run instead of the same phase of the sample runs.

We also locate the error site based on task clustering when we identify the *characteristic transition* (CT), the transition that most distinguishes the faulty cluster from the other clusters. Since bugs can cause these behavioral differences, CTs direct developers to the root cause.

For SMMs A and B, we define $CT(A, B)$ as (t, χ) where t is transition that most contributes to the dissimilarity metric $Diff(A, B)$ and χ is the magnitude of this contribution. Given a cluster $c = \{M_1, M_2, ..., M_n\}$, we compute the cluster's CT by evaluating $CT(M_i, M'_j)$ for each pair $(M_i \in c,\ M'_j \notin c)$. The CT of c is then the transition that is the CT of the most SMM pairs. If this selects more than one transition, the CT is transition with the largest average contribution magnitude. Since this method does not always produce the correct faulty transition as the top CT, *AutomaDeD* can also present the top several choices to the developer for closer examination.

3.4.4 Detection Using Transition Analysis

Our SMM-based cluster and transition isolation methods are too coarse if the effects of the bug propagate to the entire application and will fail to identify the first task(s) and transitions that the bug impacted. We can overcome this difficulty by observing individual state transitions, looking for the first that takes an unusual amount of time compared to the transition behavior seen in sample runs or earlier phases.

If the faulty effects are temporary, *AutomaDeD* computes the typical behavior of each SMM transition as a probability distribution (Gaussian or Histogram) of its observed times in the sample runs or first phase, after discarding the top and bottom 1% of the times. *AutomaDeD* uses these distributions to compute the probability of observing the time preceding each transition of the faulty phase. We then identify low probability transitions through K-Means clustering with $K = 2$, using the log of the probability to improve sensitivity to low values. We select the earliest low probability transition as the CT, which also identifies the faulty task. *AutomaDeD* can also present later low probability transitions on other tasks in case the starting times of the transitions do not correctly identify the CT.

If the faulty effects are permanent, *AutomaDeD* looks for a sudden change from one type of application behavior to another. Specifically, it scans each transition t in each task SMM M to locate the largest increase in $\theta = stdDev(t) * \overline{\nu}$, where $stdDev(t)$ is the standard deviation in the observed times preceding t. When sample runs are provided, $\overline{\nu} = \frac{1}{\nu}$, where ν is the noise weighting factor discussed in Section 3.3.3. Otherwise, $\overline{\nu} = stdDev(t)$, which is another way to reduce the algorithm's sensitivity to outliers.

θ measures the variation of the transition, which increases significantly when its behavior changes, as its prior behavior does not predict its new behavior well. *AutomaDeD* selects the transition that provides the best balance between occurring before other transitions and having a high θ. This is done by comparing transitions t and t' using to the following relation:

$$(t_{ts}, \theta) \succ (t'_{ts}, \theta) \equiv \begin{cases} \theta * (1 + t'_{ts} - t_{ts}) > \theta' & \text{if } t_{ts} < t'_{ts} \\ \theta' * (1 + t_{ts} - t'_{ts}) > \theta & \text{if } t'_{ts} < t_{ts} \end{cases} \tag{3.5}$$

where t_{ts} and t'_{ts} are the timestamps of t and t'. Thus, we consider t a better choice (ordered larger) than t' if either it has an earlier timestamp and θ is larger than θ' after being adjusted by a factor that proportionally compensates for the difference in their timestamps or it has a later timestamp and θ is larger despite θ' being inflated by the same factor.

3.4.5 Visualization of Results

AutomaDeD presents the cluster and transition isolation results through the clustered SMMs of the faulty phase, focusing on the faulty cluster and the CT. Figure 3.5

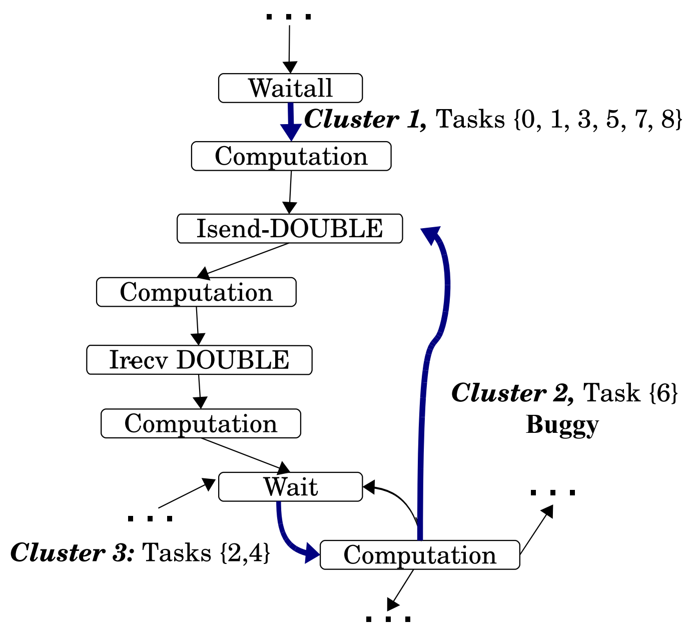


Fig. 3.5.: Output format of *AutomaDeD* after the debugging process is completed.

shows an example of the output for a 9 task NAS benchmark BT when a 10 second delay was injected into task 6 before execution of the selected `MPI_Isend` (we show only a portion of the SMM). Bold edges indicate the CTs; the clusters appear as their labels. The cluster associated with the edge (Computation, Isend-DOUBLE) corresponds to the faulty cluster.

3.5 Experimental Evaluation

3.5.1 Fault Injection Types

We empirically evaluate the effectiveness of *AutomaDeD* by injecting synthetic faults into six applications in the NAS Parallel Benchmark suite: BT, CG, FT, MG, LU and SP [48]. We omitted EP because it performs almost no MPI communication and IS because it uses MPI in only a few locations in the code, making MPI-based state demarcation inappropriate. Our fault injector, built on top of P^{N}MPI, dynamically injects a wide array of software faults at random MPI calls during MPI application runs. It supports three main classes of faults:

- Local livelock/deadlock or transient stall; emulated via a finite loop of 1, 5 or 10 seconds (`FIN_LOOP`) or an infinite loop (`INF_LOOP`)
- MPI message loss and duplication; emulated by dropping (`DROP_MESG`) or repeating (`REP_MESG`) a single MPI message,
- Extra CPU- or Memory-intensive thread; emulated by starting up a thread with a perpetual-increment loop (`CPU_THR`) or a loop that randomly reads from/writes to a 1GB region of memory (`MEM_THR`), that interfere with the remainder of the application's execution.

Our experiments ran each benchmark with input size A and 16 tasks. We executed all tasks on four-socket, quad-core nodes (the Hera cluster at LLNL), with 2.3Ghz Opteron processors, 32GB RAM per node and InfiniBand interconnect. We injected each fault type into a random task and MPI operation type (Blocking and Non-Blocking Sends and Receives, All-to-Alls, etc.), ensuring that over the entire experiment, each task and MPI operation type was injected with each fault type. For each case, we performed at least 10 random injection runs, totaling approximately 2,000 injection experiments per application. In each run we injected a single fault into a random instance of the target operation type on a random task. The execution of each application was partitioned into approximately 5 phases; the exact number depended on the application's original iteration count.

3.5.2 Results of Debugging Faults

We evaluate the accuracy of *AutomaDeD* in identifying the following aspects of the injected fault:

- The phase with the injected fault (faulty phase)

- The cluster that contains the task with the injected fault (cluster isolation)
- The error site of the injected fault (transition isolation)

We evaluate *AutomaDeD* with and without sample runs. Using sample runs corresponds to when the developer can execute an application multiple times to establish its normal behavior before analyzing a given faulty run. We evaluate two types of sample runs. For each application `A` the `FaultFree(A)` set consists of 20 runs with no injected faults, which models an ideal set of sample runs. The `Fault10(A, F)` set includes `FaultFree(A)` as well as 2 additional runs of `A` in which fault `F` was injected. This set models the more common case where application runs are affected by an infrequent non-deterministic bug that affects a certain fraction of runs (in this case ∼10%). Our experiments that do not use sample runs, denoted `NoSample`, omit any runs in which faults were injected during the first phase in order to ensure a more informative evaluation. We also omit such runs when analyzing `CPU_THR` and `MEM_THR` faults, regardless of whether or not sample runs are provided, since they provide no information about the application's behavior before the fault.

Detection of the Faulty Phase

We begin by evaluating *AutomaDeD*'s ability to detect the phase in which the fault was injected. If *AutomaDeD* does not have sample runs, the algorithm identifies the phase that is most different from the others using either the cluster-based metric or the individual task SMM-based metric. If it has sample runs, *AutomaDeD* uses one of these metrics to determine the phase that is most different from its counterpart in those sample runs.

Figure 3.6 shows the average accuracy over all applications of faulty phase detection. All of our graphs show the runs on the Y-axis in which *AutomaDeD* identifies the phase, cluster or transition relevant to the injected fault. The data series correspond to using the two metrics with each sample run configuration (`FaultFree`, `Fault10` and `NoSample`) and the different distribution methods used for the times preceding transitions (`Gaussian` and `Histogram`).

We observe that the SMM-based metric detects faulty phases more accurately than the cluster-based one, with detection accuracy over 90% for most fault types. However, the cluster-based metric better detects `CPU_THR` and `MEM_THR` when sample runs are available. In general, sample runs significantly improve faulty phase detection accuracy, with `FaultFree` and `Fault10` generally exceeding `NoSample` by 20%-30%. The difference is even larger for `CPU_THR` and `MEM_THR`. Further, `FaultFree` and `Fault10` sample runs provide similar accuracy, which suggests that moderate noise levels do not impact the SMM representation and *AutomaDeD*'s analyses significantly. Finally, SMMs based on `Histograms` produce consistently more accurate (by several percent) phase detection results than those based on `Gaussian` probability distributions because they are less sensitive to noise such as outliers. We observe similar trends for cluster and transition isolation.

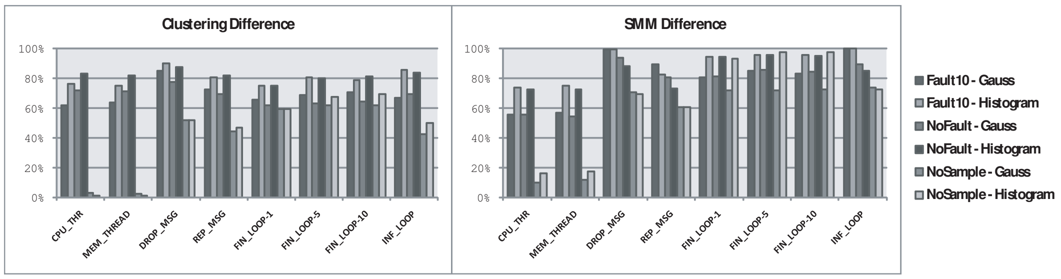


Fig. 3.6.: Average faulty phase detection accuracy

Figure 3.7 shows faulty phase detection accuracy on a per-application basis, focusing on SMMs that use `Histogram`. The data shows that detection accuracy depends strongly on the application. Further, the SMM-based metric has poorer accuracy with `CPU_THR` and `MEM_THR` primarily due to its poor results on MG, BT and SP, while it provides higher accuracy for FT than does the cluster-based metric. The SMM-based metric is more accurate for other faults because it performs more consistently across the applications. Finally, sample runs are essential for the cluster-based metric while the SMM-based metric still provides reasonable accuracy on the other fault types without them.

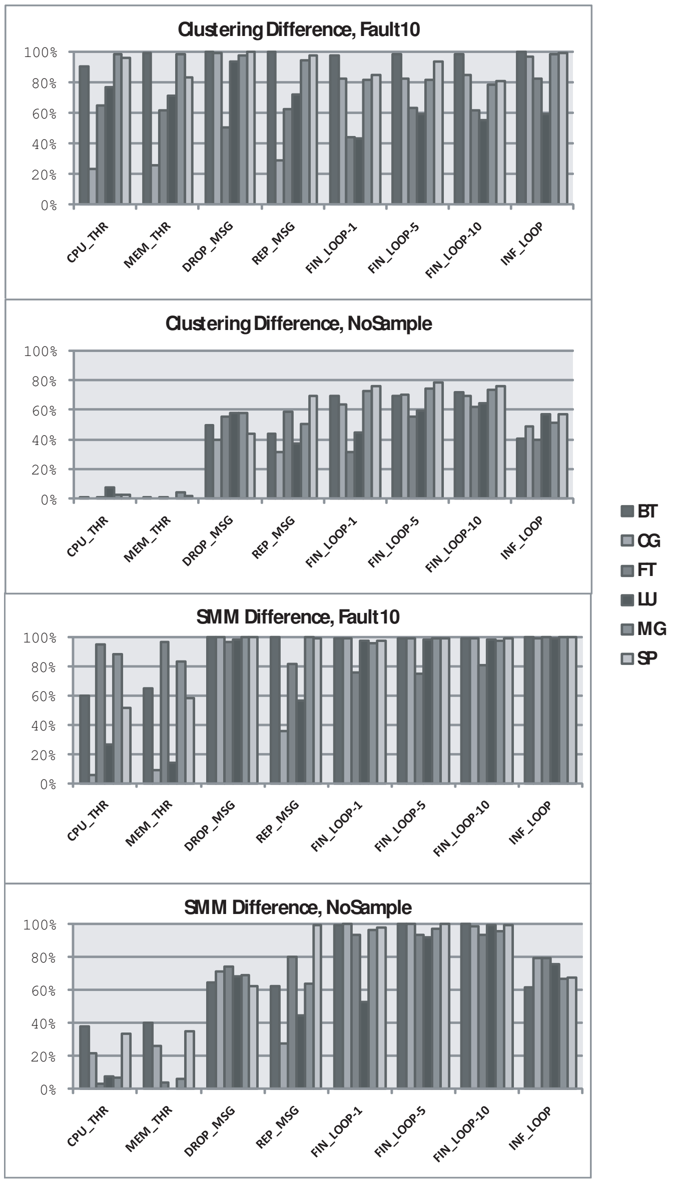


Fig. 3.7.: Faulty phase accuracy per application

Cluster Isolation

Once *AutomaDeD* identifies the faulty phase, its cluster isolation can help locate the root cause of the bug by showing the cluster that contains the task where the bug was injected. *AutomaDeD* again uses the SMM-based and cluster-based metrics to perform cluster isolation. Alternatively, it can examine the individual transitions within the faulty phase to identify those that are unlikely given the probability distribution on the transition. Our evaluation measures the accuracy of *AutomaDeD*'s cluster isolation separately from that of its faulty phase detection by always applying the techniques to the faulty phase, that is we assume the phase detection was accurate.

Figure 3.8 shows the accuracy of *AutomaDeD*'s cluster isolation on a per-application basis, focusing on SMMs that use `Histogram`. Cluster isolation using the cluster-based metric has poor accuracy for nearly all applications and fault types. The other options produce significantly better results. The abnormal transition method without sample runs and the SMM-based metric with sample runs provide the best accuracy for `CPU_THR` and `MEM_THR`, with near perfect results on half the applications. The abnormal transition method achieves high accuracy for the `FIN_LOOP` and `INF_LOOP` using sample runs.

In general, the accuracy of cluster isolation varies widely across the applications for the same fault type since the faults can propagate themselves quickly from one task to another. Thus, some task(s) other than the faulty task may exhibit behavior the most divergent from its normal activity, which can cause the cluster-based and SMM-based metrics to mis-identify them as the source of the fault. While task behavior does not confuse the abnormal transition method, it can perform poorly due to the relatively coarse granularity of SMM transitions. As such, a fault may propagate from a transition with a later starting timestamp to one that began earlier, causing the wrong transition to be identified as the fault's first manifestation. We could reduce this effect by breaking long states into smaller ones, which will improve their precision.

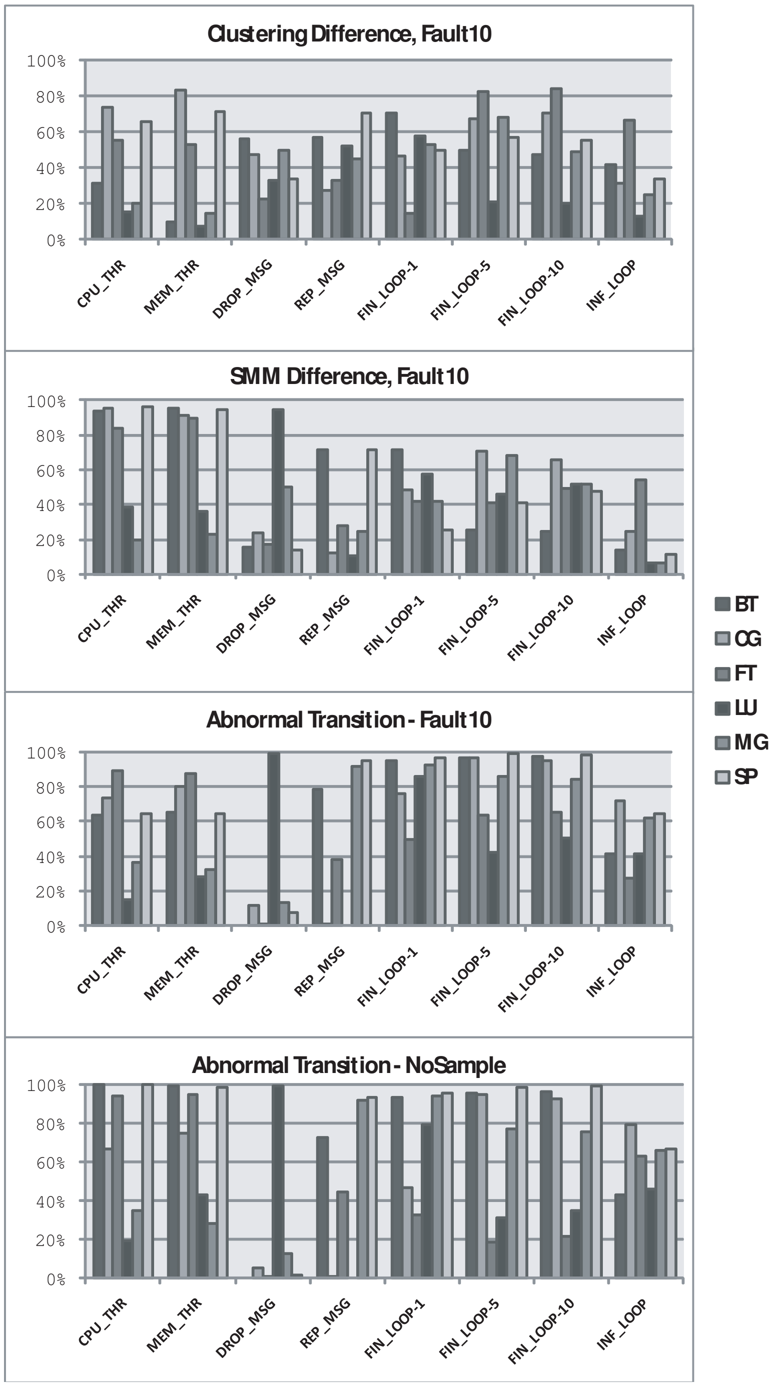


Fig. 3.8.: Cluster isolation accuracy per application

Figure 3.9 shows the percentage of runs (using the `Fault10` sample run configuration) in which the faulty task cluster consists of only one faulty task. Precisely identifying the faulty makes significantly easier for developer to identify the bug since narrows it down to a single task's control and data flow. *AutomaDeD* fully isolates the faulty task in more than 90% of the cases for `CPU_THR`, `MEM_THR`, `DROP_MESG` and `REP_MESG` and 70% for `FIN_LOOP` and `INF_LOOP`. In contrast to prior results, using `Gaussian` distributions for the times preceding transitions provides greater accuracy because they are more sensitive to outliers, which suggests that both probability distributions should be used in practice.

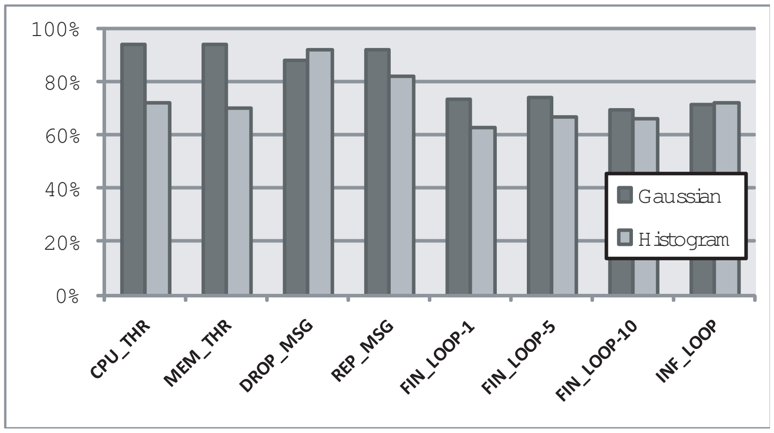


Fig. 3.9.: Isolation of a singleton cluster

Transition Isolation

AutomaDeD uses two algorithms for transition isolation. First, it compares the SMMs of the faulty cluster to those of other clusters and selects the transitions most responsible for the differences. Alternatively, it selects the earliest abnormal transition within the faulty cluster. Since our goal is to focus debugging efforts, we consider how frequently the faulty transition is the top choice or one of the top five choices of these methods. Figure 3.10 shows the results, with the clustering-based algorithm on the left and the transition-based algorithm on the right.

The clustering-based algorithm consistently ($\geq$ 90% of the time) includes the faulty transition in its top five choices for `FIN_LOOP` and `INF_LOOP`. The transition-based algorithm is less consistent across applications but when it succeeds, it usually does so with its first selection. Both methods exhibit low accuracy for `DROP_MESG` and `REP_MESG` faults because their effects manifest long after the fault is injected. They also perform relatively poorly with `CPU_THR` and `MEM_THR` because these faults cause sudden behavioral changes that resemble ordinary outlier transitions.

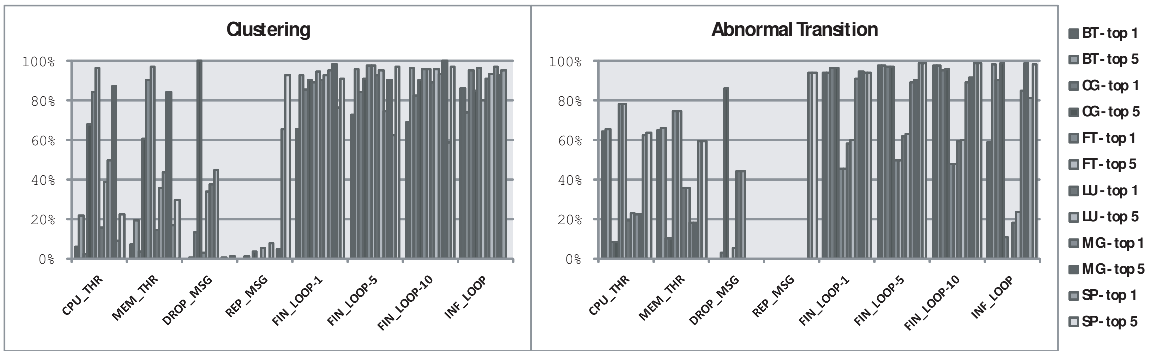


Fig. 3.10.: Transition isolation accuracy per application

3.5.3 Case Study: MVAPICH Bug

We illustrate the utility of *AutomaDeD* via a case study of applying it to a real bug in the MVAPICH-0.9.9 MPI implementation [54]. The bug occurs in its MPI task launcher, mpirun, which sometimes fails to clean up after an application, leaving processes to run concurrently with subsequent jobs. We evaluated *AutomaDeD* on this bug, which is similar in effect to our CPU- and Memory-intensive thread faults, by executing a 16- or 64-task run of as the application being debugged while simultaneously executing a 16-task run of either LU, MG or SP on the same set of nodes as the previous runaway tasks). These experiments cover the cases where runaway tasks interfere with either all or a subset of the application's tasks.

We provided *AutomaDeD* with a set of five sample runs of BT with no interference. Figure 3.11 presents average the SMM-based metric that *AutomaDeD* determines for each phase of three runs where BT ran concurrently with either LU, MG and SP (one set for 16-task and another for 64-task BT runs) as well as the average score for the five no-interference runs. The sets of sample runs used to compute each no-interference run's deviation scores excluded the run itself. The deviation scores of all no-interference phases were consistently low. In contrast, the scores of the initial phases of the three interference runs show high deviation scores, identifying the exact region of time when the shorter runs of LU, MG and SP overlapped with the execution of BT. Further, *AutomaDeD* clearly shows that the interference run of MG in one 16-task experiment began after the first phase of BT, since the deviation score starts at the baseline level, rises for three phases and then drops to the baseline.

AutomaDeD significantly aids debugging. First, it clearly identifies the performance anomaly, which might not have been noticed for a long time or blamed on extraneous factors such as network load or choice of input. Second, *AutomaDeD* determines when the interference occurs, which facilitates detection of the interference tasks from system logs or other methods. Although *AutomaDeD* can often identify the tasks most affected by the fault, it did not isolate those tasks in this case since BT is tightly coupled, which leads to the interference tasks impacting all of BT's tasks even with 64-task runs.

3.6 Discussion

Large-scale application debugging is very challenging because of the vast amount of information developers must consider to identify a bug's root cause. *AutomaDeD* focuses debugging efforts on the time period, tasks and code region where the bug is first manifested. Thus, it significantly improves developer debugging productivity by

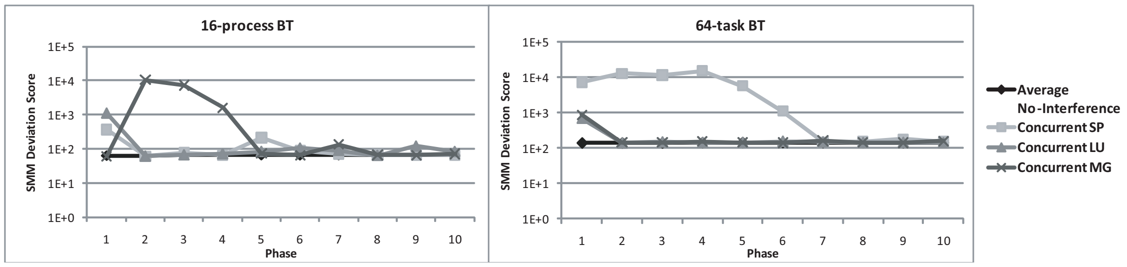


Fig. 3.11.: Phase deviation scores of MVAPICH bug use-case

reducing the amount of information that must be considered even as the application is scaled to large task counts. This work describes the fundamental approach and design of *AutomaDeD* and establishes it as a valuable addition to the developer's toolkit. Our results demonstrate that *AutomaDeD* is very accurate for key debugging tasks. In particular, it correctly identifies the faulty phase in 90% of our trials for delays, hangs and message faults and in 70% of our trials for interference faults. Given the faulty phase, *AutomaDeD*'s accurately identifies a small task set (often a single task) in which the bug occurred for over 80% of delays and hangs, over 40% for message faults and over 70% for interference faults. Given the faulty cluster, *AutomaDeD* identifies the error site with 90% accuracy for delays and hangs and 50% accuracy for interference faults.

4. SCALABLE ERROR DETECTION IN SCIENTIFIC APPLICATIONS

In this chapter we present novel techniques to perform error-detection in HPC applications in a scalable manner. We extent our previous work by using scalable clustering and sampling-based nearest-neighbor methods to isolate abnormal tasks. We also present a scalable graph compression technique that reduces the dimensionality of the problem—by reducing the number of edges in the SMM graph that is built up for each task—which increases accuracy of isolating an abnormal task. We present an evaluation of the techniques with thousands of processes in a Linux cluster at the Lawrence Livermore National Laboratory. The technique is shown to isolate faults in less than 5 seconds.

4.1 Introduction

As today's High Performance Computing (HPC) applications increase in complexity, debugging errors, performance anomalies, and unexpected behavior in these applications becomes excessively difficult. A single bug in an HPC application often affects multiple processes, so growing application scales leads to an effect on many processes. Most existing debugging techniques as implemented in tools like gdb [1], TotalView [2], or DDT [3] do not automate the debugging process: developers must manually locate errors and backtrack through interactions across processes to locate the root cause. Clearly, this approach is infeasible for large-scale parallel applications.

We previously presented *AutomaDeD* in Chapter 3, a tool that detects errors based on runtime information of control paths that the parallel application follows and the times spent in each control block. *AutomaDeD* suggests possible root causes of detected errors by pinpointing, in a probabilistic rank-ordered manner, the erroneous process and the code region in which the error arose. Intuitively, the erroneous tasks often form a small minority of the full set of tasks. Hence, they are outliers when we cluster the tasks, based on their features related to control flow and timing. Further, in the time dimension, the executions in the first few iterations are more likely to be correct than in later iterations, which we also leverage to determine correct or erroneous labels.

Almost all existing parallel debugging tools fail to scale to the process counts of today's state-of-the-art systems. For example, at the time of writing this dissertation, the largest number of processes in which TotalView has been tested is 786,432 [55]—a test performed at the Lawrence Livermore National Laboratory's Sequoia supercomputer with 1.5 million cores available. Although some tools, such as TotalView, are able to reach hundreds of thousands of processes today (and possibly a million processes soon), major designed changes are expected before they are able to run with the amount of parallelism of exascale supercomputers, i.e., in the order of hundred million processes or more. Three main factors impede scalability. First, the tools include a centralized component that performs the data analysis. Thus, tools must stream behavioral information from all the processes to this central component so that it can process the information to determine the error and, possibly, its location. Second, the tools require huge amounts of data. While many tools optimize the monitoring part quite well, the cost of shipping all information to the analysis engine and the cost of analyzing the full volume of data remains. While tools such as STAT [18] reduce the data volume that the central component must handle, they still must process the full data in their communication structure (in parallel). Third, the data structures used to maintain the information are not completely optimized for the operations that need to be performed for error detection and localization, such as comparison of information from processes that belong to the same equivalence class. Small differences in the cost of one operation, though insignificant for hundreds of processes, become significant at larger scales.

Due to scaling limitations, many existing techniques [14, 15, 56] collect information at runtime and perform analysis offline, decoupled from the main application execution. This approach may allow bug diagnosis only after a long execution in the erroneous mode. This reduces application throughput and wastes computational resources.

We aim to change the debugging scenario fundamentally. In this section we propose techniques to perform error detection and diagnosis online. In this work, we introduce novel scalable mechanisms that allow *AutomaDeD* to execute online in large-scale systems. To achieve this goal, we introduce three major design and implementation innovations:

Efficient edge comparison allows *AutomaDeD* to compare per-process graph elements efficiently. Following the model of our baseline implementation, *AutomaDeD* represents each process as a graph in which nodes are MPI calls or computation blocks between such calls. Edges in the graph have a transition probability and a time distribution. We compare edges by representing state information in the graphs using pointers instead of character strings, so that *AutomaDeD* can perform state comparisons with a few machine instructions (distinct from the baseline, which used character strings to represent application states). We perform optimal comparison of per-edge probability distributions using a lookup table instead of computing an integral.

Graph compression reduces the time to compare the behaviors of processes by merging edge chains, since the difference between two graphs is the sum of the differences between their edges. This mechanism reduces data dimensionality so that *AutomaDeD* can focus first on finding outliers (i.e, abnormal processes) using fewer dimensions, and later focus on the dimensions that a fault most affects (e.g., by using just that graph region). Our compression preserves the basic control structure of the program so that the analysis provides actionable results.

Scalable outlier detection uses distributed sampling techniques to find the erroneous processes among many parallel ones with low overhead. We use scalable clustering [57] and a novel nearest neighbor technique to find outliers efficiently.

Careful integration of the three techniques eliminates any centralized element in our solution (except for the computer where the output is presented to the user), thus making *AutomaDeD* scalable to larger and larger system sizes. This benefit requires that we must carefully sample *AutomaDeD*'s data so our input is representative of all processes and we do not miss sharp discontinuities in process behavior.

Our experiments on a Linux cluster with thousands of cores show that *AutomaDeD* scales to thousands of processes and that it can isolate erroneous tasks in a few seconds. We demonstrate its error-detection capabilities through fault injections in the NAS Parallel Benchmarks and its scalability on up to 5,000 AMG2006 [58] processes. *AutomaDeD* performs the entire error-detection analysis (i.e., abnormal process and erroneous code region isolation) in under 5 seconds.

4.2 Redesign of ***AutomaDeD***

In this section, we detail the design of the three novel aspects of *AutomaDeD*, which improves on the operational flow of the baseline that we have just described. In the improvements, we first show how edges in two graphs corresponding to two different tasks can be efficiently compared. Next, we show how the graphs corresponding to the SMMs can be compressed to improve the accuracy of identifying erroneous tasks and to reduce the computational cost of this process. Finally, we describe how identification of abnormal tasks is done in a scalable manner through two alternate methods of clustering and nearest-neighbor calculation.

4.2.1 Efficient Edge Comparison

In the error detection part, *AutomaDeD* performs pair-wise comparisons of SMMs. In Chapter 3 we presented a formula to compute the dissimilarity (or difference) between a pair of SMMs. Its main idea was to find and add up the differences

between corresponding edges in the two SMMs (that are being compared). An edge ($state_i$, $state_j$) that is present in two different SMMs, SMM_1 and SMM_2, implies that both tasks performed a state transition of the form $state_i \rightarrow state_j$ at some point in the program execution and we can compute edge differences. If an edge is present in one SMM but is not present in the other, the difference is assigned a high weight to highlight control flow differences that cause this behavior. Computing edge differences requires two steps to be computed efficiently:
(i) *Finding matching edges in two SMMs.* In order to compare an edge from an SMM, we must first determine whether a corresponding edge exists in the other SMM.
(ii) *Computing differences of edge attributes.* Once we have found corresponding edges, we compute the difference between their attributes, i.e., the differences between their transition probabilities and time probability distributions.

We efficiently perform the first step by representing SMMs using data structures that allow efficient edge searching. We use a sorted map of unique keys (a C++ map) to represent SMMs. The keys are edges and mapped data are edge attributes. This structure supports edge lookups with complexity O(log n), where n is the number of edges.

We represent SMM states by call stack *paths* collected when the program calls MPI routines. A call stack path is the list of function calls that are currently active, which includes the called functions and the offset into the functions. In previous versions of *AutomaDeD*, we used character strings to represent paths, which incurred a large overhead when we compared two states. Our current implementation supports a compact representation that uses unique references and, thus, supports direct comparison of the references. Further, since references from different tasks may point to the same path, we exchange the map between the two tasks to determine a consistent view of the paths before we compare their SMMs. This permits comparisons of edges through a few machine instructions (by reference comparisons) instead of by comparing character strings.

The second step is the edge-comparison computation. For this task we must calculate the difference between two time distributions—calculating the difference of the transition probabilities is trivial since it only involves a subtraction of two double-type values. We use the Lk-norm method (Chapter 3) with $k = 2$ to compute the difference of two probability distributions, which is based on the following formula:

$$\int_{-\infty}^{\infty} |P(x) - Q(x)|^k dx, \tag{4.1}$$

where $P(x)$ and $Q(x)$ are two continuous probability distributions of the random variable x (representing time) in the two edges. We estimate probability distributions with parametric and non-parametric methods. *AutomaDeD* uses normal distributions and histograms as the base models respectively. While histograms provide a better fit for the observed data than normal distributions, they have significantly higher memory and computational complexity. We therefore in the following use a normal distribution since we emphasize low overhead at scale.

Previous work in statistics [59] has shown that the overlap percentage of any two normal distributions can be estimated from their parameters, i.e., mean and standard deviation. A table (or nomogram) as shown in Figure 4.1 can be computed *a priori* so that we can obtain the overlap of two new distributions by inspecting the table, where the y-axis is the ratio of the largest standard deviation $SD2$ to the smallest standard deviation $SD1$, and the x-axis is the distance between the means $M1$ and $M2$ normalized by $SD1$. The key observation is that the Lk-norm of two normal distributions equals the area that does not overlap between the distributions. Therefore, *AutomaDeD* uses a similar table to estimate the value of an Lk-norm calculation without incurring in the high overhead of numerically estimating the integral in equation (4.1). The points in the table that *AutomaDeD* uses are calculated using the Lk-norm formula with parameters of two (randomly selected) normal distributions. Since we must quantize the parameters' values for which we store the results in the table, we only build a table of 500×500 values and approximate overlapping percents by interpolation. Our experiments show that a table of this size is sufficiently accurate to distinguish two normal distributions.

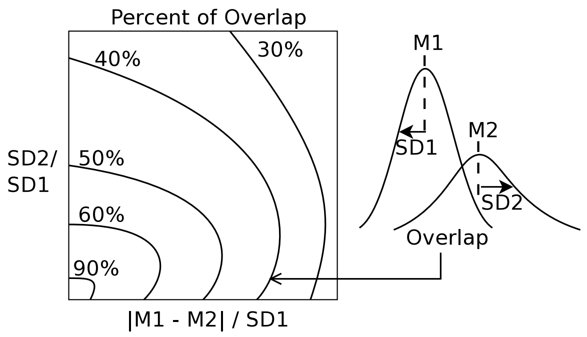


Fig. 4.1.: Percent overlap of two normal distributions.

4.2.2 Graph Compression

Motivation. Parallel programs with complex control flow result in SMM graphs with many edges. For example, in our experiments with the NAS Parallel benchmarks, graph sizes are often on the order of hundreds of edges. Large edge counts impact the accuracy of isolation of abnormal tasks in *AutomaDeD* since the problem directly corresponds to the problem of detecting outliers (i.e., abnormal tasks) in a high dimensional space. High dimensional mathematical spaces create difficulties for unsupervised machine learning techniques such as clustering and k-nearest-neighbor due to the *curse of dimensionality* [60]. Distances between all pairs of points in high dimensional data tend to become almost equal—with too many dimensions, deviations from normality in a few dimensions are not as significant. Thus, *AutomaDeD* cannot find the abnormal task. An additional problem associated with SMM graphs of large sizes is that the overhead of the task isolation phase increases because the complexity of distance calculations of SMM pairs is proportional to the number of edges. Therefore, we implement an algorithm that allows *AutomaDeD* to compress large SMMs before we perform task isolation.

Which edges can be compressed. We observe that we can merge a linear chain of states and still retain the general control flow structure of the program, i.e., the states and edges that represent the main program loops are maintained in the

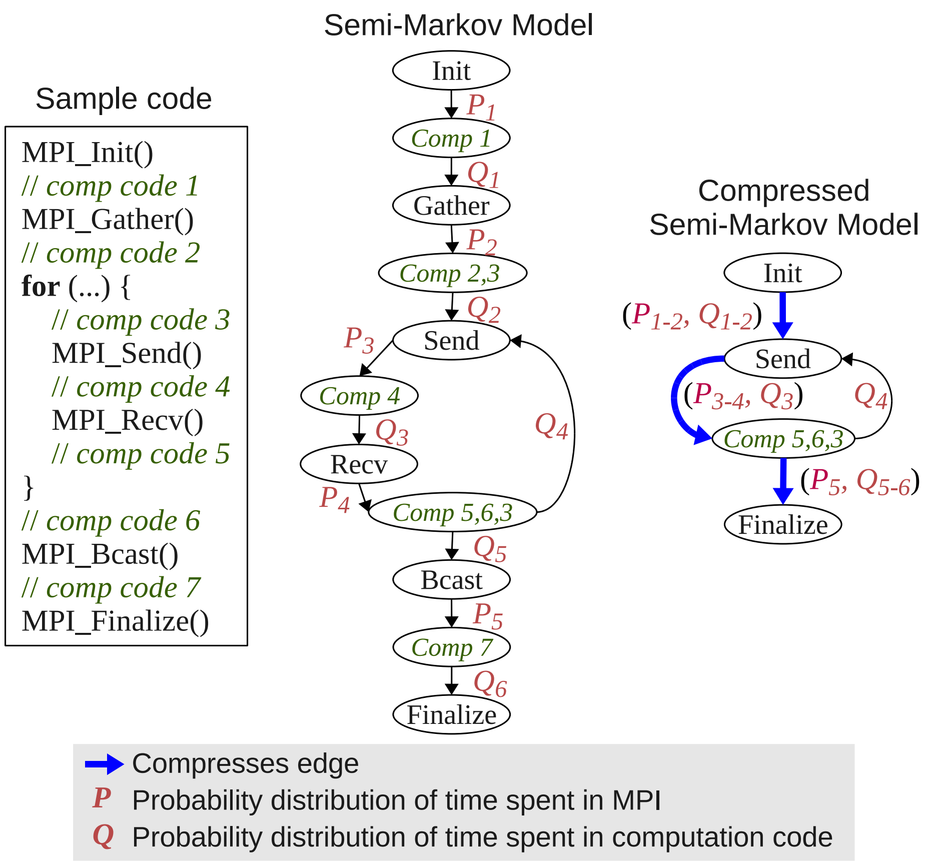


Fig. 4.2.: Compression approach.

compressed graph. Figure 4.2 illustrates this idea. The SMM represents the sample MPI code in the left part of the figure. We omit the transition probabilities associated with the edges (and only present the time distributions) for simplicity. The right part of the figure shows the compressed SMM after we apply our compression algorithm to the original SMM. We define a *sequence* of states as a linear chain of states with out-degree of one, in which the transition probabilities associated with their outgoing edges are 1.0. The compression algorithm merges sequences of states keeping the main control flow structure in the resulting compressed graph. The sequence of states after `Init` up until `Send` are summarized as only `Send`, and their edges are all merged into a single *compressed* edge.

A compressed edge contains two distributions P and Q that represent the time spent in the MPI calls and in the computation blocks of the original graph. Keeping time distributions separate in the compressed edge helps developers differentiate the parts of the code that are affected by MPI operations from parts in which the computation code in between them is the source of the problem.

How to assign attributes to compressed edges. We merge the time distributions in sequences of edges under the assumption that the underlying random variables are independent, and that normal distributions are used to fit the observed data. Given two independent random variables T_1 and T_2 that are normally distributed with parameters μ_1, σ_1 and μ_2, σ_2, the new random variable $T_1 + T_2$ is also normally distributed with parameters $(\mu_1 + \mu_2)$, $(\sigma_1 + \sigma_2)$. Therefore, when compressing a sequence of edges, the compression algorithm simply sums the parameters of the distributions in those edges, keeping distributions for communication (i.e., for MPI routines) and for computation blocks separate. As can be observed in Figure 4.2, a compressed edge has a tuple (P, Q), where P is the added distributions of communication states and Q is the added distributions of computation states.

Distributed nature of the merge process. The compression algorithm involves two steps; we now discuss our completely distributed implementation of both steps. First, *AutomaDeD* must determine the set of edges that are present in all the tasks or, if we have determined task equivalence classes, the set present within all tasks within an equivalence class. Our edge compression algorithm only targets such edges for compression. *AutomaDeD* does not compress edges that are not common in all SMMs to avoid eliminating abnormal transitions that may be present in only a few tasks. Second, the compression happens locally and concurrently at each task using the above set of edges.

Figure 4.3 illustrates this idea. We define an edge's *support* as the number of tasks in which it appears. We first apply a reduction operation that collects local information from all tasks and applies an aggregation operation on that information. The operation that the reduction performs sums the edge support of local graphs.

At the end of the reduction, the edges of states 1–4 have a support of four (because they are present in all graphs), while edges of states 4–7 only have support of three. After we perform the reduction, the root process (i.e., the one that initiates the reduction) broadcasts the reduced graph to all tasks so that every task has the set of edges to compress, i.e., those with support of four for this example. We implement the reduction with a binomial tree, which has logarithmic complexity in terms of the number of tasks. We cannot use `MPI_Allreduce` since the tasks can contribute different numbers of edges.

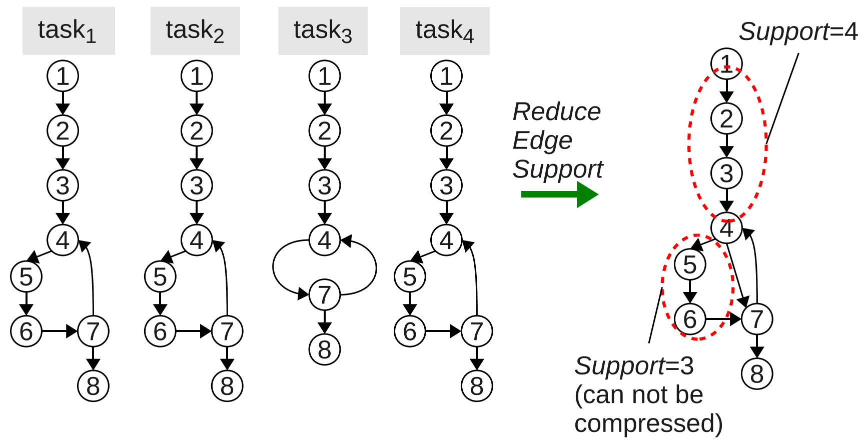


Fig. 4.3.: Global reduction of edges support.

The second step compresses the local graph for edges that are globally supported. We use a modified version of the depth-first-search algorithm to traverse the graph. The algorithm's main idea is that sequences of states can be merged until the beginning or the end of a loop is found. The algorithm assumes an adjacency-list representation so it can find state neighbors for each state as the graph is traversed.

Figure 4.4 shows the compression algorithm's pseudocode. We define a *loop-head* state as the first state in a loop, and *loop-tail* state as the last state in a loop. As we traverse the graph, we store edges in a queue until the `mergeEdgesInQueue()` function is called in loop-heads, loop-tails or in the last state. This function merges edges as described previously in Figure 4.2 keeping probability distributions for communication

```
DFSCompress(State state) {
  if (isNotFinalState(state)) {
    neighbors = getNeighbors(state)
    for each n in neighbors {
      Edge edge(state, n)
      if (edgeHasNotBeenVisited(edge)) {
        addEdgetoQueue(edge)
        if (isHeadOrTail(n)) {
          mergeEdgesInQueue()
          DFSCompress(n)
        }
      }
    }
  } else {
    mergeEdgesInQueue()
  }
}
```

Fig. 4.4.: Depth-first-search compression algorithm.

and computation code regions separate. The complexity of the algorithm is O(*number of edges*).

Need for iterative drill-down due to graph compression. Due to graph compression, when *AutomaDeD* initially provides the characteristic edge(s) that likely caused the task to become anomalous, the granularity can be more coarse than in the baseline. The granularity can be a compressed edge, which includes multiple edges from the original graph. However, *AutomaDeD* keeps the original graph and the compressed graph in memory. For the task that we determine is anomalous, we perform edge isolation locally using the fragment of the original graph that corresponds to the part of the compressed graph that we determined is anomalous. For example, if edges $\{e_1, e_2, e_3\}$ were compressed into an edge $e^{(c)}$ and the initial iteration of the edge isolation flagged $e^{(c)}$ as the anomalous edge, the next iteration can work on $\{e_1, e_2, e_3\}$ and diagnose at the same granularity as baseline. We preserve the advantage of edge compression—*AutomaDeD* does not perform any communication of the potentially large original graph (except for the reduction to infer edges' support) to other processes and does not have to perform task isolation on the original graph. The minor disadvantage is that edge isolation requires two iterations.

4.2.3 Scalable Outlier Detection

The typical use case of *AutomaDeD* isolates abnormal tasks in the time period the application fails. This can be challenging since *AutomaDeD* must extract a few abnormal tasks from many normal tasks. Naive techniques such as comparing each task against each other to find the most dissimilar task do not scale well since the complexity of these methods is quadratic with respect to the number of tasks.

We implement two scalable approaches to isolate abnormal tasks: (1) *clustering*, using CAPEK's algorithm, which first finds clusters and then determines abnormal tasks as indicated by the largest distances from their cluster centers; and (2) *nearest-neighbor*, which determines abnormal tasks based on the largest distances from their nearest neighbors. In both approaches, we sample a constant number of data points (i.e., tasks) and perform the analysis treating the sample set as representative of the entire set of points. Thus, we avoid having an unmanageable linear algorithmic complexity with respect to the number of points; both approaches scale with a complexity of the log of the number of tasks. Figure 4.5 illustrates the idea behind the two algorithms, which the next sections describe in detail.

Clustering

We have developed a novel outlier-detection technique based on CAPEK, a scalable clustering algorithm designed for large-scale, distributed data sets like those

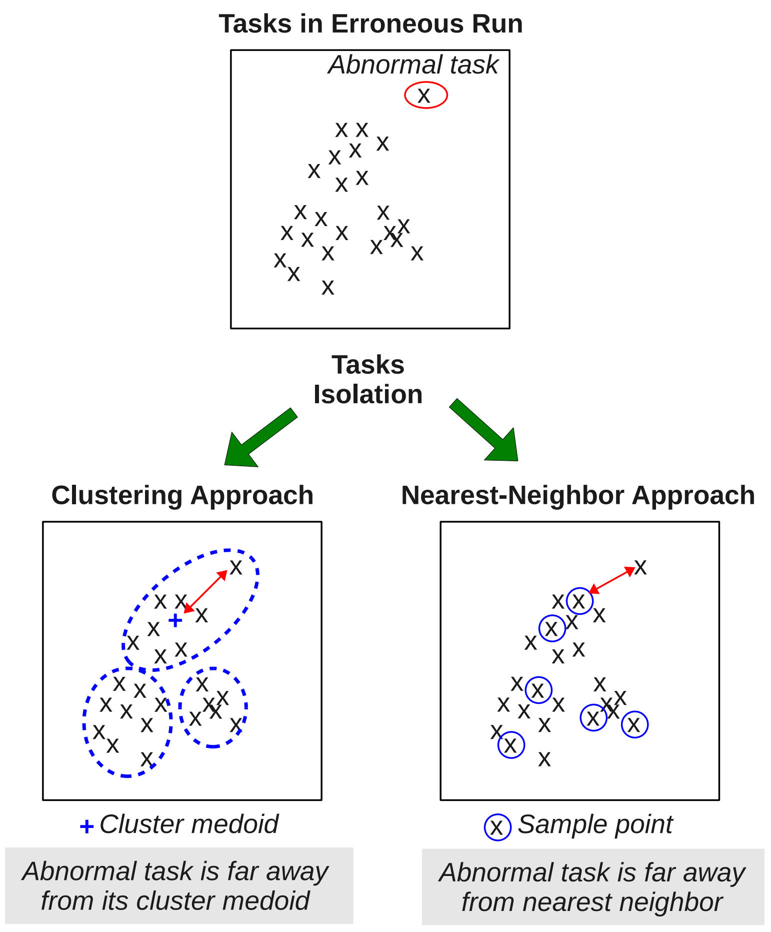

Tasks in Erroneous Run
Abnormal task
Tasks Isolation
Clustering Approach
Nearest-Neighbor Approach
Cluster medoid
Sample point
Abnormal task is far away from its cluster medoid
Abnormal task is far away from nearest neighbor

Fig. 4.5.: Clustering and Nearest-Neighbor methods to isolate abnormal tasks.

generated by parallel performance tools [57]. CAPEK finds groups, or *clusters* within distributed data sets, which gives us information about the structure of our data. For each cluster, CAPEK also determines a representative, or *medoid* m_i, and we can find outliers by finding the objects (SMM) in the data set that are furthest from their representative medoids.

Formally, CAPEK is a *K-Medoids* method. K-Medoids methods take a set of objects X, a dissimilarity function $d : X \times X \to \mathbb{R}$ and a number of clusters $k \in \mathbb{N}$ as input. They produce a *clustering*, a set of disjoint clusters $C = C_1, \ldots, C_k \subseteq X$ such that $\bigcup_{i=1}^{k} C_i = X$ and a set of medoids $M = m_1, \ldots, m_k$ such that $m_i \in C_i$. Each m_i is the representative element for cluster C_i. These methods attempt to minimize $\sum_{i=1}^{k}\sum_{x_j \in C_i} d(x_j, m_i)$, the total distance from each object to its representative medoid. The basic version of CAPEK requires the user to specify the number of clusters, k. It also allows the user to search for an ideal k using the Bayesian Information Criterion (BIC) [61]. The intuition behind this algorithm is that the medoids, m_i, will be approximately centered within their clusters, and they are thus good representatives for the clusters as a whole.

CAPEK has several advantages that make it well-suited for clustering SMM data. First, CAPEK is sampled, from which it derives its massive scalability. Traditional sequential clustering algorithms have quadratic or linear runtime, but CAPEK uses all processors for analysis to achieve logarithmic runtime, which makes our analysis feasible at scale. Other algorithms, such as the hierarchical clustering that baseline uses 3, do not readily support sampling, and thus do not scale to the system sizes that CAPEK supports.

Second, unlike *K-Means* methods [62–64], K-Medoids does not require that we can define algebraic operations, such as addition and scalar division, on the data. K-Means methods discover the synthetic means, or *centroids*, of their clusters using these operations, but we cannot directly calculate a "mean" SMM.

Finally, K-Medoids methods produce flat partitions of the data, which simplifies outlier detection. With hierarchical clustering, for example, we must choose the level in a clustering tree to describe clusters and outliers best. However, this complicated process does not scale. With a flat partition, we can detect outliers using standard deviation, which is straightforward and fast to compute. Our clustering-based outlier detection algorithm is:

(1) Perform clustering: We obtain a clustering C using CAPEK, which provides copies of the medoids m_i for all clusters to each process.

(2) Find distances from each task to its medoid: Each task computes the distance d_{ji} from its local SMM x_j to its representative, m_i. As CAPEK guarantees, this medoid will be the representative nearest to x_j.

(3) Normalize distances using standard deviation: Using parallel reductions

within each cluster, we compute the standard deviation σ_i for each cluster in logarithmic time. We then normalize each process's d_{ji} to obtain $d'_{ji} = d_{ji}/\sigma_i$, which allows us to find outliers in data sets that may exhibit several different "normal" behaviors.
(4) Find top-k outliers: d'_{ji} is a measure of how far each SMM is from its representative; we now find the top k values of d'_{ji} in parallel. The corresponding SMMs are the "most different" from their representatives. We gather these SMMs, which we report as outliers. We can trivially find the top k SMMs in logarithmic time by computing k parallel reductions in sequence.

Nearest Neighbor

Our nearest-neighbor (NN) method classifies outlier tasks based on dissimilarities between tasks and their nearest neighbors. The main idea is that an abnormal task will be far from its nearest neighbor, while normal tasks will be close to each other so their pair-wise NN distances are small. To make NN scalable, we only perform NN distance calculations against a constant number of sample tasks, rather than against all tasks. When we calculate NN distances, a sample task removes itself from the sample tasks to avoid picking itself as its nearest neighbor. Our NN outlier detection algorithm is:
(1) Sampling: We locally generate a set of random indices that represent task ranks using a deterministic pseudorandom number generator with the same seed. After this step, each process can determine if it is a sample task.
(2) Broadcasting of samples: Each sample task broadcasts its SMM. After this step, each task can compare its SMM to the sample SMMs in order to find its NN distance.
(3) Find NN distance: Each task compare its SMM to those of the sample tasks. The SMM of the smallest distance corresponds to the NN task. Finally, we perform a global reduction at the root task to find the k most abnormal tasks.

NN may not isolate abnormal tasks if a fault affects multiple tasks simultaneously. To illustrate this problem, suppose that the normal clustering of the tasks in an application is two clusters, but a fault creates a third cluster with a few similar abnormal tasks. Suppose also that we sample two tasks t_1 and t_2 from this abnormal cluster. We will determine that t_1 and t_2 are each other's nearest neighbor. The distance values for each case will be low values and these tasks will not appear in the top-k rank of outliers (or abnormal tasks), which will prevent *AutomaDeD* from isolating them. The clustering approach avoids this problem because the abnormal tasks will belong to one of the normal clusters (because of the BIC methodology to select cluster configurations) and have high distances to the cluster medoid.

Abnormal Edge Isolation

Our previous work [65] presented mechanisms to detect the code region in which a fault is first manifested after task isolation. The *characteristic transition* is the edge that contributes most to the distance of the abnormal task. We easily extend this concept to multiple rank-ordered transitions ordered by their contributions to the difference. In this work we implement a modified version of one of these techniques. We perform the two outlier detection methods as follows:
(1) *Clustering*: After clustering, each task has the medoid of its cluster. We compare the abnormal task's SMM to the medoid's SMM and sort the edges in ascending order of their dissimilarities. We flag the top-k edges as abnormal.
(2) *NN*: After the task isolation phase, we compare the abnormal task SMM to all sample SMMs. As in the clustering method, we sort edges by their dissimilarities and flag the top-k edges as abnormal.

After we perform graph compression on the SMM (and have isolated the abnormal task), *AutomaDeD* keeps a copy of the original SMM in memory and performs edge isolation using that SMM. Since we must compare edges between graphs of the same nature—always between uncompressed graphs for this case—the abnormal task must have the original SMM of the other tasks. We fulfill this requirement by sending the original graph from the compared task(s) to the abnormal task. This additional step incurs a small overhead; for example, for the clustering method we only send one graph (from the medoid task) to the abnormal task.

4.3 Experiments and Results

4.3.1 Fault Injection

We empirically evaluate the effectiveness of *AutomaDeD*'s techniques by injecting faults that commonly occur in parallel applications. We inject faults into six applications of the NAS Parallel benchmark suite: BT, SP, CG, FT, LU and MG [48]. We omit EP because it performs almost no MPI communication and IS because it uses MPI only in a few code locations. Since their MPI profiles produce small SMMs, monitoring at the granularity of MPI calls does not suit these applications. Our injector [65] uses P^{N}MPI to inject a wide range of software faults into random MPI calls during MPI application runs. We focus our fault injection campaign on the following performance faults:

• `CPU_INTENSIVE`: CPU-intensive code region, emulated by a triply nested loop.

• `MEM_INTENSIVE`: Memory-intensive code region, emulated by filling a 1GB buffer with data at random locations.

• `HANG`: Local deadlock, emulated by making a process indefinitely suspend execution.

• `TRANS_STALL`: Transient stall, emulated by making a process suspend execution for 5 seconds.

In these experiments, we ran the benchmarks at a moderate scale: 512 processes for CG, FT, LU and MG, and 529 processes for BT and SP, with input size B. We use six-core nodes (the LLNL Sierra cluster), with 2.8 GHz Intel Xeon processors, 24 GB of RAM per node and InfiniBand interconnect. Each experiment injects a single fault into a single random task during MPI communication operations (e.g., blocking and non-blocking sends and receives, all-to-all, broadcasts and barriers). For each benchmark, fault type, and detection technique, we perform 10 runs, for a total of 960 experiments.

4.3.2 Fault Injection Results

When we inject a fault, *AutomaDeD* performs the error-detection analysis by default at the end of the run during `MPI_Finalize`. For some benchmarks, the SMM created at the end of the run can be quite large, even after compression. For example, in LU and MG, the compression algorithm can only compress the graph to around 120 edges (from originally around 250 edges), which is still a large number of edges for the outlier detection techniques to work accurately. In these cases, *AutomaDeD* divides the run into user-defined phases to reduce the size of per-phase SMMs to a manageable size and performs the analysis in the faulty phase. In Section 3.4.2, we presented a technique to detect the abnormal phase in a run. For these experiments, we assume that *AutomaDeD* is provided with the faulty phase, which the user can pinpoint or our prior algorithm can detect. In cases where a fault causes the application to suspend execution and it does not allow the creation of a new SMM at the end of a phase or during `MPI_Finalize`, such as in the `HANG` fault, the analysis is executed when *AutomaDeD* does not observe any state transitions for a "long" period of time; a parameter that can be configured by the user or estimated by *AutomaDeD* from previous runs (by taking the maximum transition time). In our experiments, we use 60 seconds for this parameter.

We use two metrics to evaluate error-detection and localization quality: *task-isolation recall*—the fraction of runs in which the task in which we inject the fault is in the top-5 abnormal processes that the task-isolation method (separately clustering or NN) outputs; *edge-isolation recall*—for cases in which *AutomaDeD* correctly isolates the abnormal processes, the fraction of runs in which the code region in which we inject the fault is in the top-5 abnormal edges.

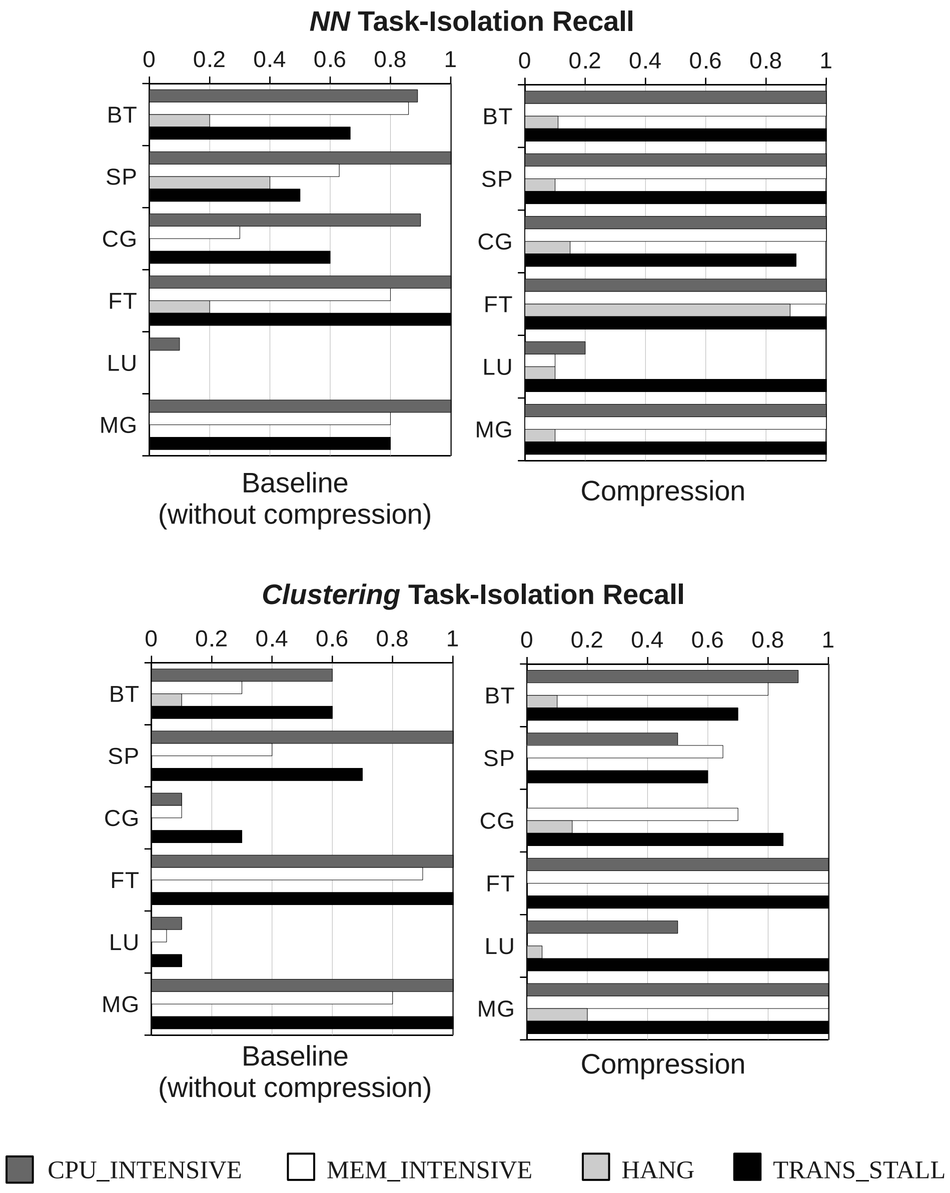


Fig. 4.6.: Task-isolation results for NN and clustering.

Figure 4.6 shows the task isolation results for the baseline (without compression) and the compression approach, for the two outlier detection methods. Table 4.1 shows graph sizes for the baseline and the compression method for faults that do not suspend the execution of the program. When the program's execution is suspended due to a

Benchmark	Original	Compressed	Comp. Ratio
BT	207	55	3.76
SP	179	55	3.25
CG	129	66	1.95
FT	21	4	5.25
LU	46	29	1.59
MG	81	33	2.45

Table 4.1: Edge counts for fault injection experiments.

hang or segmentation fault, the size of the graph can vary depending on the number of transitions observed in the last snapshot time and is therefore not meaningful.

As we observe from Figure 4.6, compressing the SMM improves the accuracy of detecting the anomalous process in both the NN and the clustering methods. For example, in the NN method when we inject the `CPU_INTENSIVE` fault in BT, recall in the baseline approach is about 85% whereas with compression it is 100%. In the clustering method, for the same benchmark and fault, recall is improved from 60% in baseline to 90% with compression. Compression improves process-isolation recall because the dimensionality reduction that results from merging contiguous edges eliminates noisy (unimportant) dimensions from the outlier-detection analysis and allows *AutomaDeD* to focus its power on the significant dimensions.

These results also suggest that the NN method detects errors better than the clustering method for half of the tested benchmarks, while both perform equally well for the other half. However, we expect clustering to have better accuracy than NN for cases in which a fault manifestation affects more than one process, as previously discussed.

Figure 4.7 shows the edge isolation results. *AutomaDeD* provides a high overall edge-isolation recall for most injected faults. If *AutomaDeD* correctly detects the faulty task in the previous step, it can guide the developer, with a high accuracy, to the code region in which the fault first manifests itself as a timing abnormality. This positive result demonstrates that the compression of the graph does not affect this step, nor does the sampling approach that we used to make NN or clustering scalable. Thus, our sampling strategy is probably unbiased and provides representative samples.

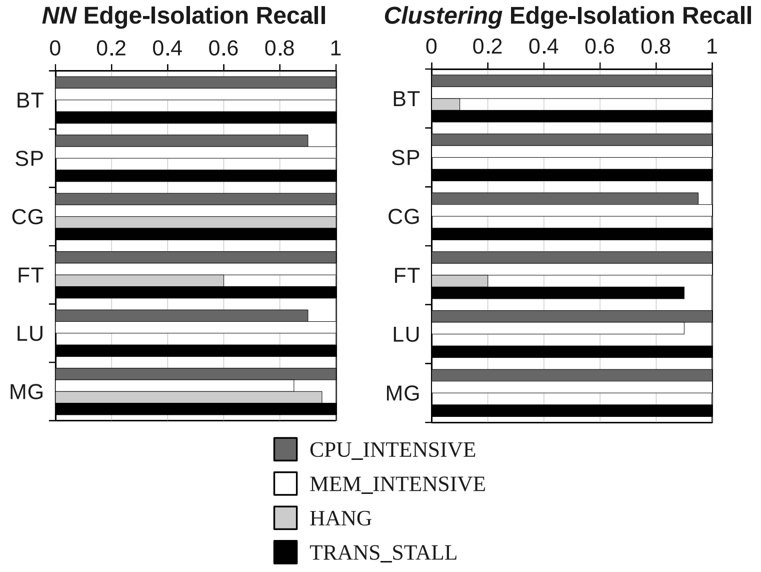


Fig. 4.7.: Edge-isolation with NN and clustering.

4.3.3 Performance Results

We evaluate the performance improvement achieved by calculating Lk-norm values using a pre-computed table. Figure 4.8 shows times for the baseline case (estimating integral (4.1)) and the pre-computed case. We measure the time to calculate the dissimilarity between two SMMs while we vary the number of edges in the SMMs. The use of a pre-computed table improves performance significantly. For example, for two large SMMs of 1000 edges, the dissimilarity calculation takes 0.45 seconds when computing Lk-norms online, while it takes 10 milliseconds in the pre-computed case.

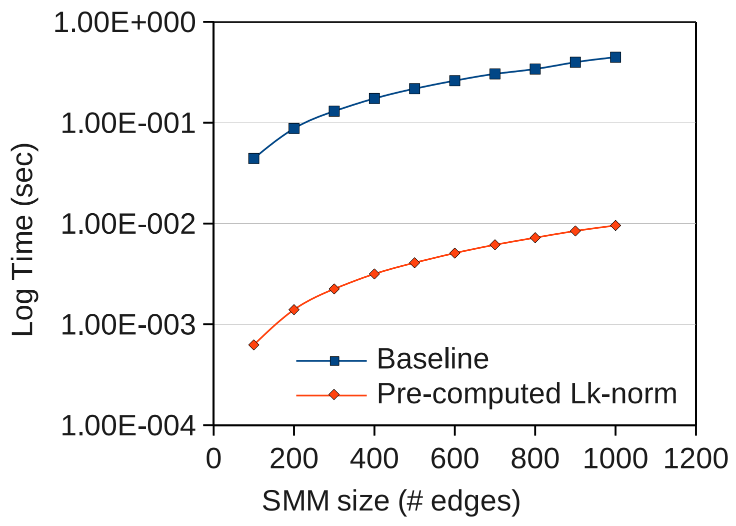


Fig. 4.8.: Lk-norm computation times.

We evaluate *AutomaDeD* at larger scales and measure the time to perform the entire error-detection analysis ending in the edge isolation. We measure the individual times for each part of the analysis: compression, edge- and task-isolation, as we vary the number of tasks to more than 5,000. For this experiment we use the Algebraic MultiGrid (AMG) 2006 benchmark from the Sequoia benchmark suite [66], a scalable iterative solver and preconditioner for solving large unstructured sparse linear systems. We run experiments in the same cluster as our fault injection experiments. The analysis uses the SMM that corresponds to the entire execution of AMG. Figure 4.9 shows the results of these experiments. The graph size without compression is (on average) 192 edges, and with compression is 158, with a compression ratio of $192/158 = 1.22$.

As we observe from Figure 4.9, we can apply *AutomaDeD* at increasingly large scales with relatively little overhead for the error-detection analysis. For example, for runs of 5,832 tasks, the analysis takes less than 5 seconds for both the NN and the clustering methods. Graph compression only incurs a small overhead, on the order of 170 milliseconds for the largest runs. However it substantially improves the accuracy

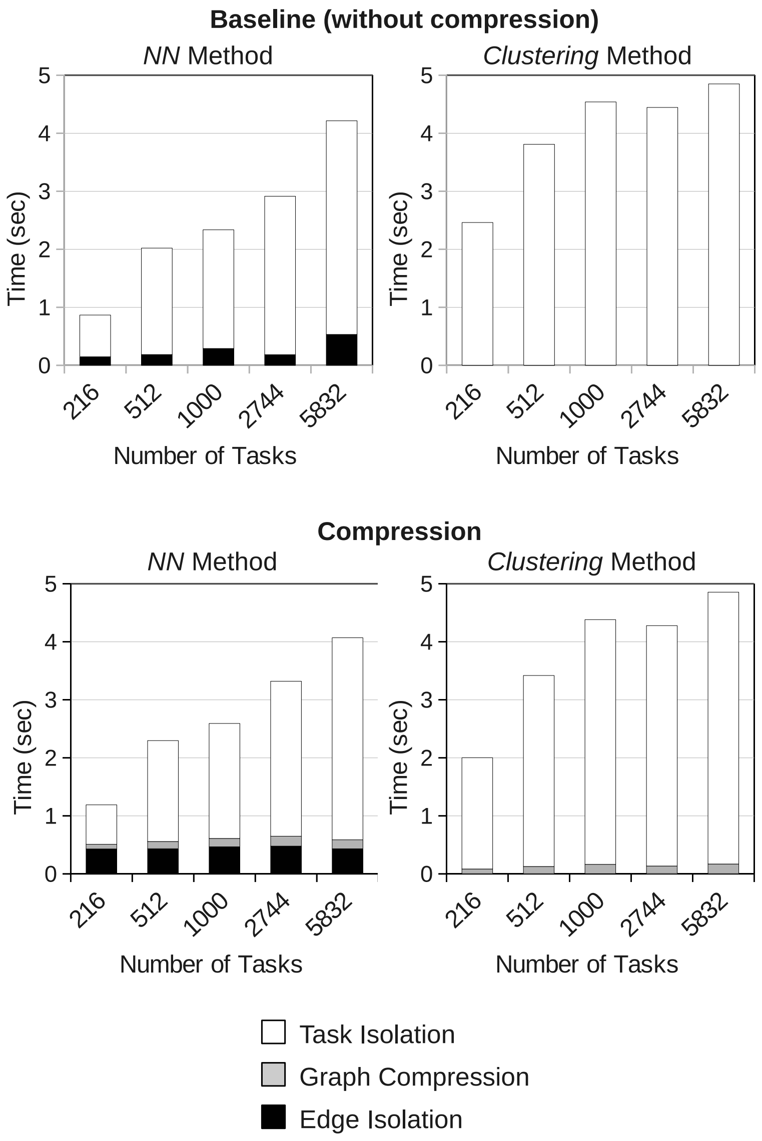


Fig. 4.9.: Time to isolate tasks and edges for the AMG2006 benchmark.

of error detection, as shown in the fault injection results (Figure 4.6). Compression requires little time because the core of the computation is performed locally in each process with a relatively small number of edges, (e.g., less than 250 edges for the AMG benchmark and the NAS Parallel Benchmarks). Despite compression decreasing edge support, this collective operation communicates little data since pairs of states in edges are represented as pointers (instead of strings as in our baseline).

The edge isolation step in NN (around 440 milliseconds) is larger than in the clustering method (around 4.5 milliseconds) because it compares edges of the abnormal processes to *multiple* sample SMMs, while the clustering method only compares edges of the abnormal process to one sample SMM, i.e., the medoid SMM. However, as the edge isolation results show, comparing edges to only one sample SMM suffices to produce a similar edge-isolation recall. These results demonstrate that the scalable techniques implemented in *AutomaDeD* make it suitable for online analysis in production runs. That is, *AutomaDeD* could be applied at multiple points in time as the application executes (possibly for several days) to find erroneous tasks and code regions.

To see the trend of the analysis time, we compute trend curves of the total time as Figure 4.10 shows—the total time corresponds to the sum of the three steps that are shown in Figure 4.9. Logarithmic curves accurately model the observed data, which matches our expectation that the cost of our analysis scales logarithmically with system size. The algorithmic complexity of the techniques in the outlier detection step, which has an O(`log n`) scaling, dominate this cost. Evaluating the equations of the trend curves, the analysis would take 8.67 seconds for 10,000 tasks, and 11.29 seconds for 100,000 tasks, for the clustering method. While we realize that such extrapolations are problematic and not always accurate, they do show that there should be no inherent limits to scaling our approach and that *AutomaDeD* has the potential to be appropriate as an online tool at our target large scales.

4.4 Discussion

We have implemented novel techniques in *AutomaDeD* that enables it to achieve scalability by optimizing it at different levels of its procedures. First, we minimize the time to compare elements of task models by using efficient data structures and approximation methods. Second, we reduce the sizes of the models to an appropriate magnitude, which eliminates noisy dimensions when finding the task affected by a

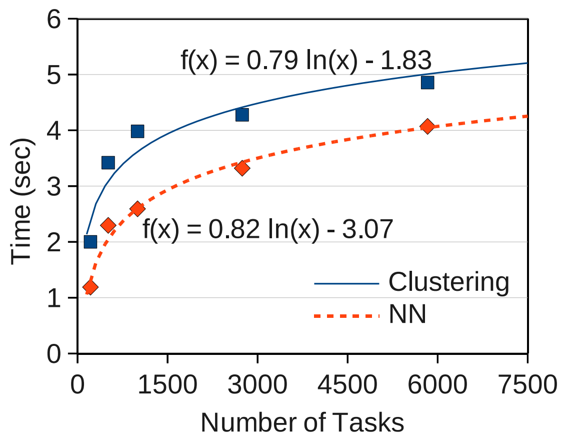


Fig. 4.10.: Trend lines for the total analysis time.

fault. Finally, we use sampling-based techniques such as CAPEK's clustering and scalable nearest neighbor to deal with the increasing number of parallel tasks that are present in today's largest systems. Our implementation scales easily to thousands of tasks and it can identify erroneous tasks and code regions in a few seconds. With this performance, *AutomaDeD* can be used not only in debugging runs, but also in production runs in an online manner in which *AutomaDeD*'s analysis would be applied periodically as the application runs (e.g., at boundaries of application phases) to detect problems automatically.

5. PROBLEM LOCALIZATION IN SCIENTIFIC APPLICATIONS

In this chapter we present novel techniques to perform problem localization in HPC applications. We extent our previous work, *AutomaDeD*, to help developers understand and fix performance failures at scale. The proposed technique probabilistically infers the least progressed task in MPI programs using Markov models of execution history and dependence analysis. This analysis guides program slicing to find code that may have caused a failure. In a blind study, we demonstrate that our tool can isolate the root cause of a particularly perplexing bug encountered at scale in a molecular dynamics simulation. Further, we perform fault injections into two benchmark codes and measure the scalability of the tool. Our results show that it accurately detects the least progressed task in most cases and can perform the diagnosis in a fraction of a second with thousands of tasks.

5.1 Introduction

We present a framework that provides insight into performance faults in large-scale parallel applications. Our framework identifies the *least progressed (LP)* task (or tasks) probabilistically in parallel code by using a Markov Model (MM) as a lightweight, statistical summary of each task's control-flow history. MM states represent MPI calls and computation, and edges represent state transitions (i.e., transfer of control between two code regions). This model lets us associate faults with code locations. However, in parallel applications, faults may lie on separate tasks from their root causes, so we introduce *progress dependence* to diagnose performance faults in parallel applications. We create a *progress dependence graph* (PDG) to capture wait chains of non-faulty tasks that depend on the faulty task to progress. We use these chains to find the LP task in parallel. Once we find the LP task, we apply program slicing [67] on the task's state to identify code that may have caused it to fail. We implement this framework as an extension to the *AutomaDeD* tool (which was described in Chapters 3 and 4).

To ensure scalability, we use a novel, fully distributed algorithm to create the PDG. Our algorithm uses minimal per-task information, and it incurs only slight runtime overhead for the applications we tested. Our implementation is non-intrusive, using the MPI profiling interface to intercept communication calls, and it does not require separate daemons to trace the application as other tools do (e.g., TotalView [2], STAT [17]).

We evaluate *AutomaDeD* by performing fault injections on AMG2006 and LAMMPS, two of the ASC Sequoia benchmarks. *AutomaDeD* finds a faulty task in a fraction of a second on up to 32,768 tasks. *AutomaDeD* accurately identifies the LP task 88% of the time, with perfect precision 86% of the time. We show that *AutomaDeD* can diagnose a difficult bug in a molecular dynamics code [68] that manifested only with 7,996 or more processes. *AutomaDeD* quickly found the fault—a sophisticated deadlock condition.

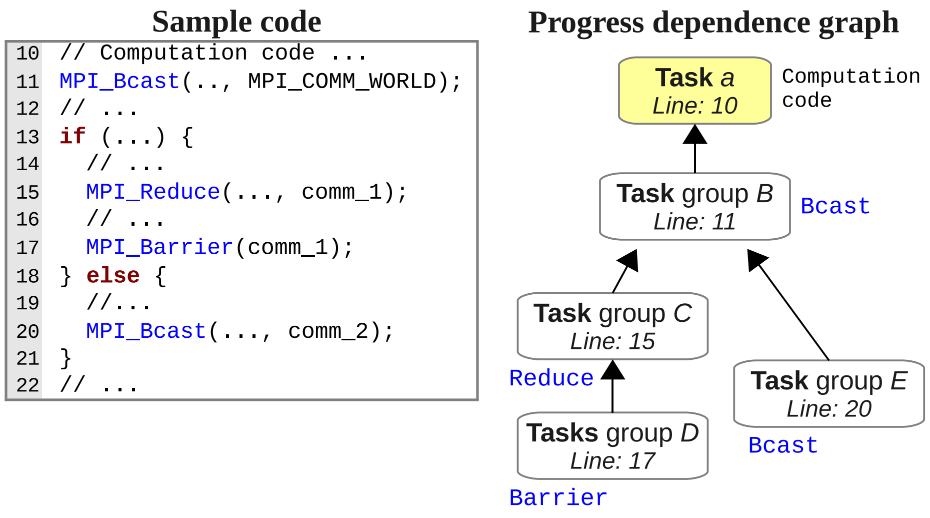
5.2 Overview of the Approach

5.2.1 Progress Dependence Graph

A *progress-dependence graph* (PDG) represents dependencies that prevent tasks from making further execution progress at any given time. It facilitates pinpointing performance faults that cause failures such as program stalls, deadlocks and slow code regions, and in performance tuning the application (e.g., by highlighting tasks with the least progress).

A PDG starts with the observation that two or more tasks must execute an MPI collective in order for (all of) them to move forward in the execution flow. For example, `MPI_Reduce` is often implemented in MPI using a binomial tree for short messages [69]. Since the MPI standard does not require collectives to be synchronized, some tasks could enter and leave this state — the `MPI_Reduce` function call — while others remain in it. Tasks that only send messages in the binomial tree enter and leave this state, while tasks that receive (and later send) messages block in this state until the corresponding sender arrives. These blocked tasks are *progress dependent* on other (possibly delayed) tasks.

This definition formalizes progress dependence (for collective operations): *Let the set of tasks that participate in a collective operation be X. If a task subset $Y \subseteq X$ has reached the collective operation while another task subsets $Z \subseteq X$, where $X = Y \cup Z$ has not yet reached it at time t such that the tasks in Y blocked at t waiting for tasks in Z then Y is progress-dependent on Z, which we denote as $Y \xrightarrow{pd} Z$.*



```
10 // Computation code ...
11 MPI_Bcast(.., MPI_COMM_WORLD);
12 // ...
13 if (...) {
14   // ...
15   MPI_Reduce(..., comm_1);
16   // ...
17   MPI_Barrier(comm_1);
18 } else {
19   //...
20   MPI_Bcast(..., comm_2);
21 }
22 // ...
```

Fig. 5.1.: Progress dependence graph example.

Figure 5.1 shows an example PDG in which task a blocks in (computation code) line 10. Task a could block for many reasons, such as a deadlock due to incorrect thread-level synchronization. As a consequence, a group of tasks B block in `MPI_Bcast` in line 11 while other tasks proceed to other code regions — tasks group C, D and E block in code lines 15, 17, and 20. No progress-dependence exists between groups C and E because they are in different execution branches.

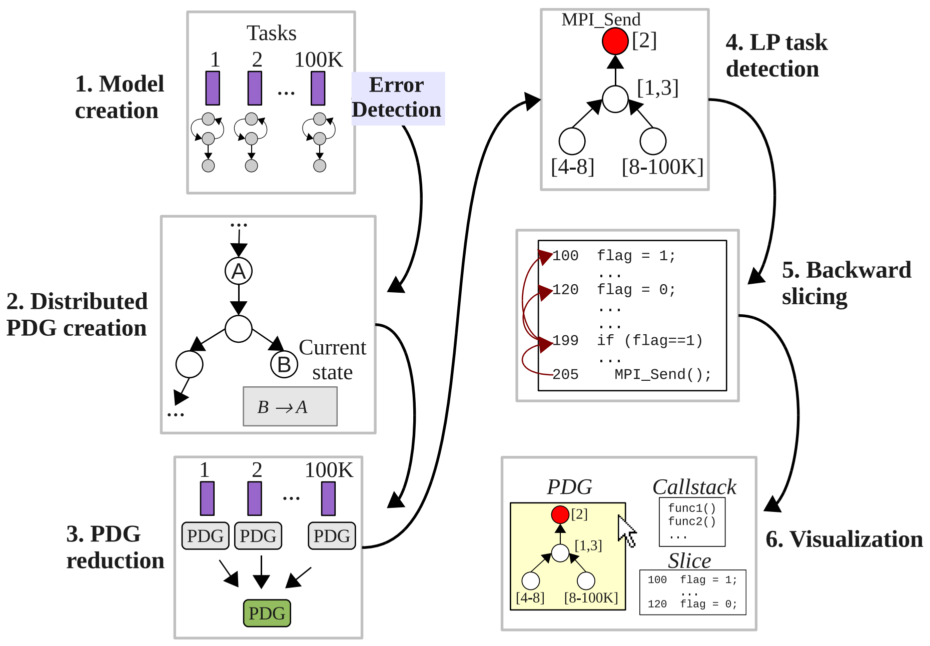
Point-to-Point Operations: In blocking point-to-point operations such as `MPI_Send` and `MPI_Recv`, the dependence is only on the peer task, which we formalize as: *If task x blocks when sending (receiving) a message to (from) task y at time t then x is progress dependent on y, i.e., $x \xrightarrow{pd} y$.* This definition also applies to nonblocking operations such as `MPI_Isend` and `MPI_Irecv`. The main difference is that the dependence does not apply directly to the send (or receive) operation, but to the associated completion (e.g., a wait operation or a user-level test loop). If a task x blocks on `MPI_Wait`, for example, we infer the task y, on which x is progress dependent, from the request on which x waits. Similarly, if x spins on a user test loop, for example by calling `MPI_Test` within a loop, we infer the peer task on which x is progress dependent from the associated request (within the user loop). On the receiving end, we can also infer the dependence from other test operations such as `MPI_Probe` or `MPI_Iprobe`. In any case, we denote the progress dependence as $x \xrightarrow{pd} y$.

PDG-Based Diagnosis: A PDG can intuitively pinpoint the task (or task group) that originates a performance failure. In Figure 5.1, task a can be blamed for originating the program's stall since it has no progress dependence on any other task (or group of tasks) in the PDG. It is also the least progressed (LP) task.

From the programmer's perspective, the PDG provides useful information in debugging and performance tuning. First, given a performance failure such as the one in Figure 5.1, the PDG shows where to focus attention, i.e., the LP task(s). Thus, debugging time is substantially reduced, as the programmer can focus on the execution context of one (or a few) task(s) rather than on possibly thousands of tasks. Second, we can efficiently apply static or dynamic bug-detection methods based on the LP task(s) state. *AutomaDeD* applies slicing [67] using the state of the LP task as initial criterion, which substantially reduces the search space of slicing when compared to slicing the execution context of each task (or representative task group) separately and then combining this information to find the fault.

PDG Versus Other Dependency Graphs: A PDG is similar to the dependency graph used in MPI deadlock detection [70–72] since both graphs express dependencies between groups of processes; however, a PDG hierarchically describes the execution progress of MPI tasks. A PDG addresses questions—that cannot be answered directly from deadlock-detection dependency graphs—such as: Which task has made the least progress? Which tasks does the LP task prevent from making progress? In contrast, knots in traditional dependency graphs can detect real and potential deadlocks. We do not detect deadlocks by checking for knots in a PDG. However, since a PDG combines dependencies arising from MPI operations, it can indicate that a deadlock caused a hang. Performance failures are a superset of hangs; deadlocks or other causes can lead to hangs. Our case study with a real-world bug in Section 5.5.1 shows an example in which we use a PDG to identify that a deadlock was the root-cause of a hang.

5.2.2 Workflow of Our Approach



```
flag = 1;
...
flag = 0;
...
...
if (flag==1)
...
  MPI_Send();
```

```
func1()
func2()
...
```

```
flag = 1;
...
flag = 0;
```

Fig. 5.2.: Overview of the diagnosis work flow.

Figure 5.2 shows the steps in *AutomaDeD* to diagnose performance problems. Steps 1–3 are distributed while steps 4–6 are performed in a single task.

(1) Model creation. *AutomaDeD* captures per-MPI-task control-flow behavior in a Markov model (MM). These models are similar to the semi-Markov models (SMM) that *AutomaDeD* creates as described in Chapters 3 and 4, except that we do not compute a time distribution for each edge. MM states correspond to two code region types: *communication* regions, i.e., code executed within an MPI function; and *computation* regions, i.e., code executed between two MPI functions.

(2) Distributed PDG creation. When a system detects a performance fault (either *AutomaDeD* or a third-party system), *AutomaDeD* uses a distributed algorithm to create a PDG in each task. First, we use an all-reduce over the MM state of each task, which provides each task with the state of all other tasks. Formally, if a task's local state is s_{local}, this operation provides each task with the set $S_{\text{others}} = s_1, \ldots, s_j, \ldots, s_N$, where $s_j \neq s_{\text{local}}$. Next, each task probabilistically infers its own local PDG based on s_{local} and S_{others}.

(3) PDG reduction. Our next distributed step reduces the PDGs from step (2) to a single PDG. The reduction operation is the union of edges in two PDGs, i.e., the union (in each step of the reduction) of progress dependencies.

(4) LP task detection. Based on the reduced PDG, we determine the LP task and its state (i.e., call stack and program counter), which we use in the next step.

(5) Backward slicing. We then perform backward slicing using Dyninst [73]. This step finds code that could have led the LP task to reach its current state.

(6) Visualization. Finally, *AutomaDeD* presents the program slice, the reduced PDG and its associated information. The user can attach a serial or parallel debugger to the LP task based on the PDG. The PDG also provides other task groups and their dependencies. The slice brings the programmer's attention to code that affected the LP task, and allows them to find the fault.

5.3 Design

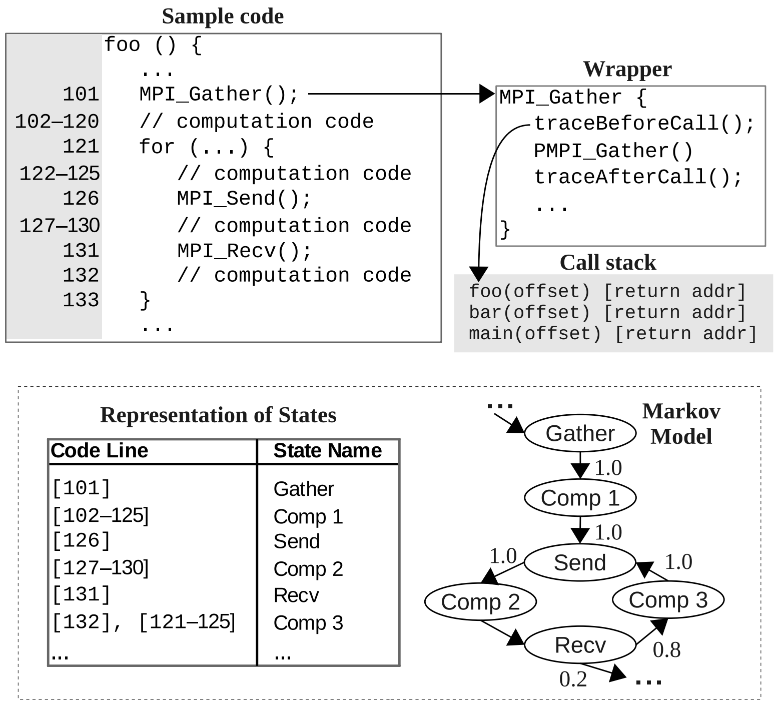
5.3.1 Summarizing Execution History

A simple approach to save the control-flow execution history directly might build a control-flow graph (CFG) based on executed statements [74]. Since large-scale MPI applications can have very large CFGs, *AutomaDeD* instead captures a compressed version of the control-flow behavior using our MM with communication and computation states. The edge weights capture the frequency of transitions between two states.

Figure 5.3 shows how *AutomaDeD* creates MMs at runtime in each task. We use the MPI profiling interface to intercept MPI routines. Before and after calling the corresponding PMPI routine, *AutomaDeD* captures information such as the call stack, offset address within each active function and return address. We assume that the MPI program is compiled using debugging information so that we can resolve function names. Our modeling assumes that if two MM states are the same, i.e., they correspond to the same beginning or end of an MPI function call with the same call stack and addresses, then the two corresponding MPI calls are the same, i.e., they are in the same file and line of code of the monitored application. This assumption applies for both point-to-point and collective operations.

5.3.2 Progress Dependence Inference

In this section, we discuss how we infer progress dependence probabilistically from our MMs. We restrict the discussion to dependencies that arise from collective operations, since dependencies from point-to-point operations do not require our probabilistic analysis. For example, if task t_i is waiting for another task in `MPI_Recv`, *AutomaDeD* uses the parameters of the MPI call to determine on which task t_i is progress-dependent. In cases when a task blocks in `MPI_Wait`, for example when using non-blocking operations, *AutomaDeD* uses request handlers to identify the matching



Sample code

```
         foo () {
           ...
    101    MPI_Gather();
102–120    // computation code
    121    for (...) {
122–125        // computation code
    126        MPI_Send();
127–130        // computation code
    131        MPI_Recv();
    132        // computation code
    133    }
           ...
```

Wrapper

```
MPI_Gather {
   traceBeforeCall();
   PMPI_Gather()
   traceAfterCall();
   ...
}
```

Call stack

```
foo(offset) [return addr]
bar(offset) [return addr]
main(offset) [return addr]
```

Representation of States

Code Line	State Name
[101]	Gather
[102–125]	Comp 1
[126]	Send
[127–130]	Comp 2
[131]	Recv
[132], [121–125]	Comp 3
...	...

Fig. 5.3.: Markov model creation.

progress-dependence task. We cannot infer progress dependence for MPI_ANY_SOURCE, in which case *AutomaDeD* omits this progress-dependence edge, which reflects the probabilistic nature of our approach.

In contrast to dependencies from point-to-point operations, collective operations do not allow us to infer progress dependencies from the parameters of the MPI call. The parameters in a collective operation are buffers, data types, root process and communicators. We can infer the tasks that have to execute a collective operation from the communicator, i.e., the tasks that have to reach this (communication) state, but we cannot infer progress dependencies (assuming that we know the current states of all tasks). As an example consider the collective operation in Figure 5.4 in which two tasks, x and y, are members of the *comm* communicator. Suppose that task x is blocked in the collective operation itself and that task y is blocked in another state. Within task x, we can obtain the members of the communicator and infer that task y also has to reach this state but we do not know whether it has already visited this state (i.e., it is in a state after the collective operation) or it has not yet visited this state (i.e., it is in a state before the collective operation). Effectively we cannot determine $x \xrightarrow{pd} y$ or $y \xrightarrow{pd} x$.

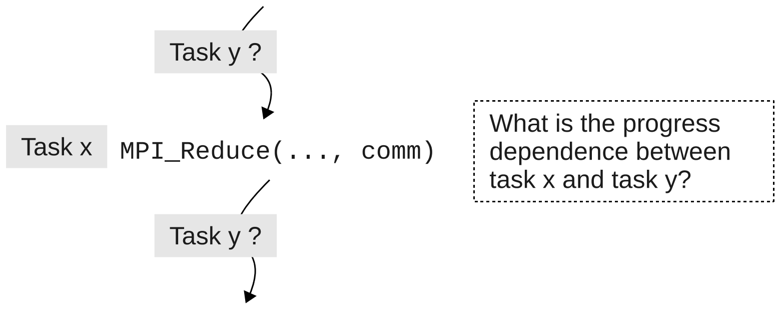


Fig. 5.4.: Example of a collective operation executed by two tasks.

AutomaDeD probabilistically infers progress dependence between a task's local state and the states of other tasks. Intuitively, our MM models the probability of going from state x to state y via *some* path $x \rightsquigarrow y$. If a task t_x in x must eventually reach y with high probability then we can determine that a task t_y in state y could be waiting for t_x in which case we infer that $y \xrightarrow{pd} x$ (for simplicity, we represent progress dependencies in terms of task states).

Figure 5.5 illustrates how we infer progress dependence from our MMs. Five tasks $(a, b, c, d$ and $e)$ are blocked in different states (1, 3, 5, 8, and 10 respectively). To estimate the progress dependence between task b and task c, we calculate that the path probability $P(3, 5)$, the probability of going from state 3 to state 5 over all possible paths, which is 1.0. Thus, *task c is likely to be waiting for task b, since according to the observed execution, if a task is in state 3 it always must reach state 5.* To estimate progress dependence more accurately, we consider the possibility of loops and evaluate the backward path probability $P(5, 3)$, which in this case is zero. Thus, task c cannot reach task b, so we can consider it to have progressed further than task b so $c \xrightarrow{pd} b$.

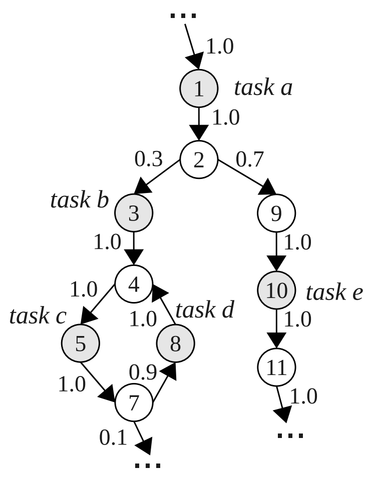


Fig. 5.5.: Sample Markov model with five blocked tasks.

Notice that the the Markov model in Figure 5.5 is a global view of all the Markov models (within all tasks) but *it is not* created at any time our PDG-construction algorithm. Different tasks could have different Markov models. As we explain later in the description of Figure 5.6, our PDG-construction algorithm only uses the local Markov model and the state of all the tasks to infer progress dependencies. The reason behind this design choice—rather than creating a global model as in Figure 5.5—is to reduce the cost of exchanging local models between all the tasks. Markov models could be large (in the order of thousands of states) and performing an all-to-all reduction of them can impact the scalability of our design.

Resolving conflicting probability values. When a backward path probability $P(j, i)$ is zero, a task in state j has made more progress than a task in state i. However, if the forward path probability $P(i, j)$ is 1.0 and the backward path probability is nonzero then the task in state j might return to i. For example, for tasks d and c in Figure 5.5, $P(8, 5) = 1.0$ but $P(5, 8) = 0.9$. In this case, task d must eventually reach state 5 to exit the loop so we estimate that $c \xrightarrow{pd} d$; our results demonstrate that this heuristic works well in practice.

The dependence between task b and task e is null, i.e., no progress dependence exists between them. They are in different execution branches so the forward and backward path probabilities between their states, i.e., $P(3, 10)$ and $P(10, 3)$, are both zero. The same holds for the dependencies between task e and task c or d.

Table 5.1: Dependence based on path probabilities.

$P(i,j)$			$P(j,i)$				
0	$0 < P < 1$	1	0	$0 < P < 1$	1	Dependence?	Type
✓			✓			No	
✓				✓		Yes	$t_i \xrightarrow{pd} t_j$
✓					✓	Yes	$t_i \xrightarrow{pd} t_j$
	✓		✓			Yes	$t_i \xleftarrow{pd} t_j$
	✓			✓		?	
	✓				✓	Yes	$t_i \xrightarrow{pd} t_j$
		✓	✓			Yes	$t_i \xleftarrow{pd} t_j$
		✓		✓		Yes	$t_i \xleftarrow{pd} t_j$
		✓			✓	?	

General progress dependence estimation. To estimate the progress dependence between tasks t_i and t_j in states i and j, we calculate two path probabilities: (i) a forward path probability $P(i, j)$; and (ii) a backward path probability $P(j, i)$. A path probability is the likelihood of going from one state to another over all possible paths. We use Table 6.2 to estimate progress dependencies. If both probabilities are zero (i.e., the tasks are in different execution branches), no dependence exists between the tasks. When one probability is 1.0 and the other is less than 1.0, the first *predominates* the second. Therefore, the second probability determines the dependence. For example, if the second is $P(j, i)$ we determine $t_j \xrightarrow{pd} t_i$ since execution goes from i to j. If one probability is zero and the second is nonzero, then the second predominates the first. Therefore, the first probability determines the dependence. For example, if the first is $P(i, j)$ we determine $t_i \xrightarrow{pd} t_j$ because execution could go from j to i but not from i to j.

We cannot determine progress dependence for two cases: when both probabilities are 1.0; and when both probabilities are in the range $0.0 < P < 1.0$. The first case could happen when two tasks are inside a loop and, due to an error, they do not leave the loop and block inside it. In this case both backward and forward path probabilities are 1.0, so it is an undefined situation. The probabilities in the second case simply do not provide enough information to decide. For these cases, *AutomaDeD* marks the edges in the PDG as undefined so that the user knows that the relationship could not be determined. These cases occur infrequently in our experimental evaluation. When they do, the user can usually determine the LP task by looking at tasks that are in one group or cluster. Section 5.5 gives examples of how the user can resolve these cases visually.

Algorithm. Figure 5.6 shows our local PDG construction algorithm, which takes an MM and *statesSet*, the states of all other tasks as input. We compute the dependency between the local state and *statesSet*. We represent dependencies as integers (0: no dependence; 1: forward dependence; 2: backward dependence; 3: undefined). We save the PDG in an adjacency matrix. The function `dependence` determines dependencies based on all-path probabilities (calculated in `probability`) and Table 6.2 (captured in `dependenceBasedOnTable`).

The overall complexity of the algorithm is $O(s \times (|V| + |E|))$, where s is the number of states in *statesSet*, and $|V|$ and $|E|$ are the numbers of states and edges of the MM. In practice, the MMs are sparse graphs in which $|E| \approx |V|$, so the complexity is approximately $O(s \times |E|)$.

Comparison to postdominance. Our definition of progress dependence is similar to the concept of postdominance [75] in which a node j of a CFG postdominates a node i if every path from i to the *exit* node includes j. However, our definition does not require the *exit* node to be in the MM (postdominance algorithms require it to create a postdominator tree). Since a fault could cause the program to stop in any state, we are not guaranteed to have an exit node within a loop. Techniques such as assigning a dummy exit node to a loop do not work in general for fault diagnosis be-

```
 *Input:* mm (Markov model), closure (transitive closure
of the mm), statesSet (set of states)
*Output:*depMatrix (PDG adjacency-matrix representation)

progressDependenceGraph() { /* Builds PDG */
  State localState ← getLocalState(mm)
  for each State s in statesSet
    if s is not localState {
      d ← dependence(localState, s)
      depMatrix[localState, s] ← d
    }
}

/* Calculates dependence between two states */
dependence(State src, State dst) {
  p ← probability(src, dst)
  d ← dependenceBasedOnTable(p)
  return d
}

/* Calculates reachability probability */
probability(State src, State dst) {
  p ← 0
  if src can reach dst in closure {
    for each Path p between src and dst
      p ← p + pathProbability(src, dst)
  }
  return p
}
```

Fig. 5.6.: Algorithm to create the PDG.

cause a faulty execution makes it difficult (or impossible) to determine the right exit node. In order to use postdominance theory, we could use static analysis to find the exit node and map it to a state in the MM. However, our dynamic analysis approach is more robust and should provide greater scalability and performance.

5.4 Scalable PDG-Based Analysis

This framework is implemented in *AutomaDeD* in C++ and uses the Boost Graph Library [76] for graph-related algorithms such as depth-first search. In this section, we focus on the implementation details that ensure scalability.

5.4.1 Error Detection

We assume that a performance problem has been detected, for example, because the application is not producing the expected output in a timely manner. The user can then use *AutomaDeD* to find the tasks and the associated code region that caused the problem. *AutomaDeD* includes a timeout detection mechanism that can trigger the diagnosis analysis, and it can infer a reasonable timeout threshold (based on the mean time and standard deviation of state transitions). The user can also supply the timeout as an input parameter. Our experiments with large-scale HPC applications found that a 60 second threshold is sufficient.

5.4.2 Distributed Inference of the PDG

Helper thread. *AutomaDeD* uses a helper thread to analyze the MM (that is built in the main thread). The core of the dependence inference, Steps 2–3 in Figure 5.2, is distributed while only one task (MPI rank 0 by default) performs the inexpensive operations in Steps 4–6. *AutomaDeD* uses `MPI_THREAD_MULTIPLE` to initialize MPI so that multiple threads can call MPI. On machines that do not support threads, such as BlueGene/L, we save all MMs to the parallel file system when we detect an error. *AutomaDeD* then reads these MMs for analysis in a separate MPI program.

Distributed algorithm. The following steps provide more detail of Steps 2–3 in the workflow:

(1) We first perform a reduction over the current state of all tasks to compute the

Table 5.2: Some examples of dependence unions.

No	Task x	Task y	Union	Reasoning	OR operation
1	$i \rightarrow j$	null	$i \rightarrow j$	first dependence dominates	$1 + 0 = 1$
2	$i \rightarrow j$	$i \rightarrow j$	$i \rightarrow j$	same dependence	$1 + 1 = 1$
3	$i \leftarrow j$	$i \leftarrow j$	$i \leftarrow j$	same dependence	$2 + 2 = 2$
4	$i \rightarrow j$	$i \leftarrow j$	$i?j$	undefined	$1 + 2 = 3$
5	null	null	null	no dependence	$0 + 0 = 0$

statesSet of all tasks.

(2) We next broadcast *statesSet* to all tasks.

(3) Each task uses the algorithm in Figure 5.6 to compute its local PDG from its local state and *statesSet*.

(4) Finally, a reduction of the local PDGs calculates the union of the edges (forward or backward). Table 5.2 shows examples of some union results. In case 1, a dependence is present in only one task so the dependence predominates. In cases 2 and 3, the dependencies are similar so we retain it. In case 4, they conflict so the resulting dependence is undefined. We efficiently implement this operator using bitwise `OR` since we represent dependencies as integers.

We cannot use `MPI_Reduce` for our reduction steps (for example, tasks can contribute states of different sizes) so we implement custom reductions that use binomial trees. These operations have $O(\log p)$ complexity where p is the number of tasks. Assuming a scalable broadcast implementation, the overall complexity is also $O(\log p)$. Our algorithm can therefore scale to the task counts found on even the largest HPC systems.

5.4.3 Determination of LP Task

We compute the LP task (or task group) from the reduced PDG. *AutomaDeD* first finds nodes with no outgoing edges based on dependencies from collectives and marked them as LP. If more than one node is found, *AutomaDeD* discards nodes that have point-to-point dependencies on other non-LP tasks in different branches. Since *AutomaDeD* operates on a probabilistic framework (rather than on deterministic methods [17]), it can incorrectly pinpoint the LP task, although such errors are rare according to our evaluation. However, in most of these cases, the user can still determine the LP task by visually examining the PDG (by looking for nodes with only one task).

5.4.4 Guided Application of Program Slicing

Background. Program slicing transforms a large program into a smaller one that contains only statements that are relevant to a particular variable or statement. For debugging, we only care about statements that could have led to the failure. However, message-passing programs complicate program slicing since we must reflect dependencies related to message operations.

We can compute a program slice statically or dynamically. We can use static data and control flow analysis to compute a static slice [67], which is valid for all possible executions. Dynamic slicing [77] only considers a particular execution so it produces smaller and more accurate slices for debugging.

Most slicing techniques that have been proposed for debugging message-passing programs are based on dynamic slicing [78–80]. However, dynamically slicing of a message-passing program usually does not scale well. Most proposed techniques have complexity at least $O(p)$. Further, the dynamic approach suffers high costs to generate traces of each task (typically by code instrumentation) and to aggregate those traces centrally to construct the slice. Some approaches reduce the size of dynamic slices by using a global predicate rather than a variable [79, 80]. However, the violation of the global predicate may not provide sufficient information to diagnose failures in complex MPI programs.

We can use static slicing if we allow some inaccuracy. However, we cannot naively apply data-flow analysis (which slicing uses) in message-passing programs [81]. For example, consider the following code fragment:

```
 1 program() {
 2    ...
 3    if (rank == 0) {
 4      x = 10;
 5      MPI_Send(...,&x,...);
 6    } else {
 7      MPI_Recv(...,&y,...);
 8      result = y * z;
 9      printf(result);
10      ...
```

Traditional slicing on `result` in line 9 identifies statements 7, 8, and 9 as the only statements in the slice, but statements 3–9 should be in the slice. Statements 4-5 should be in the slice because the value x sent is received as y which obviously influences z. Thus, we must consider the SPMD nature of the program in order to capture communication dependencies. The major problem with this *communication-aware slicing* is the high cost of analyzing a large dependence graph [81] to construct a slice based on a particular statement or variable. Further, the MPI developer must decide on which tasks to apply communication-aware static slicing since applying it to every task is infeasible at large scales.

Approach. *AutomaDeD* progressively applies slicing to the execution context of tasks that are representative of behavioral groups, starting with the groups that are most relevant to the failure based on the PDG. *AutomaDeD* uses the following algorithm:

(1) Initialize an empty slice S.

(2) Iterate over PDG nodes from the node corresponding to the LP task to nodes that depend on it, and so on to the leaf nodes (i.e., the most progressed tasks).

(3) In each iteration i, $S = S \bigcup s_i$ where s_i is the statement set produced from the state of a task in node i.

This slicing method reduces the complexity of manually applying static slicing to diagnose a failure. The user can simply start with the most important slice (i.e., the one associated with the LP task) and progressively augment it by clicking the "next" button in a graphical interface, until the fault is found.

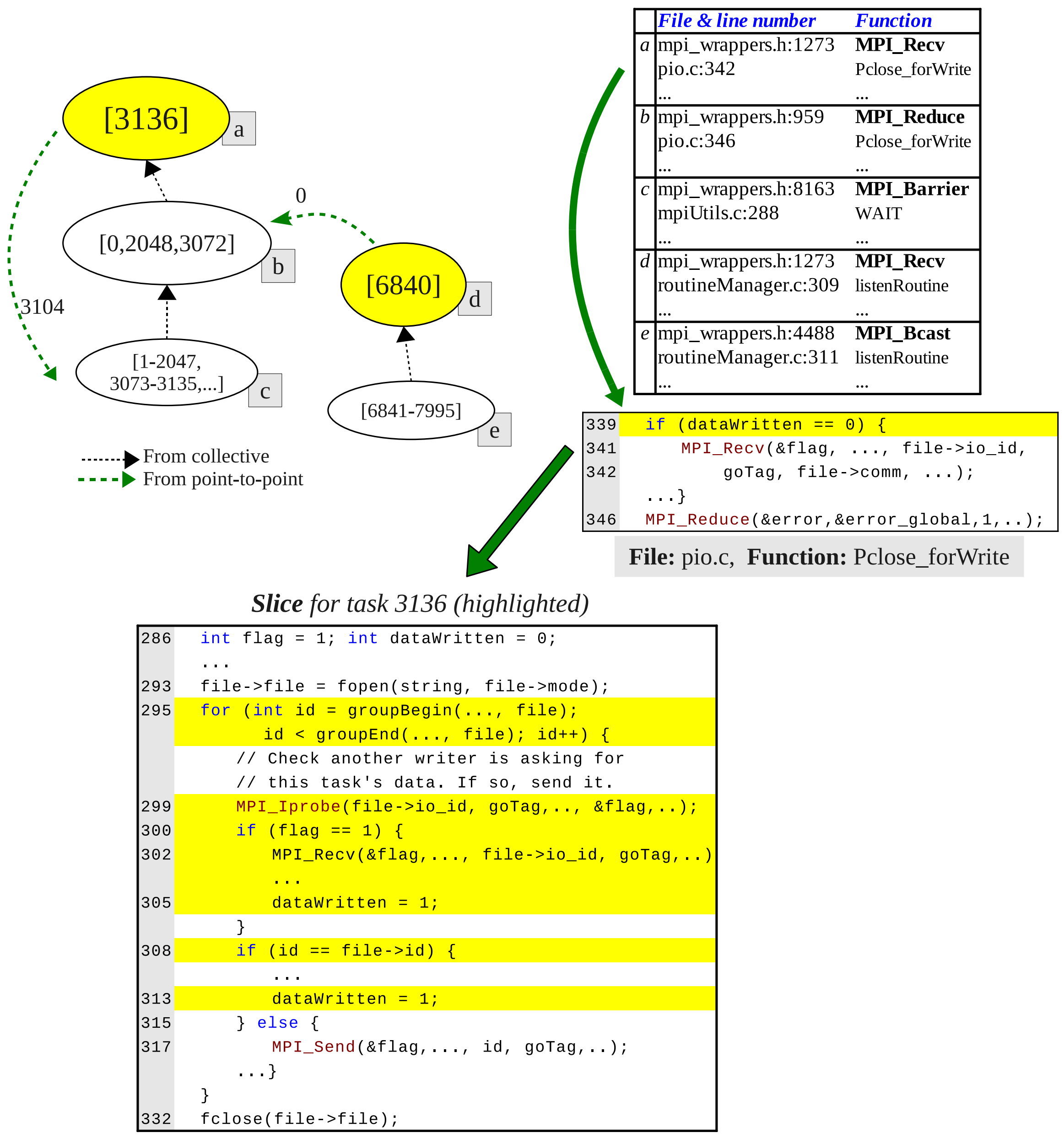
5.5 Evaluation

We demonstrate how *AutomaDeD* diagnose a difficult bug in a molecular dynamics program that manifested only at large scale. We also perform 280 experiments to evaluate *AutomaDeD* in a controlled setting: 50 fault injection experiments; 160 slowdown and memory usage experiments; and 70 scalability experiments. The next sections provide the experimental settings and main results.

5.5.1 Case Study

An application scientist challenged us to locate an elusive error in ddcMD, a parallel classical molecular-dynamic code [68]. The bug manifested as a hang that emerged intermittently only when run on Blue Gene/L with 7,996 MPI tasks. Although he had already identified and fixed the error after significant time and effort, he hoped that we could provide a technique that would not require tens of hours. In this section, we present a blind case study, in which we were supplied no details of the error, that demonstrates *AutomaDeD* can efficiently locate the origin of faults.

Figure 5.7 shows the result of our analysis. Our tool first detects the hang condition when the code stops making progress, which triggers the PDG analysis to identify MPI task 3,136 as the LP task — *AutomaDeD* first detects tasks 3,136 and 6,840 as LP tasks and then eliminates 6,840 since it is point-to-point dependent on task 0, a non-LP task, in the left branch. The LP task in the `a` state, causes tasks in the `b` state that immediately depend on its progress to block, ultimately leading to a global stall through the chain of progress dependencies. This analysis step reveals that task 3,136 stops progressing as it waits on an `MPI_Recv` within the `Pclose_forWrite` function.



	File & line number	*Function*
a	mpi_wrappers.h:1273 pio.c:342 ...	**MPI_Recv** Pclose_forWrite ...
b	mpi_wrappers.h:959 pio.c:346 ...	**MPI_Reduce** Pclose_forWrite ...
c	mpi_wrappers.h:8163 mpiUtils.c:288 ...	**MPI_Barrier** WAIT ...
d	mpi_wrappers.h:1273 routineManager.c:309 ...	**MPI_Recv** listenRoutine ...
e	mpi_wrappers.h:4488 routineManager.c:311 ...	**MPI_Bcast** listenRoutine ...

```
339   if (dataWritten == 0) {
341       MPI_Recv(&flag, ..., file->io_id,
342            goTag, file->comm, ...);
      ...}
346   MPI_Reduce(&error,&error_global,1,..);
```

```
286   int flag = 1; int dataWritten = 0;
      ...
293   file->file = fopen(string, file->mode);
295   for (int id = groupBegin(..., file);
             id < groupEnd(..., file); id++) {
          // Check another writer is asking for
          // this task's data. If so, send it.
299       MPI_Iprobe(file->io_id, goTag,.., &flag,..);
300       if (flag == 1) {
302           MPI_Recv(&flag,..., file->io_id, goTag,..)
              ...
305           dataWritten = 1;
          }
308       if (id == file->id) {
              ...
313           dataWritten = 1;
315       } else {
317           MPI_Send(&flag,..., id, goTag,..);
          ...}
      }
332   fclose(file->file);
```

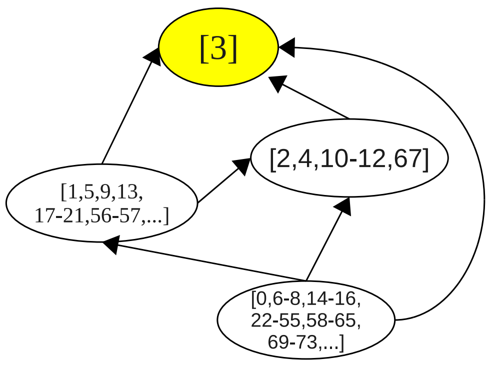
Fig. 5.7.: Output for ddcMD bug.

Once it identifies the LP task, *AutomaDeD* applies backward slicing starting from the a state, which identifies `dataWritten` as the data variable that most immediately pertains to the current point of execution. Slicing then highlights all statements that could directly or indirectly have affected its state.

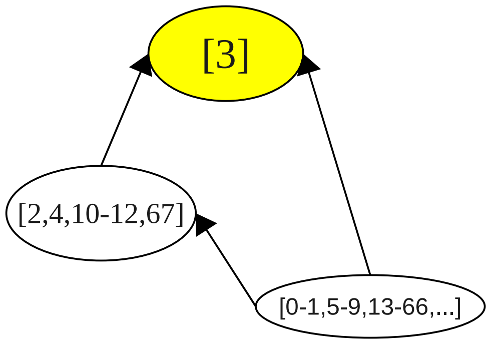
The application scientist verified that our analysis precisely identified the location of the fault. ddcMD implements a user-level, buffered file I/O layer called `pio`. MPI tasks call various `pio` functions to move their output to local per-task buffers and later call `Pclose_forWrite` to flush them out to the parallel file system. Further, in order to avoid an I/O storm at large scales, `pio` organizes tasks into I/O groups. Within each group, one writer task performs the actual file I/O on behalf of all other group members. A race condition in the complex writer nomination algorithm — optimized for a platform-specific I/O forwarding constraint — and overlapping consecutive I/O operations causes the intermittent hang. The application scientist stated that the LP task identification and highlighted statements would have provided him with critical insight about the error. He further verified that a highlighted statement was the bug site.

More specifically, on Blue Gene/L, a number of compute nodes perform their file I/O through a dedicated I/O node (ION) so `pio` nominates *only one* writer task per ION. Thus, depending on how MPI tasks map to the underlying IONs, an I/O group does not always contain its writer task. In this case, `pio` instead nominates a non-member task that belongs to a different I/O group. This mechanism led to a condition in which a task plays dual roles: a non-writer for its own I/O group and the writer for a different group.

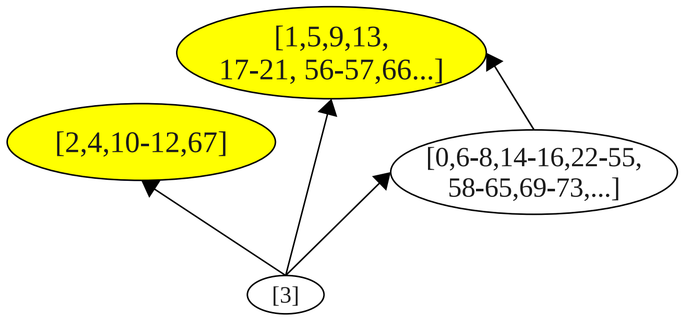
Figure 5.7 shows the main loop of a writer. To receive the file buffer from a non-writer, the group writer first sends a request to each of its group members to send the file buffer via the `MPI_Send` at line 317. The group member receives that request via the `MPI_Recv` at line 341 and sends back the buffer size and the buffer. As shown in the loop, a dual-purpose task has an extra logic: it uses `MPI_Iprobe` to test whether it must reply to its non-writer duty while it performs its writer duty. The logic is introduced in part to improve performance. However, completing that non-writer duty frees its associated writer task to move on from MPI blocking communications. The hang arises when two independent instances of `pio` are simultaneously processing two separate sets of buffers. This pattern occurs in the application when a small data set is written immediately after a large data set. Some tasks can still be performing communication for a large data set while others work on a small set. Because the MPI send/recv operations use tags that are fixed at compile time, messages from a small set could be confused for those for a large set of `pio` and vice-versa, leading to a condition in which a task could hang waiting for a message that was intercepted by a wrong instance.



(a) hypre_ParVectorCopy



(b) hypre_CSRMatrixMatvec



(c) MPI_Irecv

Fig. 5.8.: Examples of PDGs indicating LP tasks (highlighted)—errors are injected in task 3.

This error only arose on this particular platform because the dual-purpose condition only occurs under Blue Gene's unique I/O forwarding structure. We also theorize that the error emerges only at large scales because this scale increases the probability that the dual-purpose assignments and simultaneous `pio` instances occur. The application scientist had corrected the error through unique MPI tags in order to isolate one `pio` instance from another.

5.5.2 Fault injections

Applications. To evaluate *AutomaDeD*, we inject faults into two Sequoia benchmarks: AMG2006 and LAMMPS [82]. These codes are representative of large-scale HPC production workloads. AMG2006 is a scalable iterative solver for large structured sparse linear systems. LAMMPS is a classical molecular dynamics code. For AMG-2006, we use the default 3D problem (test 1) with the same size in each dimension. For LAMMPS, we use "crack", a crack propagation example in a 2D solid.

Injections. We inject a local application hang by making a randomly selected process suspend execution for a long period to activate the timeout error detection mechanism in *AutomaDeD*. We use Dyninst [73] to inject the fault into the application binaries as a sleep call at the beginning of randomly selected function calls (20 user, 5 MPI). Our injector first profiles a run of the application so that we randomly choose from functions that are used during the run. This ensured that all injections resulted in errors. We use a higher proportion of user function calls because more user functions than MPI functions are used at runtime. These function calls capture a wide range of program behaviors including calls inside complex loops as well as ones at the beginning or end of the program. We perform all fault-injection experiments on a Linux cluster with nodes that have six 2.8 GHz Intel Xeon processors, 24 GB of RAM and InfiniBand interconnect. We use 1,000 tasks in each experiment.

Coverage results. We use three metrics to evaluate diagnosis quality: *LPT detection recall*, the fraction of cases in which the set of LP tasks that *AutomaDeD* finds includes the faulty task; *isolation*, the fraction of cases in which the faulty task is not detected but it is the only task in a PDG node (i.e., a singleton task); and *imprecision*: the percentage of the total number of tasks in the LP task set that *AutomaDeD* finds that are not LP tasks; we should have only one task in the set since we inject in a single task. Figure 5.8(a)-(b) shows two cases of correct LPT detections, which should have only the one task into which we inject the error for these experiments. A singleton task appears suspicious to a user so we consider isolation as semi-successful. Figure 5.8(c) shows an example of isolation — the PDG isolates faulty task {3} (although *AutomaDeD* failed to select it as the LP task).

Table 5.3: LPT detection performance for AMG2006.

No	Function	LPT detected	Isolated	Imprecision
1	hypre_BoxGetSize	✓		0
2	HYPRE_SStructMatrixSetObjectType	✓		0
3	hypre_PCGSetup	✓		0
4	hypre_BoomerAMGSetOverlap	✓		0
5	MapProblemIndex	✓		0
6	GetVariableBox	✓		0
7	hypre_ParVectorCopy	✓		0
8	hypr_CommPkgDestroy	✓		0
9	hypre_CSRMatrixMatvec	✓		0
10	hypre_Rand	✓		0
11	HYPRE_IJVectorCreate	✓		0
12	DistributeData	✓		0
13	hypre_ParKrylovMatvec	✓		0
14	hypre_BigQsort0	✓		0
15	hypre_ParKrylovFree	✓		0
16	HYPRE_SStructGridSetExtents	✓		0
17	SetCosineVector	✓		0
18	HYPRE_SStructGridDestroy	✓		0
19	enter_on_lists	✓		0
20	IntersectBoxes	✓		0
21	MPI_Allgather	×	✓	0.99
22	MPI_Iprobe	✓		0
23	MPI_Irecv	×	✓	0.05
24	MPI_Test	✓		0
25	MPI_Waitall	×	✓	0.03

Table 5.4: LPT detection performance for LAMMPS.

No	Function	LPT detected	Isolated	Imprecision
1	Neighbor::decide	✓		0
2	FixNVE::final_integrate	×	✓	0.65
3	RanPark::uniform	✓		0
4	Atom::check_mass	✓		0
5	Neighbor::init	✓		0
6	Output::write	✓		0
7	Verlet::setup	✓		0
8	FixSetForce::post_force	✓		0
9	PairLJCut::compute	✓		0
10	Atom::memory_usage	✓		0
11	AtomVecAtomic::create_atom	✓		0
12	Domain::set_local_box	✓		0
13	Thermo::compute_vol	✓		0
14	DumpAtom::pack	✓		0
15	Region::options	✓		0.35
16	Input::lattice	✓		0
17	Thermo::init	✓		0
18	Comm::reverse_communicate	✓		0
19	Neighbor::decide	✓		0
20	Modify::initial_integrate	✓		0
21	MPI_Cart_get	×	✓	0.31
22	MPI_Allreduce	✓		0
23	MPI_Recv	✓		0
24	MPI_Scan	✓		0
25	MPI_Wait	×	✓	0.01

Tables 5.3 and 5.4 show the results of the fault-injection experiments. *AutomaDeD* detects the LP task accurately most of the time (for AMG2006, all 20 user calls and

2 MPI calls; for LAMMPS, 19 user calls and 3 MPI calls). *AutomaDeD* isolates the LP task in all cases that it is not detected.

AutomaDeD has low imprecision: 43 (out of 50) injections resulted in no incorrect tasks in the LP set, i.e., *AutomaDeD* incorrectly flagged the LP task only 7 times. In these 7 cases, *AutomaDeD* can detect the faulty task by finding the isolated task in the PDG. Only one AMG2006 case gives high imprecision (0.99) since progress dependencies are undetermined (and the PDG had only one node). Three remaining cases had low imprecision of 0.01 to 0.05. LPT detection recall is higher for user calls than MPI calls because if a task blocks in a computation region, the remaining tasks are likely to block in the next communication region, which follows the computation region in our MM with probability one and, thus, *AutomaDeD* is likely to detect the dependence. Alternatively, if a task blocks in a communication region, the other tasks likely block in another communication region, which is necessarily not an adjacent MM state so *AutomaDeD* has a lower probability of finding the LP task. Nonetheless, *AutomaDeD* isolates the faulty task in all cases that it does not correctly detect the LP task.

5.5.3 Performance

Scalability We run AMG2006 and LAMMPS with up to 32,768 MPI tasks on an IBM BlueGene/P and measure the time for *AutomaDeD* to perform the distributed part of its analysis (i.e., Steps 2–4 in its workflow). In each code, we inject an error close to its final execution phase in order to have the largest possible MM (to stress *AutomaDeD* with the largest input graph). We used BlueGene/P's SMP mode in which each node has one MPI task with up to four threads.

Figure 5.9 shows the results of these experiments. In each run, we measure three main steps: `BUILD_PDG` (Steps 2 and 3); `FIND_LP_TASK` (the first part of Step 4 in which the helper thread identifies the LP task); `OUTPUT` (the second part of Step 4, which post-processes the final PDG). In `OUTPUT`, *AutomaDeD* eliminates duplicate

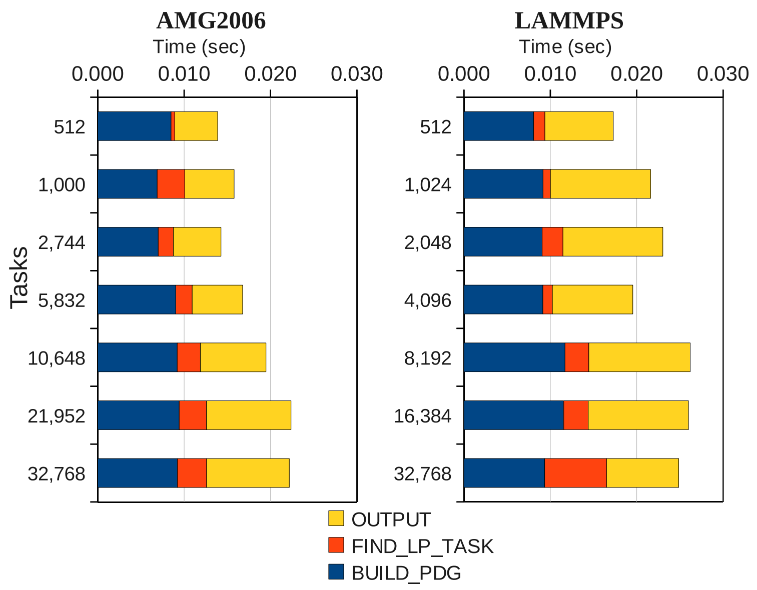


Fig. 5.9.: Time of distributed analysis (steps 2–4 in workflow) on BlueGene/P.

edges in the PDG that may result from the distributed merge processing of PDGs. It also groups MPI task ranks into ranges of the form [x–y] and adds these ranges to the corresponding PDG nodes. Figure 5.9 shows that `FIND_LP_TASK` contributes the least to the analysis overhead. Intuitively, finding the LP task is simple once we have built the PDG. `BUILD_PDG` is the core of the analysis and, so, accounts for the most overhead. However, in practice PDGs tend to be small (i.e., usually 2–10 different states) because our approach builds them probabilistically using Markov models. Thus, they do not impose intractability problems for *AutomaDeD* to analyze them and to find the LP task. Our results demonstrate the scalability of *AutomaDeD*. The distributed analysis takes less than a second on up to 32,768 MPI tasks. The low cost of this analysis suggests that we can trigger it at multiple execution points with minimal impact on the application run.

Slowdown and memory usage. Table 5.5 shows application slowdown and *AutomaDeD* memory usage for AMG-2006, LAMMPS, and six NAS Parallel benchmarks: BT, SP, CG, FT, LU and MG [48]. We omit EP because it performs almost no

MPI communication and IS because it uses MPI only in a few code locations. Since their MPI profiles produce small MMs, monitoring at the granularity of MPI calls does not suit these applications. Slowdown is the ratio of the application run time with *AutomaDeD* to the run time without it. Memory usage shows the proportional increase in program heap usage when we use *AutomaDeD* and it is calculated by the following formula:

$$\text{increase} = \frac{\text{memory-with-tool}}{\text{memory-without-tool}} \tag{5.1}$$

Since *AutomaDeD* operates as a linked library, its memory usage increases the memory usage of the tasks themselves.

Table 5.5: Slowdown and proportional increase in memory usage.

Benchmark	Slowdown	Memory-usage Increase
LAMMPS	1.59	6.11
AMG2006	1.46	10.36
BT	1.08	3.75
SP	1.67	5.14
CG	1.14	2.21
FT	1.05	1.01
LU	1.39	5.37
MG	1.04	1.04

Since tasks can have different memory usage (depending on their behavior), we used the task with the highest memory usage to calculate the increase. *AutomaDeD* incurs little slowdown – the worst is 1.67 for SP – because the overhead is primarily the cost of intercepting MPI calls and updating the MM, steps that we have highly optimized. For example, to optimize MM creation, we use efficient C++ data structures and algorithms such as associative containers and use pointer comparisons (rather than string-based comparisons) to compare states. Memory usage is moderate for most benchmarks; the largest is AMG2006 (10.36), which has many (unique) states in its execution. The factor that most affects slowdown is the number of MPI calls from different contexts since this increases the number of states that *AutomaDeD* creates. Benchmarks with slowdown of 1.3 (or more) call MPI routines from different contexts more often than the others. Applications with higher slowdown (AMG2006, LAMMPS, SP and LU) also exhibit higher *AutomaDeD* memory use due to the number of states.

5.6 Limitations

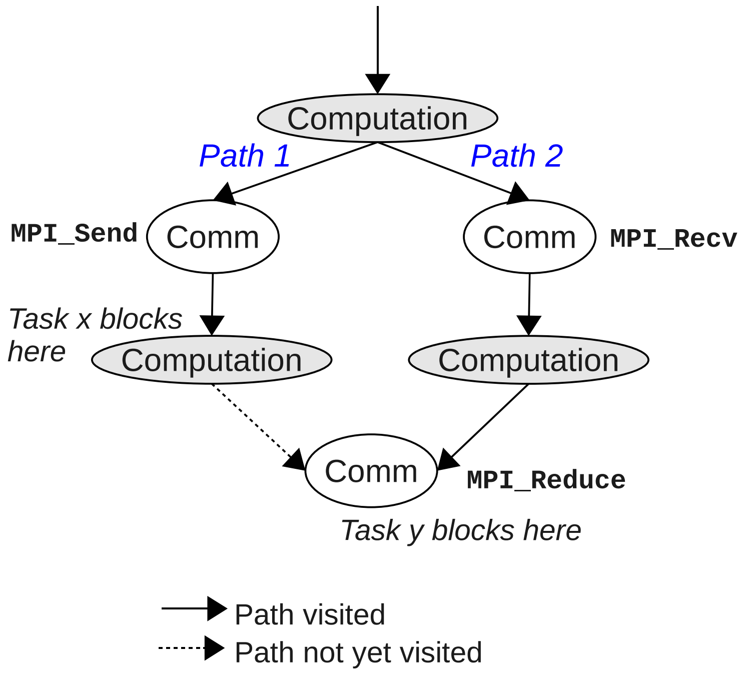


Fig. 5.10.: Fault occurring in a code with multiple paths.

A limitation of our technique is the inability to infer progress dependence between two tasks when no path between their corresponding states has been seen before a failure. To illustrate this, consider Figure 5.10 in which task x takes path 1 and task y takes path 2, and they both participate of the `MPI_Reduce` operation at the end of the figure. Assuming that it is the first time that these paths are taken, if task x blocks indefinitely before reaching `MPI_Reduce`, task y might block in `MPI_Reduce` (if it depends on x). Since there is no path between the state in which x is and the state in which y is, the corresponding path probabilities are zero and therefore the dependence is null at the end of the distributed PDG algorithm. There are two situations in which this limitation is void:

- Paths are visited within a loop and at least one iteration of the loop has occurred before task x fails. This ensures that there is at least one path between the states of tasks x and y and therefore the progress dependence can be inferred using transition probabilities in the Markov model.

- Path 1 is visited by other task different than x and y. Suppose that task z visits path 1 at least once and it reaches `MPI_Reduce` before a failure. The progress-dependence between tasks x and y will be inferred when the PDG creation algorithm ends. Two cases can occur: (1) like task y, task z blocks in `MPI_Reduce`—the algorithm infers $z \xrightarrow{pd} x$ and $y \xrightarrow{pd} x$ since z and y are in the same state; (2) task z blocks in a state that is after `MPI_Reduce`—the algorithm infers $z \xrightarrow{pd} y$ and $z \xrightarrow{pd} x$. In (2), the PDG shows both tasks x and y as least-progressed tasks which provides an accurate LP task detection but without perfect precision (since task y is not a LP task). Case (2) is very likely to occur because of the single-program-multiple-data (SPMD) nature of MPI programs.

5.7 Discussion

Our novel debugging approach can diagnose faults in large-scale parallel applications. By compressing historic control-flow behavior of MPI tasks using Markov models, our technique can identify the least progressed task of a parallel program by inferring probabilistically a progress-dependence graph. We use backward slicing to pinpoint code that could have led to the unsafe state. We design and implement *AutomaDeD*, which diagnoses the most significant root-cause of a problem. Our analysis of a hard-to-diagnose bug and fault injections in three representative large-scale HPC applications demonstrate that *AutomaDeD* identifies these problems with high accuracy, where manual analysis and traditional debugging tools have been unsuccessful. The distributed part of the analysis is performed in a fraction of a second with over 32 thousand tasks. The low analysis cost allows its use multiple times during program execution.

6. PROBLEM LOCALIZATION IN COMMERCIAL APPLICATIONS

In this section we present a system called Orion to automate the problem-localization process in commercial distributed applications. Orion models the application's run-time behavior through pairwise correlations of multiple metrics in the system, within multiple non-overlapping time windows. When correlations deviate from those of a learned correct model due to a bug, our analysis pinpoints the metrics and code regions that are most likely associated with the failure. The framework is demonstrated with real-world failure cases in four distributed applications: HBase, Hadoop DFS, a Purdue campus-wide Java application for checking availability of lab machines, and a regression testing framework from IBM. Our results show that Orion is able to pinpoint the metrics and code regions that developers need to concentrate on to fix the failures.

6.1 Introduction

Failures can come from different layers of the system—network, hardware, operating system, middleware, and application layers. So in the general case, it is necessary to monitor the behavior of all the layers to understand the origin of a failure. In this section, we focus on the problem of debugging, or root cause analysis, for failures in distributed commercial applications.

We focus on bugs that affect the control flow in a general sense, which in turn affects one or more metrics at the OS, middleware, or application layers. An example of such a failure would be a resource leak prior to the application hanging. We treat performance anomalies also as one kind of failure. Among bugs that we do not handle are bugs that lead to data corruption or failures that do not affect a system-measurable metric.

Debugging for distributed applications. Research on distributed systems has developed foundational techniques that can help in debugging, such as program tracing and replay debugging [83–85], model checking [23,24,27], and log analysis [28–31]. But there remains work to be done to build on these foundational techniques to create a usable debugging tool specifically for distributed applications. Debugging in distributed applications brings in the additional aspect that a problem may be related to a problem either in the software code that is executing on this node or on a communicating node or network problems. This is precisely the goal that we have in this research piece — to build a usable debugging tool for distributed applications. Our goal is that the tool should indicate a region in code where the fault first becomes active. This point is prior to the fault manifesting itself as a user-visible failure and is more likely to be close to the region of code where the bug lies. From a definition standpoint, we have two concepts — first, is a fault becoming activated and the second, fault becoming an externally visible failure. The fault becomes activated when say the buggy software code is executed or the unavailable network service is accessed. The effect of this is not immediately visible externally, say to a user. When that happens, we say that the fault has metastasized into an externally visible failure. The tool should also not need much domain knowledge or knowledge of the kind of fault that can occur. Thus, we would rule out an approach where the region of code or the metrics to examine would need to be identified manually prior to the tool being deployed.

ORION***: Design approach***. We present ORION, a software framework for localizing the origin of problems in distributed applications and a specific instantiation of the framework[1]. ORION works, simply put, by profiling a wide variety of metrics as the application is executing, either at declared instrumentation points (such as, method entry or exit) or asynchronously with a fixed periodicity. Then through ma-

[1]ORION is the Greek God for hunting, albeit not for bugs.

chine learning techniques it verifies if the runtime profile is *similar enough* to profiles created offline of non-faulty execution of the application. If it is not, then Orion goes back through the logs in an offline analysis phase to indicate which metrics caused the divergence of the profiles and from that, to the region of suspect code. The mechanism is probabilistic and ultimately a rank-ordered list is provided to the developer for inspection and possible fixing. This design approach is shared with a few prior software systems [35, 86]. However, unlike these prior systems, which only gather traces from one or two dimensions of the application, e.g., CPU and memory, Orion performs application profiling along a large number of metrics, which do not have to be hand-picked by the developer. Further Orion uses correlations between the metrics to diagnose subtle errors, which would not be caught by the predominant approach of checking if a metric goes above or below a constant threshold value [87, 88]. We term our approach *multi-dimensional profiling*. We have found that, in our case studies, it would be difficult to *a priori* determine the one or two performance indicators and specific code points to examine for a bug, as most existing work assumes.

Summary of findings. We deploy Orion and evaluate it with four diverse applications. The first two are Java-based open source applications that mimic elements of the Google software stack — HBase [89] and Hadoop [90]. The third is an on-campus Java Enterprise Edition (JEE) application, called StationsStat, that is used by students to check availability and load of computers throughout the Purdue campus labs. The fourth is an IBM regression testing application for its full system simulator called Mambo. In all these applications, we focus on failures that were difficult to debug manually. In all these cases, we find that the root cause is related to the top three abnormal metrics or code regions given by Orion.

6.2 Overview

6.2.1 Measurement Gathering

System administrators use Orion to detect software bugs, performance anomalies or unexpected runtime condition that affect end users. Orion collects measurements of multiple metrics at different levels in the system, e.g. operating-system-, middleware- and application-level by means of third-party monitoring tools, such as, the `/proc` file system for gathering operating-system process-level metrics. Using collected measurements, Orion builds models of normal behavior, which allows the localization of the failures origins.

We gather measurements from multiple layers in the system to have a full knowledge of the application's behavior. Our approach considers a wide range of metrics (see Table 6.1 for the full list) and automatically zooms into the metrics that are relevant to the failure and for the particular window in time where the fault first becomes manifested. Some examples of the metrics that we gather at the different levels are:

- **End-user Application:** per-servlet statistics such as processing time, request and exception counts.
- **Middleware:** cache hits and accesses, number of busy and created threads, and request processing time from the middleware layer (such as Apache Tomcat).
- **OS:** cpu- and memory-usage, context switches, file descriptors, disk reads/writes (KB), packets received and transmitted, stack size.

Notice that Orion's algorithm is not restricted to only these metrics—a user can use any metrics in the analysis if she thinks a problem could be related to them. Our algorithm for suspicious metric selection eliminates metrics that provides less insight in the problem localization process and highlights only the metrics that are most strongly associated with the problem.

6.2.2 Metrics

Table 6.1 shows the list of metric types that we analyzed, grouped by layer. For the Hadoop, HBase, and Mambo Health Monitor, we use only OS metrics. Notice that the list is not the *number* of metrics that we actually analyzed because some metrics types have multiple instances. For example, the `data_source_numActive` is the number of active connections per database. The StationsStat system has two databases so it has two instances of this metric. The same occurs for Java servlets, containers and server metrics—a server can have multiple containers and database connections while each container can have multiple servlets. The OS metrics are based on statistics of a Linux process—if a Java server only creates one Linux process, it would have only one instance of each of the OS metrics.

6.2.3 Profiling

Orion can perform multi-dimensional profiling in two ways: *synchronous* and *asynchronous*. In the asynchronous mode, the metric collection (*aka*, profiling) happens asynchronously to the application. The measurement gathering is done by a process separate from the application process(es). The asynchronous mode does not interfere directly with the monitored application and therefore is a lightweight method. In a typical asynchronous profiling session, Orion collects OS metrics values from the Linux `/proc` file system and middleware- and application-metrics values from server containers by querying Java JMX connectors (such as with the Apache Tomcat JMX connectors) via a separate Linux process. This method requires offline processing to "line up" the metric collection points with the execution points of the application and that is simply done by using the common time base, since all the involved processes execute on the same machine.

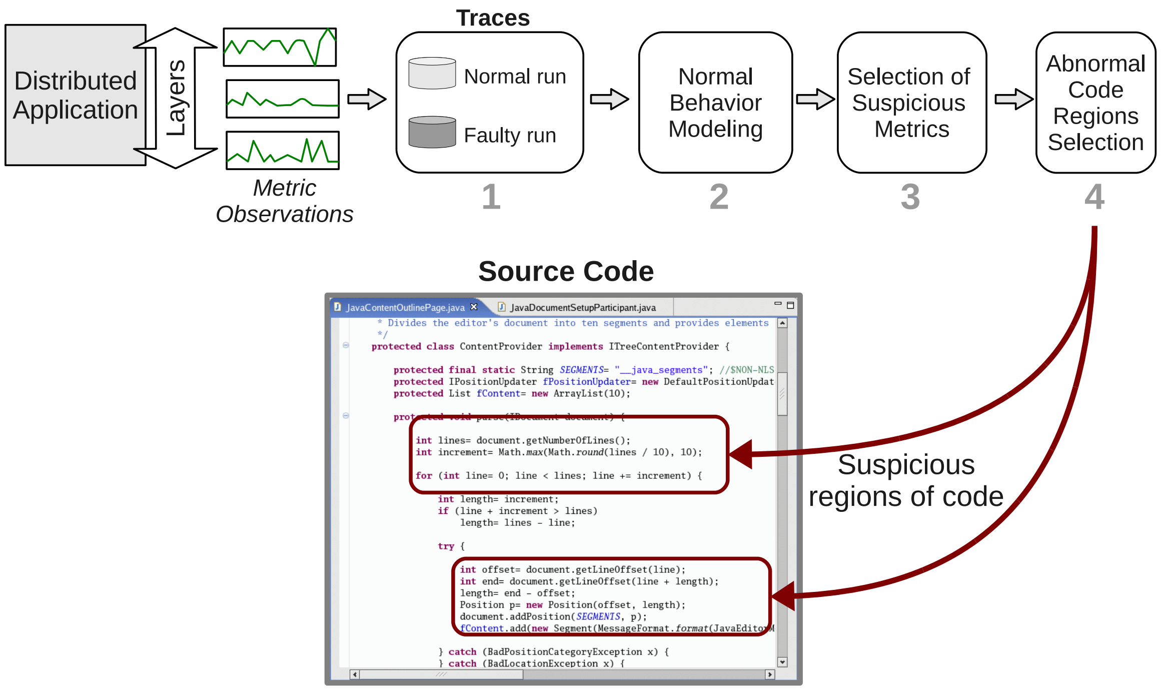
Synchronous profiling provides a mechanism for automatically annotating the code region with a set of measurements corresponding to the code region. Whenever a code region begins and ends, this method collects metric measurements and labels them

Table 6.1: Metric descriptions.

Metric	**Description**
Operating System	
minor_faults	minor page faults
major_faults	major page faults
utime	user-level CPU time
stime	system-level CPU time
num_threads	number of threads
vsize	virtual memory size
rss	RAM memory
processor	CPU number last executed on
stack_size	size of the stack
rchar	read characters from disk
wchar	written characters to disk
read_bytes	read bytes from disk
write_bytes	written bytes to disk
canceled_write_bytes	canceled written bytes to disk
num_file_desc	open file descriptors
nicRcvBytes	received bytes from NIC
nicRcvPckts	received packets from NIC
nicSentBytes	sent bytes from NIC
nicSentPckts	sent packets from NIC
IPInTruncatedPkts	truncated IP packets
IPInOctets	received IP octets
IPOutOctets	sent IP octets
Application	
servlet_processingTime	processing time (per servlet)
servlet_maxTime	max processing time (per servlet)
servlet_requestCount	requests (per servlet)
servlet_errorCount	errors (per servlet)
datasource_maxWait	max waiting time (per database)
datasource_numIdle	idle connections (per database)
datasource_maxActive	max active time (per database)
datasource_numActive	active connections (per database)
Middleware	
request_handler_bytesSent	bytes sent (per container)
request_handler_bytesReceived	bytes received (per container)
request_handler_requestCount	requests(per container)
request_handler_maxTime	max processing time (per container)
request_handler_processingTime	processing time (per container)
request_handler_errorCount	errors (per container)
cache_hits	cache hits (per server)
cache_accesses	cache accesses (per server)
number_threads	active threads (per thread pool)

with the corresponding code region name. For Java applications (such as in the Hadoop and HBase case studies), we use `Javaassist` to instrument binary code and to collect measurements at the beginning and at the end of the classes/methods we are interested in.

6.2.4 Workflow of our Approach


Distributed Application
Layers
Metric Observations
Traces
Normal run
Faulty run
1
Normal Behavior Modeling
2
Selection of Suspicious Metrics
3
Abnormal Code Regions Selection
4
Source Code
JavaContentOutlinePage.java
JavaDocumentSetupParticipant.java
Suspicious regions of code

```
     * Divides the editor's document into ten segments and provides elements
     */
    protected class ContentProvider implements ITreeContentProvider {

        protected final static String SEGMENTS= "__java_segments"; //$NON-NLS
        protected IPositionUpdater fPositionUpdater= new DefaultPositionUpdat
        protected List fContent= new ArrayList(10);

        protected void parse(IDocument document) {

            int lines= document.getNumberOfLines();
            int increment= Math.max(Math.round(lines / 10), 10);

            for (int line= 0; line < lines; line += increment) {

                int length= increment;
                if (line + increment > lines)
                    length= lines - line;

                try {

                    int offset= document.getLineOffset(line);
                    int end= document.getLineOffset(line + length);
                    length= end - offset;
                    Position p= new Position(offset, length);
                    document.addPosition(SEGMENTS, p);
                    fContent.add(new Segment(MessageFormat.format(JavaEditorM

                } catch (BadPositionCategoryException x) {
                } catch (BadLocationException x) {
```

Fig. 6.1.: Overview of problem determination workflow in ORION.

Figure 6.1 shows the steps in ORION to diagnose failures:

1. **Trace collection:** ORION uses two set of traces to localize the origin of problems: a *normal* and an *abnormal* trace file. The normal trace file is obtained by collecting metrics of the application when failures are not manifested. This can be the case of runs of a earlier bug-free application version or, in the case of intermittent failures, sections of a failed run where the fault did not manifest itself. The abnormal trace file is obtained when the failure is manifested externally. Labeling a run as one or the other is a manual process. In practice, we find that the failure manifestation is quite obviously detectable and therefore the labeling is not difficult.

2. **Normal-behavior modeling:** ORION creates a baseline model using the normal-behavior traces. Given traces of n metrics, the algorithm splits traces

into equally sized time windows and calculates pairwise correlations between all the n metrics. These correlation serve as a summary of the expected behavior of the application in different time windows. We observed that there is not a single point for normal behavior, since there are different workload patterns, and therefore our model captures a region rather than a single point.

3. **Suspicious metric selection:** The file with abnormal traces is used to select the metrics that are correlated with a failure. From all the n original metric, the top 3 most abnormal metrics are presented to the user (ranked by an abnormality measure). The user can then focus on finding the origin of the problem based on the abnormal metrics. For example, in our case study with the Purdue application StationsStat, identifying the suspicious metric as the number of SQL connections mapped directly (with human intervention) to the part of the SQL driver that handled SQL connections.

4. **Abnormal code-region selection:** There are cases when the suspicious metric information in step 3 is not sufficient to infer the origin of a failure. For example, a metric like CPU utilization may be affected by any region of code, even when ORION has identified a window in time. ORION selects the code regions that have a high degree of association with the suspect metric(s). This is done by adding annotations in the original trace files with the code regions that are executed for each measurement collection. ORION then highlights suspicious code regions so that the user can focus on finding bugs that could have caused the problems within them.

6.3 Design

6.3.1 Modeling Sequential Data

Many bugs and performance anomalies develop a characteristic temporal pattern that can only be captured by analyzing measurements of metrics in a sequential manner (rather than by observing instantaneous snapshots of values). Using the set of normal traces, we build a baseline model that captures temporal patterns between metrics using correlation coefficients.

Observation window. Traces are split into non-overlapping windows of the same size. A window can be viewed as a matrix $S \times N$ in which S is the number of records (or samples) and N is the number of metrics. The set of records comprises one *observation window.* Since we do not know *a priori* the optimal size of observation windows (S), i.e., the window size that is sufficient to capture the temporal patterns that a failure shows, our algorithm sweeps through multiple sizes for the windows within a range. The algorithm then finds the k-most abnormal windows (irrespective of its size) and, within those abnormal windows, the correlations and metrics that cause the unusual patterns. For our evaluation, we use a k value of 3. Section 6.3.2 describes our algorithms for the selection of suspicious metrics and code regions.

Correlation coefficients. For each observation window, ORION builds a vector of (pair-wise) correlation coefficients between all the metrics:

$$CCV = [cc(1,2), cc(1,3), \ldots, cc(N-1,N))], \tag{6.1}$$

where $cc(i,j)$ is the correlation coefficient of metrics i and j, $i \neq j$. We denote this vector as a *correlation coefficient vector* or CCV. Correlation coefficients are calculated using the *Pearson correlation-coefficient* formula:

$$cc(X,Y) = \frac{1}{N-1} \sum_{k=1}^{N} \left(\frac{X_k - \bar{X}}{s_X} \right) \left(\frac{Y_k - \bar{Y}}{s_Y} \right) \tag{6.2}$$

where N is the number of elements of observations in the window, $\bar{X}$ and $\bar{Y}$ are the mean of variables X and Y respectively, and s_X and s_Y are the standard deviations of X and Y.

Normal-behavior Model. Using the normal-behavior traces, our framework creates a baseline model which is used in step 3 (from the main workflow) to select suspicious metrics. The model is a set of normal-behavior CCVs obtained by splitting

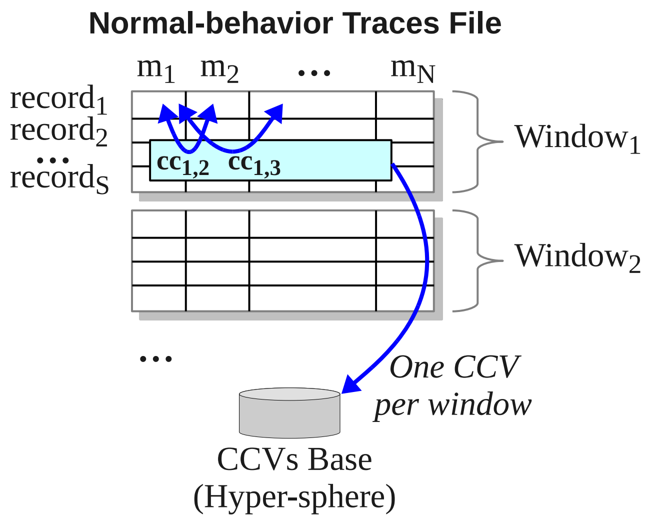


Fig. 6.2.: Steps in the creation of the normal-behavior hyper-sphere.

traces into observations windows and computing a CCV for each window. We term this model as a *hyper-sphere*. Figure 6.2 shows the process of creating this hyper-sphere. The number of points in it corresponds to the number of windows that we obtained from the normal-behavior traces, and the dimensions (or features) are correlation-coefficients of metric pairs. Notice that, if we have N metrics in the analysis, the dimension of the hyper-sphere is $D = \frac{N(N-1)}{2}$.

The idea of using a hyper-sphere where dimensions are correlation coefficients is that we can use a nearest-neighbor approach to pinpoint abnormal observation windows from the faulty traces. Since an observation window is translated to a single data point (i.e., a CCV), we can treat the problem of finding abnormal windows as an outlier detection problem via nearest-neighbor, i.e., an abnormal window would correspond to the CCV point that is the farthest away from the hyper-sphere.

6.3.2 Detection of Suspicious Metrics

Motivation. The main motivation of our technique is that the manifestation of bugs and performance faults will change the correlations between some (affected) metric(s) and the rest of the metrics, while maintaining the legitimate correlations in the other metrics. To illustrate this idea, consider a bug where unused database connections are kept open—metrics such as file descriptors and open sockets will be affected by the bug and will exhibit a different temporal pattern than during workloads where the bug is not activated. However, correlations among the other metrics will not be affected.

Our goal is that, when faults are manifested, ORION finds the metric(s) that is (are) mostly associated with failures. This is performed by ranking metrics according to their contributions to correlation breakups and by selecting the top-k metrics in this ranking. The application developer can subsequently focus on reviewing the code which affects these suspicious metrics to locate the root cause of the problem.

Algorithm overview. The goal of our metric selection algorithm is to show to the user the metrics that are most likely associated with the origin of a problem. The input of the algorithm is a normal-behavior hyper-sphere and traces of a failed run. The output of the algorithm is a list of metrics that are ranked by abnormality degree. The algorithm is presented in Figure 6.3.

The algorithm has the following main steps:

Statistics creation per window: We create observation windows of multiple sizes from the failed-run traces file. For each window, we calculate two statistics: (1) the *nearest-neighbor* (NN) distance of the window from the hyper-sphere representing normality. This distance is calculated by first computing a CCV from the window and then by finding the euclidean distance between the CCV and the closest point in the hyper-sphere using the formula $d = \sqrt{\sum_{i=1}^{D}(cc_i - bb_i)^2}$, where cc_i and bb_i are correlation coefficients; (2) the dimensions that have the highest weight in making the CCV far away from the hyper-sphere. A dimension here corresponds to a correlation coefficient.

Abnormal window selection: Windows are sorted by their NN distance from high to low and only the top-k windows in the list are taken for further analysis ($k = 3$ for our evaluation). These windows correspond to time periods when abnormal behavior

```
/* Get statistics of failed-run windows */
for each size $s$ in range $r$:
  $setOfWindows$ ← create windows set of size $s$
  for each window $w$ in $setOfWindows$:
    $d$ ← find NN distance of $w$ in the hyperSphere
    $ccs$ ← get top abnormal corr. coefficients of $w$
    Append {$w$, $d$, $ccs$} to $tuplesList$

/* Select the most abnormal windows*/
Sort $tuplesList$ based on distance $d$ (high to low)
$abnormalWindows$ ← get top elements in $tuplesList$

/* Build histogram of most abnormal metrics */
for each window $w$ in $abnormalWindows$:
  for each correlation $cc$ corresponding to $w$:
    From $cc$ add metrics X and Y to $histogram$

Print the most frequent metrics in $histogram$
```

Fig. 6.3.: Algorithm to select the suspicious metrics from traces of a failed run.

is manifested.

Select most abnormal metrics: Once the top-k abnormal windows are ranked, within each window, the correlation coefficients (CCs) are ranked by how much they contribute to the NN distance of that window. Now the top-k CCs are taken from each abnormal window, giving a total of $k \times k$ CCs. Recall that each CC involves two metrics. With these short-listed CCs, the metrics that are present in them are counted up and the top-k most frequently occurring metrics are flagged as the most abnormal metrics. The careful reader would have noticed that we are using the same parameter for filtering the top choices (windows, CCs, metrics). In theory, they are different parameters, but in practice the same value ($k = 3$) works well and reduces the search space of parameters, a desirable outcome for any deployable tool.

Window sizes. It is difficult to determine without domain knowledge, what the optimal size of the observation window should be. Therefore, ORION sweeps through a range of observation window sizes (between sizes of 100 and 200 samples in our evaluation). It then uses the NN distance of all these windows of various sizes to do its abnormal window determination as described above.

6.3.3 Detection of Anomalous Code Regions

Motivation. Often for performance and correctness bugs, knowledge of the metrics that went awry is not sufficient for fixing the problem—developers still need to look for the problem origin within the code lines of the program. After the suspicious metrics are detected, ORION highlights code regions that make this metric abnormal so that developers can focus on them to fix the problem. ORION first finds suspicious periods of time in which a metric shows an unusual temporal pattern (i.e., an abnormal window). Then within that period, ORION looks for outlier observations, i.e., an abnormal code region. Figure 6.4 shows the general idea behind it.

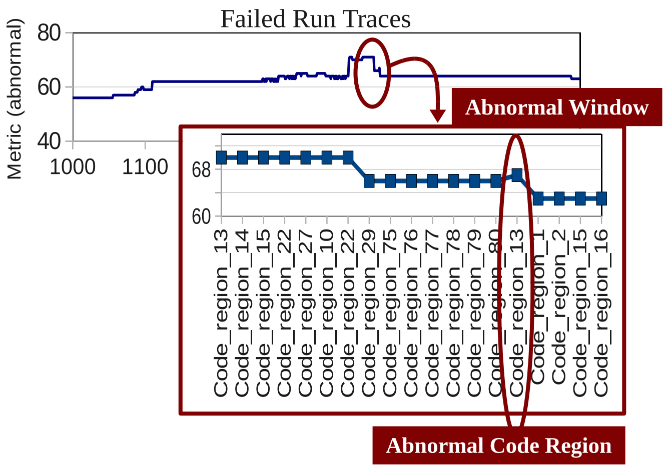


Fig. 6.4.: Example of the steps in detecting the abnormal code region.

Algorithm requirements. The algorithm for detecting abnormal code regions is similar to the previous algorithm for selecting the abnormal metrics. A major difference is that only one metric is used from the traces files for the analysis, i.e., the abnormal metric. The user can opt to execute this algorithm using the top-two (and top-three and so on) abnormal metric(s) if the top-one abnormal metric doesn't help in finding the problem origin. The input of the algorithm are traces files (from the normal and the failed run) such as in Figure 6.2 but with only one column—this column corresponds to measurements of the abnormal metric. We assume that each record in this file is annotated with a code region.

Comparing one-dimensional windows. In the suspicious-metric selection algorithm, we find the difference between two windows (normality and the failed run that we are debugging) by calculating the Euclidean distance of their corresponding *CCVs*. Here we have a single metric and we summarize the information of the window by calculating aggregates of its values: *average*, *standard deviation*, *minimum*, *maximum* and *sum*. These aggregates become the features of a window. ORION then finds the dissimilarity between two windows by computing the euclidean distance using these aggregates as the features.

```
/* Get statistics of failed-run windows */
for each size $s$ in range $r$:
  $normalWins$ ← get windows from normal traces
  $failedRunWins$ ← get windows from abnormal traces
  for each window $w$ in $failedRunWins$:
    $d$ ← NN distance of $w$ from $normalWins$
    Append {$w$, $d$} to $tuplesList$

/* Select the most abnormal windows*/
Sort $tuplesList$ based on distance $d$ (high to low)
$abnormalWindows$ ← get top elements in $tuplesList$

/* Build histogram of abnormal code-regions */
for each window $w$ in $abnormalWindows$:
  Add code regions in $w$ to $histogram$

Print the most frequent code regions in $histogram$
```

Fig. 6.5.: Algorithm to select the suspicious code regions from traces of the failed run.

Algorithm overview. The algorithm is shown in Figure 6.5. First, we construct a set of windows (of different sizes) from traces of the normal run and another set from traces of the failed run. Second, we find nearest-neighbor distances of the windows of the failed-run from the normal-behavior windows. Then, to select the most abnormal windows, we rank the failed-run windows based on the NN distances (from high to low) and select only the top-k windows. Finally, we build a histogram of the occurrences of code regions in these abnormal windows. The idea behind this histogram is that, as we observe from our case studies, the faulty code regions in performance bugs execute frequently in the most unusual periods of time. The top-three most frequent code regions are shown to the user.

6.4 Evaluation

We demonstrate ORION in four types of large-scale distributed applications. Our goal is to show how ORION can be applied to complex code bases to diagnose a variety of real-world failures. Table 6.2 summarizes the case studies that we considered in our evaluation. In all of the cases, ORION reduces substantially the time spent in localizing the problem origin by showing the metric most perturbed by the fault and if needed further, the code region corresponding to the above metric. The process is fully automated so users do not need to have full understanding of the application and its components dependencies.

Table 6.2: Summary of case studies in ORION's evaluation.

Application	**Language**	**Fault**	**Problem Manifestation**	**Localization scope**	**Metric collection**
Hadoop	Java	File descriptor leak	Application crashes	Metric & code region	Synchronous
HBase	Java	Deadlock	Application hangs	Metric & code region	Synchronous
StationsStat	Java	SQL driver bug	Application is unresponsive	Metric	Asynchronous
IBM MHM	Tcl	NFS connection is broken	Simulation fails	Metric & code region	Asynchronous

6.4.1 Case 1: Hadoop DFS

Hadoop is an open-source framework that supports data-intensive distributed applications [90]. It enables applications to work with thousands of computational nodes and a large amount of data. We use Orion to diagnose a file descriptor-leak bug that occurred in the Hadoop Distributed File System (HDFS) in version 0.17. The bug report is HADOOP-3067.

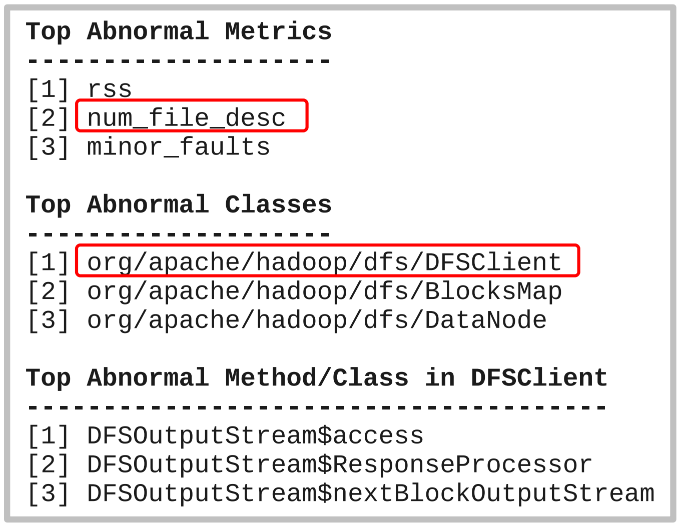
For this case study, we collected all the OS level metrics given in Table 6.1. We used synchronous profiling to obtain the traces. All the Java classes and all the public methods within each class are instrumented. Since we are debugging the Hadoop DFS, we only consider the `java/org/apache/hadoop/dfs` package. A total of 45 different Java classes and 358 methods within these classes are instrumented.

```
In DFSInputStream:
} catch (IOException e) {
     ioe = e;
     LOG.warn("Failed to connect to...");
+    } finally {
+    IOUtils.closeStream(reader);
+    IOUtils.closeSocket(dn);
+    dn = null;
+    }

In BlockReader (used by DFSOutputStream):
-  return super.read(buf, off, len);
+  int nRead = super.read(buf, off, len);
+  if (nRead >= 0 && needChecksum() && ...) {
+  checksumOk(dnSock);
+  }
+  return nRead;
```

Fig. 6.6.: Sample section of the pacth to fix the HDFS bug.

This bug is manifested as a failure in one of the HDFS tests (the `TestCrcCorruption` test.) The bug origin is that subclasses `DFSInputStream` and `DFSOutputStream` of the main class `DFSClient` did not handle open sockets correctly by not closing them when they are not used anymore. Some sections of the patch that developers used to fix this bug are shown in Figure 6.6. The patch included changes to the following code: class `DFSClient`, and subclasses `DFSInputStream` and `BlockReader` (which is used internally by `DFSOutputStream`). We used the buggy version and revision (0.17) of the code to obtain traces of a failed run, and code from a previous revision where the `TestCrcCorruption` test passed to get traces of a normal run.



```
Top Abnormal Metrics
--------------------
[1] rss
[2] num_file_desc
[3] minor_faults

Top Abnormal Classes
--------------------
[1] org/apache/hadoop/dfs/DFSClient
[2] org/apache/hadoop/dfs/BlocksMap
[3] org/apache/hadoop/dfs/DataNode

Top Abnormal Method/Class in DFSClient
--------------------------------------
[1] DFSOutputStream$access
[2] DFSOutputStream$ResponseProcessor
[3] DFSOutputStream$nextBlockOutputStream
```

Fig. 6.7.: Results from ORION for the HDFS bug.

Figure 6.7 shows ORION's results. The top-three abnormal metrics presented by ORION are: (1) `rss` (resident set size–non-swapped physical memory), (2) `num_file_desc` (number of open file descriptors), (3) `minor_faults` (number of minor page faults). The 2nd metric is associated with problem origin since an increase in the number of open sockets caused by the bug affects directly the number of open file descriptors. Metrics (1) and (3) are both memory related and we believe they are (erroneously) pinpointed as suspicious because the bug also causes abnormal patterns of memory consumption. We speculate that this happens because the leaked socket descriptors are using up memory and some non-linear effect kicks in when the leak gets large enough. However, we have not confirmed this through detailed targeted tests.

ORION also presents abnormal code regions, first, based on Java classes and second, based on subclasses (of the abnormal classes) and methods within them. ORION correctly pinpoints `DFClient` as the most abnormal class. Within `DFClient`, ORION highlights `DFSOutputStream` as the main abnormal subclass. This is only partially correct — part of the bug fix is in `BlockReader` which is used internally by `DFSOutputStream` and `DFSInputStream`; however, `DFSOutputStream` does not require changes to fix the bug.

Table 6.3: Average use of file descriptors per class in HDFS bug.

Rank	**Class**	**Average # File Descriptors**
1	NamespaceInfo	6.0
2	INodeDirectory	1.31
3	INode	1.29
4	UnderReplicatedBlocks	1.25
5	DatanodeInfo	1.24
6	DataNode	1.21
7	DatanodeBlockInfo	1.2
8	DFSClient	1.16
9	DataBlockScanner	1.14
10	NameNode	1.13

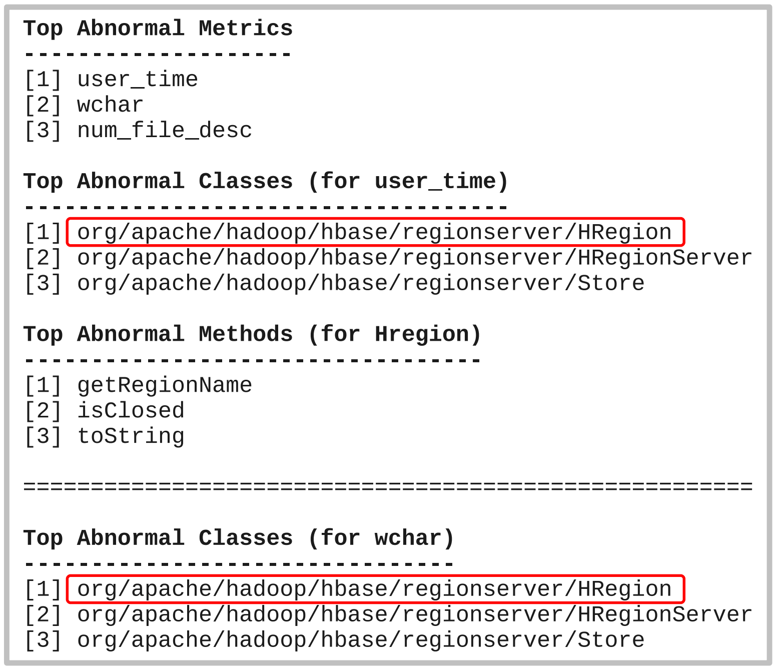
To see if a simpler, and currently practiced, approach can lead the developer to the origin of the bug faster, we set up the following hypothetical steps for hunting this bug. Suppose that a simple profiling tool indicates a high number of file descriptors in use. The developer then proceeds to examine which classes use file descriptors most. The answer to this question is shown in Table 6.3. Average number of file descriptors used within the method is calculated by taking an average across all invocations of the methods of that class. From this, the developer would be likely to inspect the classes appearing near the top. It is only when one gets to the 8th ranked class that one gets to the class where the bug lies, `DFSClient`. Thus, this will lead to significant time manually inspecting classes 1–7 and ruling them out as the source of the bug.

6.4.2 Case 2: HBase

HBase is an open-source, distributed, column-oriented database that is modeled after Google's BigTable [89]. It operates on top of distributed file systems like the HDFS and is designed with the capability of processing very large scale of data with MapReduce. We use Orion to collect and analyze the metrics of a deadlock bug in HBase 0.20.3. The bug report is HBASE-2097. For this case, we collect all the OS-level metrics shown in Table 6.1. There are 27 java classes that are instrumented from the `hadoop/hbase/regionserver/` package to collect the traces. They include classes to handle region columns, store data files, logs and many other abstractions. These classes include 184 methods which are all instrumented.

The bug is the result of two locks being acquired in an incorrect order. The same bug is there in two methods `HRegion.put` and `HRegion.close`. The manifestation of the bug is an application's hang. Curiously, the bug is introduced in one of the previous patches. It is activated by running the HBase `PerformanceEvaluation` testing tools in standalone mode. This tool is used to evaluate HBase's performance and its scalability as servers are added. The bug manifestation is intermittent—it manifests on average 75% of the time—making it particularly difficult to localize.

We ran a previous version, where `PerformanceEvaluation` consistently succeeded, to generate normal behavior traces and applied Orion against the traces of a failed run when the deadlock manifests. Figure 6.8 shows the results. The top abnormal metric, i.e., `user_time`, is the amount of CPU time used by the task while executing at the user level. This metric *per se* does not provide much insight into the failure origin since it is difficult to correlate that to a code region. We observe that, using `user_time` as our abnormal metric, the most abnormal code region is `HRegion` class, which is in fact where the bug lies. Further, considering the 2nd most abnormal metric `wchar`, an I/O counter that represents the number of bytes which the program has caused, or will cause to be written to disk, the flagged code region is also the



```
Top Abnormal Metrics
--------------------
[1] user_time
[2] wchar
[3] num_file_desc

Top Abnormal Classes (for user_time)
------------------------------------
[1] org/apache/hadoop/hbase/regionserver/HRegion
[2] org/apache/hadoop/hbase/regionserver/HRegionServer
[3] org/apache/hadoop/hbase/regionserver/Store

Top Abnormal Methods (for Hregion)
----------------------------------
[1] getRegionName
[2] isClosed
[3] toString

=======================================================

Top Abnormal Classes (for wchar)
--------------------------------
[1] org/apache/hadoop/hbase/regionserver/HRegion
[2] org/apache/hadoop/hbase/regionserver/HRegionServer
[3] org/apache/hadoop/hbase/regionserver/Store
```

Fig. 6.8.: Results from ORION for the HBase bug.

`HRegion` class. This confirms that `HRegion` is where the developer needs to focus her attention.

The bug patch shows that in fact the bug resides in the `HRegion` class. This class stores data for a certain region of a table and all columns for each row—a given table consists of one or more HRegions. The patch flips the order of acquiring the two locks (a write lock and then a read lock) and consequently the order of releasing them. It puts the change in both the `HRegion.put` and the `HRegion.close` methods. We speculate that spinning on locks in the deadlock situation causes the `user_time` metric to go awry.

The abnormal methods within `HRegion` that are flagged by ORION are `getRegionName`, `isClosed` and `toString` (Figure 6.8). They do not correspond to the methods where the bug lies (`put` and `close`). Through a detailed investigation, we identify the cause of this error of ORION. The three flagged methods are invoked much more often within `HRegion` than are the erroneous methods. However, the three flagged methods and the erroneous methods occur close together in time. The algorithm in Orion, after it has zoomed into a time window where the fault manifested itself, considers frequencies of methods within that suspect time window to decide which methods to flag. This causes it to flag the most frequently invoked `getRegionName`, `isClosed` and `toString` methods.

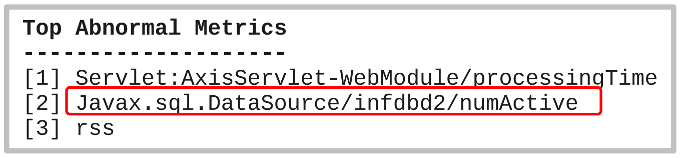
6.4.3 Case 3: StationsStat

StationsStat is a Java multi-tier application that is used to check the availability of workstations on Purdue's computing labs. Students across the campus use StationsStat on a daily basis to check the number of available Windows or Mac workstations for each lab on campus. StationsStat is managed by Purdue's IT department (ITaP) and runs in Apache Tomcat 5.5 on a RedHat Enterprise Linux 5 virtual-machine with a 2.8GHz CPU and 4GB RAM, with an in memory SQL database.

Due to an unknown bug, periodic failures were observed in which the application became unresponsive. System administrators received failure reports through alerts of their monitoring system, Nagios, or from user phone calls reporting the problem. Since the problem root cause was unknown, the application was restarted and the problem appeared to go away temporarily.

StationsStat's administrators tracked 495 metrics from the OS, middleware, and application layers at 1 minute intervals (using asynchronous profiling) for more than two months. Table 6.1 shows all the metric types that are collected.

StationsStat was a challenging scenario for Orion, not only because the problem's root-cause was unknown, but also because there was no error-free data available to create the normal-behavior hyper-sphere. Fortunately, Orion can still work in this scenario by using *almost error-free* data. The administrators noticed that, after restarting StationsStat, the next failure was often seen only after a week or more—symptoms seemed to suggest that the problem, possibly a resource exhaustion bug, grew progressively from a service restart to a failure. Orion therefore used a data segment collected right after a restart to build the hyper-sphere representing normality. ORION also filtered out constant metrics in this phase which resulted in 70 non-constant metrics that it used in the rest of the analysis.



```
Top Abnormal Metrics
--------------------
[1] Servlet:AxisServlet-WebModule/processingTime
[2] Javax.sql.DataSource/infdbd2/numActive
[3] rss
```

Fig. 6.9.: Results from ORION for the StationsStat case.

In contrast with the previous cases, here we conducted a blind experiment in which ORION gives us the rank ordered suspicious metrics without us knowing the actual root-cause of the problem. Figure 6.9 shows the abnormal metrics that ORION finds. We then compared ORION's answer to the application developers best guess of the root-cause. The results show that the suspicious metrics given by ORION matched well with what the developer thought to be the origin of the problem. The 2nd metric flagged by ORION is the number of active SQL connections to one of the databases. The application had only one localized region where it made calls to the SQL driver that Tomcat used to handle database connections. The developer concluded that the SQL driver code was buggy since it was obvious that the few lines of SQL driver invocation code in his application was not. Sure enough, upgrading the SQL driver fixed the problem and the application continues to run today (under the name "AvailableStations") providing an important function to students all over campus. Interestingly, the top metric flagged by ORION—the processing time of a servlet— had nothing to do with the bug. On further investigation, we find that this is due to large differences in workload between our normal and abnormal data sets. The normal data set is collected right after the server restart, while the abnormal data set is collected after the server has been in operation for a while. This negative result points to the importance of getting the normal and the abnormal data sets under similar workload conditions.

In this case study, there was no need for the additional step of ORION where it maps the abnormal metric to a code region. This is because only one small localized region of the application code had anything to do with SQL, which was implicated by the metric.

6.4.4 Case 4: Mambo Health Monitor

The Mambo Health Monitor is a continuous regression test system for the IBM Full System Simulator, commonly known as Mambo. Mambo is a computer architecture simulator for systems based on IBM's Power(TM) architecture. Mambo has been used in the development and testing of a wide range of systems, including IBM's Power line of server systems (Power5, Power6, Power7), the Cell(TM) processor used in the Sony Playstation3(TM), and IBM's BlueGene supercomputing systems. The Mambo Health Monitor (MHM) executes tests on the simulator to detect regressions in behavior that may be introduced by new development. The tests are drawn from a large test suite that covers the key functionality in all the major target systems supported by Mambo. Test results are stored in an SQL database and are accessed through a web-based interface that offers numerous summary and detail display formats. The MHM is similar in concept to a number of continuous regression packages, such as Hudson (`http://hudson-ci.org/`), but was written from scratch to serve the specific needs and environment of Mambo development. Figure 6.10 shows the main elements of the system.

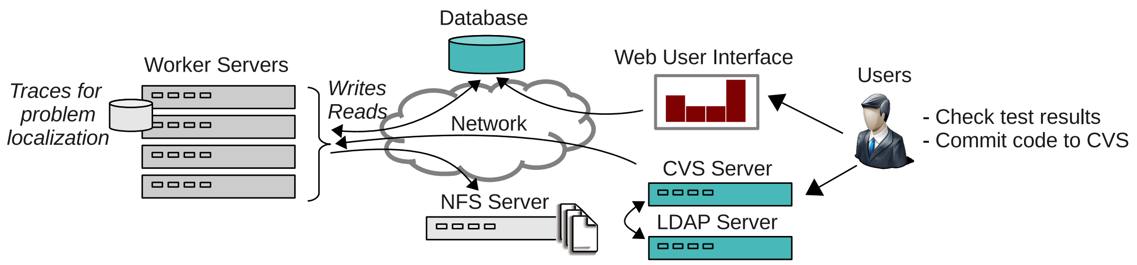


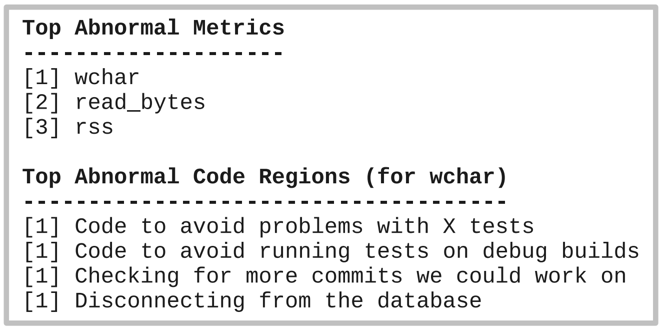
Fig. 6.10.: Mambo Health Monitoring system.

Failures. MHM runs test cases and reports results into a common shared database. There is a farm of servers which serve as Mambo Health Monitor clients. Each client accesses a database to determine which tests for which version of the Mambo code, corresponding to its own platform, have not been run. The client then checks out the code from a CVS repository after authentication and proceeds to run the regression test. The execution of the regression test sometime requires specialized resources from the node on which it is executing, such as, a virtual network port. Upon completion, the client writes the result in the database, success or failure, along with some informational items, such as, performance results.

A test-case MHM is running can fail due to a problem in the environment or an actual problem with the architecture being simulated. It was considered important to distinguish between the two cases. Examples of environment-related problems causing a test to fail are many: a flaky NFS connection that fails intermittently, a cron job fails to get authenticated with the LDAP server, Linux failing to map the simulator's network port to the machine's network port, /tmp filling up. A problem like this can cause a developer to falsely believe that her architecture code is buggy when in reality the problem lies in the environment. A key source of difficulty is that these problems are often transient and the different software elements do not have error messages that correspond to the actual problem. We choose the problem of losing NFS mount as it has been a frequent problem for users of MHM over its seven year lifespan.

Fault injection. We emulate NFS problems by dropping outgoing NFS packets with a probability of 0.1. The NFS packet dropping functionality is implemented by adding an iptables rule at the start of the faulty run.

Code annotations. We run ORION in an asynchronous mode on MHM for it to be less intrusive to the application. The asynchronous profiling process collects metrics at a 1 sec granularity. MHM was instrumented at 48 instrumentation points in 1400 lines of Tcl source code, which resulted in 2227 records. The Tcl script invokes library-like Perl and bash scripts. Since these had been rigorously tested, we were told that they should be kept out of scope of our problem localization effort. The instrumentation code records a timestamp and an identifier for the code region. Unlike in the other applications, this code did not have finely granular methods (and of course no classes). Therefore, the points to insert instrumentation was a subjective decision and this was done based on the liberal amount of comments in the code. Our instrumentation covers the starts and the ends of crucial operations, such as, CVS operations, NFS operations, and database operations. It is in fact finer in its granularity and also covers other structures, such as `if-then-else` blocks.



```
Top Abnormal Metrics
--------------------
[1] wchar
[2] read_bytes
[3] rss

Top Abnormal Code Regions (for wchar)
----------------------------------
[1] Code to avoid problems with X tests
[1] Code to avoid running tests on debug builds
[1] Checking for more commits we could work on
[1] Disconnecting from the database
```

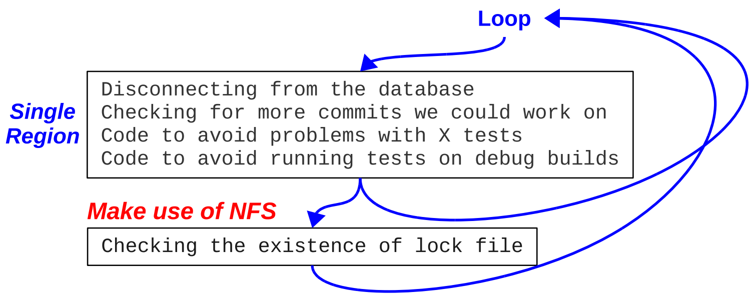

Loop
Single Region
Disconnecting from the database
Checking for more commits we could work on
Code to avoid problems with X tests
Code to avoid running tests on debug builds
Make use of NFS
Checking the existence of lock file

Fig. 6.11.: Results from ORION for the Mambo Health Monitor problem.

Results. We collect traces of a normal and of a failed run. In both runs, we use the same machine as the MHM client and keep the workload pattern the same. Figure 6.11 shows the results of applying ORION to the traces. First, notice that the two top abnormal metrics are metrics related to I/O, i.e., `wchar` and `read_bytes` (written characters to disk and read bytes from disk, respectively). Next, we find abnormal code regions using `wchar` as our abnormal metric—ORION ranks equally four different code regions as the figure shows. Notice that none of the pinpointed regions perform any operation that makes use of NFS, the root cause. However, when we look at the code (Figure 6.11), we notice that these regions are short and are always executed together inside a loop, so they can be grouped into one region. We also notice that, right after this grouped region, there is a code region that makes use of the NFS, i.e., the `Checking-the-existence-of-lock-file` region. This region performs I/O to access a file that is mounted using NFS and is affected by the injected fault.

The reason the NFS-related region is not ranked as an abnormal region (in fact is ranked top 4 by ORION) is that measurement inaccuracies emerge from asynchronous profiling. These regions of code (demarcated by our instrumentation points) are small compared to the frequency of metric collection. Hence, it becomes difficult to accurately map the metric collection points to within an instrumented code region. Hence, a design decision in ORION from this insight is that when asynchronous profiling is used and the instrumented code region is "small", ORION pinpoints to the user not only the abnormal code region but also one region before and one region after the abnormal one. This strategy works well in this case since the code region that is actually affected by the fault is right after the region that ORION selects as abnormal.

6.4.5 Overhead

Table 6.4 shows the overhead of the profiling, training and the problem localization steps in seconds. Profiling represents the time it takes to collect a trace record, i.e., a sample of all the metrics. The training step involves creating the hyper-sphere of normal behavior. The problem-localization step involves finding the abnormal metric and, subsequently, the abnormal code region. Localization takes more time than testing because our algorithm creates a large number of windows (of multiple sizes) and it iterates over all the windows to find abnormal behavior.

Table 6.4: Summary of overhead (in seconds) per application.

Application	Profiling	Training	Localization
Hadoop	0.00103	31.06	213.06
HBase	0.00103	150.1	769.38
StationsStat	0.00585	217.43	1645.1
JHM	0.00585	12.48	16.66

6.5 Practical Implications

A direction we have not explored is the sensitivity of our methods to a larger number of metrics because we are using all the metrics that are readily available through standard instrumentation packages. With a larger number of metrics, first, the time for profiling and analysis would increase. Second, the accuracy of the results could suffer due to the abnormality being lost in a sea of normality, what is referred to in the machine learning literature as the curse of dimensionality [60].

We observe that, as Orion drills down deeper looking for problematic code regions (class - nested subclasses - method), it provides less accurate results. An example of this case is the Hadoop bug in which we are able to identify the abnormal code region at a Java class granularity but not at a method granularity.

Applications that do not provide delimiters for the important code regions would require manual effort from the developer (like with the JHM system) to indicate what are good instrumentation points. For most applications however, we expect Orion will automatically annotate the entry and exit point of methods.

A trade-off of our asynchronous profiling approach is the difficulty of mapping metric samples to code regions accurately, when the code region is short relative to the time it takes to sample all the metrics. However, this comes with the advantage of minimal perturbation of the application. So, for asynchronous profiling, Orion provides code regions that are adjacent to whatever code region it finds as abnormal.

For synchronous profiling, we only instrument a subset of all the code regions that are executed at runtime to limit the runtime overhead, e.g., by only instrumenting Java classes of a particular package. To decide on a particular package, we can use the developer intuition of the origin of the problem. For example, if the bug arises only when a particular package is updated, she could only instrument that package to look for the bug.

6.6 Discussion

We propose ORION, a system to perform root cause analysis for failures in distributed applications. Out of a comprehensive set of monitored metrics, our technique pinpoints the metric and a window that is most highly affected by a failure and subsequently highlights the code region that is associated with the origin of the problem. Our algorithm models the application behavior through pairwise correlations of multiple metrics, and when failure occurs, it finds the correlations (and hence the metrics) that deviate from normality. Our case studies with four large-scale distributed applications show the utility of the tool — ORION can localize the origin of real-world failures at a granularity of metrics and code regions in the matter of minutes.

7. FAILURE PREDICTION IN COMMERCIAL APPLICATIONS

In this chapter we present Augury, a framework for predicting failures and detecting errors in commercial distributed applications. Augury tracks spatial and temporal relations in a comprehensive set of metrics from the stack-to-application stack (similar to Orion as explained in Chapter 6), and thus does not force the system administrator to second guess which metrics to consider in the analysis. The detection algorithms are computationally inexpensive, accurate, and typically predicts failures 15 minutes or more before the event. Its efficacy is demonstrated through synthetic injections, from real-world bug cases in the Android OS and a bug from a Purdue campus-wide Java application for checking availability of lab machines (StationsStat).

7.1 Introduction

Motivation. Today's distributed applications are composed of a large number of hardware and software components. Many of these applications require continuous availability despite being built out of unreliable components. Therefore, system administrators need efficient techniques for error-detection that can operate online—as the application runs—and that can detect errors and anomalies with small delay—the time between the error manifestation and its detection should be short. Preventing an error from becoming a user-visible failure is a desirable characteristic. Automatically predicting impending failures based on observed patterns of measurements can trigger mitigation techniques, such as microrebooting [5], redirection of further requests to a healthy server, or simply starting a backup service for the data.

Today's enterprise-class distributed systems routinely collect a plethora of metrics by monitoring at various layers—system-level, middleware-level, and application-level. Many commercial and open-source tools exist for collecting these metrics, such as HP OpenView, Sysstat, and Ganglia. Examples of useful metrics are: CPU, memory, storage, and network bandwidth usage at the system level; resource usages in a Java EE container (such as Tomcat or JBoss) at the middleware level; number of requests and number of exceptions in servlets at the application level. A common class of techniques for error detection works as follows. From values of metrics collected during training runs, a model is built up for how the metrics should behave during normal operation. The techniques differ in what exactly they model and the sophistication of the model—models can be built for instantaneous values of the metrics (that need to be below or above some threshold), trends in the values of metrics individually, or correlations in the values of pairs of metrics. At runtime, a comparison is made between what is indicated by the trained model with another model that is built up with metric values collected at runtime. If there is enough divergence between what the two models indicates, an error (or anomaly) is flagged.

Current approaches toward building error-detection systems based on statistical analysis of runtime metrics suffer from one or more of the following problems:

- *Their models do not consider multiple metrics simultaneously* [7, 91]. Many software bugs and performance faults are manifested in such a way that the correlations between measurements of different metrics are broken and these bugs are then missed. Some approaches do use multiple metrics but do not use correlations between values of the metrics [43] and therefore are not able to catch such problems (Figure 7.1 shows an example).

- *Some models do not consider observations of a metric as a sequence of measurements* [6, 92]. Many software bugs, for example those related to performance problems, develop a distinctive temporal pattern that can only be captured by analyzing measurements in a sequential manner rather than through instantaneous snapshots of the metric values.

- *The overwhelming majority of techniques do not perform any prediction.* Many of the current error-detection systems run in a *reactive* mode by flagging alarms when a failure occurs rather than in a *proactive* mode by anticipating a failure. Failure prediction has been a hot topic in the past years, however, to the best of our knowledge all the failure-prediction systems [43] suffer from either the first or the second problem (or both).

- *Existing approaches often consider a restricted set of metrics for modeling* [7]. A reduction in the number of metrics is done so that the online performance of the technique can keep pace with the distributed system. These approaches do not work in general because a fault can be manifested in a metric that is filtered out initially, making the error-detection system incapable of catching such type of faults. Therefore the challenge is to consider a broad class of metrics at all three levels and yet keep the algorithms efficient enough for online operation in production environment.

Summary of Contributions. In this Chapter we describe AUGURY, a system to perform error detection and failure prediction that overcomes the problems that are present in the existing approaches. AUGURY addresses the current problems by combining the following techniques:

1. **Sequential Multi-Metric Analysis:** We address the first challenge by considering a large set of metrics from the system, middleware, and application levels. AUGURY uses time series models for capturing the evolution of metric values with time. We use pairwise correlations between the metrics to detect subtle errors. While simply comparing the values of a single metric with a threshold is practical and useful in catching many errors (provided appropriate thresholds can be determined), this approach misses many subtle errors and also flags false alarms for subtle legitimate interactions between metrics.

   To understand this, consider a motivating example observed when a file-descriptor leak is manifested in the version of Android OS, nicknamed Gingerbread (shown

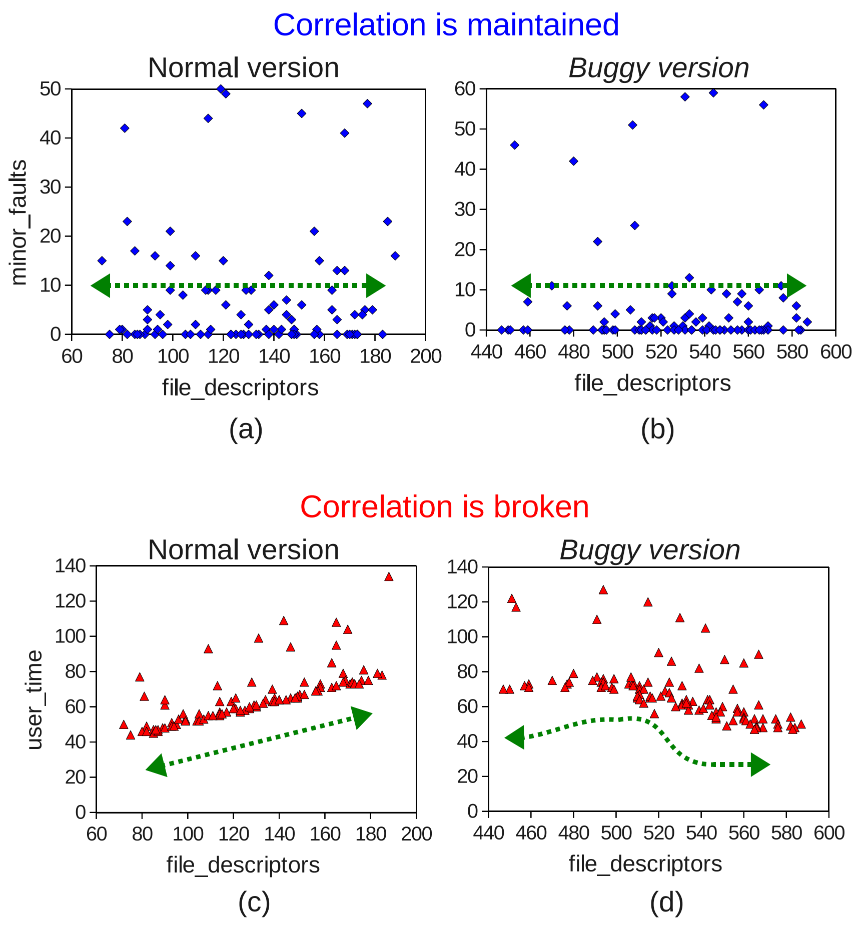


Fig. 7.1.: Example of a correlation that is broken and a correlation that is maintained.

in Figure 7.1) [93]. Here we run a workload program that performs CPU-intensive tasks after opening files using a Java class with a bug in it (the *buggy* version), and another version of the same program that opens files using the non-buggy Java class (the *normal* version, fixed in Gingerbread by August 2010). The left-hand figures ((a) and (b)) show the normal and buggy versions of a correlation that is *not* broken by the bug. The absolute values of the `file_-descriptor` metric surpasses the maximum values seen in the normal version (up to 590 file-descriptors compared to 190 in the normal version), but the correlation between the two metrics is not destroyed. The right-hand figures ((c) and (d)) show a correlation that is broken because of the bug—as the number of file-descriptors that are left open increases but the user-level CPU usage does not, in the buggy version. ORION and AUGURY share the same goal of using a large number of metrics for the analysis.

2. **Failure Prediction:** AUGURY has a predictive operational mode that uses time series models (that have been created offline through training data of typical workloads) and recent observations of the metric's values to forecast what the values will be in a future time window. It then computes the pairwise correlations of the metrics and compares them to the learned legitimate correlation values. The pairwise correlations for all the metrics are represented as a hypersphere (with a high dimension of n^2 where n is the number of metrics) and the comparison is done through the nearest neighbor distance. AUGURY can leverage if certain flagged failures are included in the training data, but does not depend on such data. Thus, through this mode, AUGURY is able to predict impending failures, as long as there is a progressive escalation of some latent errors into the failure. Such prediction is useful in practice because proactive recovery can be initiated. A larger look-ahead before the failure is preferable because of the lag involved in a successful recovery process. For example, with software rejuvenation, a mechanism for proactive recovery, it is generally agreed that the time to rejuvenate stateful distributed applications is of the order of 15 minutes (and higher depending on the amount of state) [94].

We develop AUGURY in C++ and apply it to detect and predict failures in: **(1) StationsStat:** a multi-tier application that is used to check the availability of workstations on Purdue's computing labs. The application suffered from an unknown bug that made it fail periodically by becoming unresponsive to end users. AUGURY predicted the majority of the failure cases with 51 minutes of lookahead on average; **(2) RUBiS:** a multi-tier application that mimics an auction site by injecting a suite of four classes of errors—programming, performance, configuration and network—and for each class, quantify the capability of predicting the failure, and failing that, of detecting the error.

We compare the result from Augury individually to that obtained from a regression-based technique for prediction of failures. The online performance of Augury appears to be fast enough to be used in deployed systems—it takes less than 10 ms in performing predictions and it can predict failures 2-33 minutes in advance. In terms of asymptotic running time, if correlations of n metrics have to be considered, then the running time of any algorithm will be $O(n^2)$, which is the running time of Augury as well. We reduce the dimensionality n by using a fast shortcut approach for some metrics whose values are extremely stable through all the workloads. This reduces the dimensionality for our testbed application from 143 to 50 metrics.

From this study, we gain the following two primary insights. First, it is possible to predict failures even without forecasting values of metrics. The failure look-ahead time is significant enough for mitigation actions (typically 15–51 minutes in our experiments). This insight applies particularly to progressive resource exhaustion failures, which can be due to leaks, runaway processes, or load imbalance among multiple cores. Second, for many faults, only a few metric correlations break. The total number of metrics and hence the number of pair-wise correlations is large (we have 10,000 pair-wise correlations in our testbed). Therefore, techniques are needed that can uncover such crucial needles in the correlation haystack.

7.2 Overview

System administrators use Augury to detect software bugs, performance anomalies or unexpected runtime condition that affect end users. Like Orion (from Chapter 6), Augury collects measurements of multiple metrics at different levels in the system, e.g. operating-system-, middleware- and application-level. Metrics can be collected in production environment—while the system experiments regular user-generated workloads—or in profiling environment—when synthetic workloads are applied to the system in an offline manner. The overhead of collecting the measurements is not significant. Using collected measurements, Augury builds models of normal-behavior, which allows prediction of imminent failures, or detection of failures when prediction is not feasible. The failure prediction steps are performed in an online manner so that system administrators can take actions proactively to avoid failures.

Figure 7.2 shows an overview of the approach. For a given set of metrics, we collect observations for each periodically in a fixed-size time window—we call this an *observation window*. For each window, we find a mathematical model λ that explains well the observed temporal pattern and that can perform *forecasts* k-steps ahead for each metric. Since it would be computationally expensive to infer sequential-data models in an online manner, we create a database of normal-behavior models *a priori* and find models that best fit the observations in an online manner (Section 7.4.3 explains more how models are selected from the database). Given a sequence of observations, Augury performs a forecast of the metric's values for an adjoining future time window.

Failure prediction is performed by calculating a *correlation coefficient* for each pair of metrics and by looking at deviations of these coefficients from normal-behavior coefficients (which are collected offline). The rationale behind this approach is that bugs and performance faults will change the correlations between the affected metric(s) and the rest of the metrics, while maintaining the legitimate correlations in the other metrics. To illustrate this idea, consider a bug where unused database connections are kept open. In this case, metrics such as `file_descriptors` and `open_sockets` will be affected by the bug and will exhibit a different temporal pattern than during workloads where the bug is not activated. However, correlations among the other metrics will not be affected.

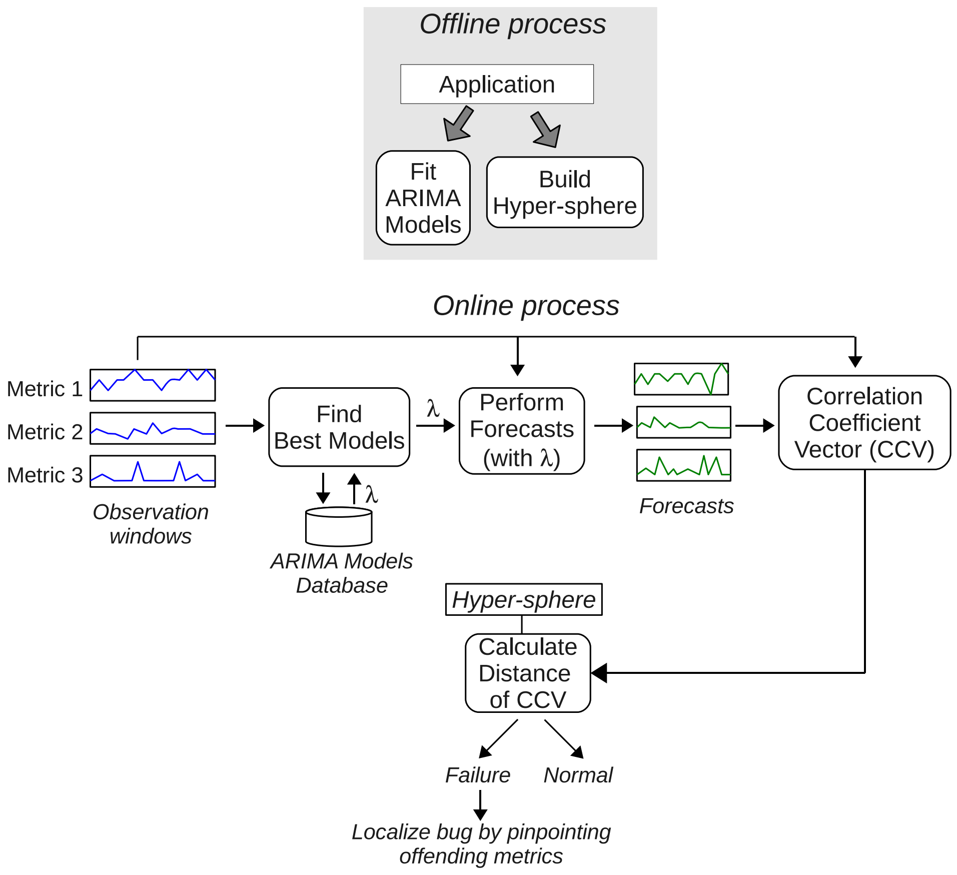


Fig. 7.2.: Overview of AUGURY's approach.

7.3 AUGURY: Building Blocks

In this section, we describe the components that form the building blocks in the design of AUGURY and provide relevant background material for each.

7.3.1 Sequential-Data Models

Many bugs and performance anomalies develop a characteristic temporal pattern that can only be captured by analyzing measurements of metrics in a sequential manner rather than by observing instantaneous snapshots of the metric values. AUGURY uses time series analysis models to capture the temporal patterns of metric observations. A time series model is a stochastic model that summarizes meaningful statistics of a time series—a sequence of data points, taken at successive times spaced at uniform time intervals. The model provides the ability to forecast future observations based on known past observations in a computationally efficient way. In particular, we use autoregressive integrated moving average (ARIMA) models since they are powerful and yet computational efficient for forecasting values of metrics (as long as the forecasted series is short, say between 15 and 20 steps ahead). A model is referred to as an ARIMA(p, d, q) where p, d, and q are (non-negative) integers that refer to the order of the *autoregressive*, *integrated*, and *moving average* parts of the model respectively [95]. The autoregressive part can be seen as a linear regression of the current value of the series against one or more prior values, whereas the moving average part is conceptually a linear regression of the current value against white noise (or error from imperfections in estimating previous values) of one or more prior values of the series. The integrated part is used to differentiate the series in case it is non-stationary and so it allows modeling complex temporal patterns.

To explain better how ARIMA models are used in AUGURY, consider an ARIMA(2,0,2) model as an example. With this model we can forecast a new observation X_t, given previous observations X_{t-1}, X_{t-2}, using the following equation:

$$X_t = C + \varepsilon_t + \sum_{i=1}^{p=2} \varphi_i X_{t-i} + \sum_{i=1}^{q=2} \theta_i \varepsilon_{t-i}, \tag{7.1}$$

where $\varphi_1, \ldots, \varphi_p$ and $\theta_1, \ldots, \theta_q$ are parameters of the model, C is a constant, and the $\varepsilon_t, \varepsilon_{t-1}, \ldots$ terms are white noise terms which are assumed to be independent identically-distributed random variables sampled from a normal distribution with zero mean. In Section 7.4.2 we explain these parameters are inferred using training data. A nice property of ARIMA models—that makes them computationally efficient—is that small values of parameters p, d, and q are usually good enough to model most of the time series encountered in practice (normally $p, q \leq 2$ and $d \leq 1$, and in rare cases $p, q = 3$ and $d = 2$ is needed) [95].

7.3.2 Correlation-Coefficient Vectors

Once we have forecasts for each metric, we attempt to determine whether a failure will occur in the near future by finding deviations from normal-behavior given the observed plus the forecasted information. The steps involved in this part are the following:

1. **Forecast concatenation:** Given a sequence of forecasts, AUGURY concatenates the forecasts to the end of the observation window. At this point the window contains past, present and future information. This step is done individually for all the metrics.

2. **Correlation coefficients:** AUGURY builds a vector of all (pair-wise) correlation coefficients $(cc(1,2), cc(1,3), \ldots, cc(n-1,n))$, where $cc(i,j)$ is the correlation coefficient of metrics i and j, $i \neq j$, and n is the number of metrics. We denote this vector as a *correlation coefficient vector* or CCV. As in ORION, correlation coefficients are calculated using the *Pearson correlation-coefficient* formula:
$$cc(X,Y) = \frac{1}{N-1} \sum_{k=1}^{N} \left(\frac{X_k - \bar{X}}{s_X} \right) \left(\frac{Y_k - \bar{Y}}{s_Y} \right), \tag{7.2}$$
where N is the number of elements of observations in the window, $\bar{X}$ and $\bar{Y}$ are the mean of variables X and Y respectively, and s_X and s_Y are the standard deviations of X and Y.

3. **Anomalous behavior classification:** To determine whether a CCV is normal or abnormal, we determine how different it is from normal-behavior CCVs that are created previously from the same application through the training phase. We assume that users of AUGURY have access to normal-behavior application traces which allow the creation of the CCVs. The traces can be obtained by collecting metrics while the application runs in production environment, or by imposing a set of workloads to the application that are representative of the workloads that are expected in production environment. CCV is considered as

abnormal if the distance to its nearest neighbor in the set of the normal-behavior *CCV*s exceeds a threshold.

7.3.3 Classification of Anomalous Behavior

Normal-behavior *CCV*s will form a *hyper-sphere* of $m = \binom{n}{2}$ dimension. A *CCV* is flagged as abnormal if it is far away from the hyper-sphere. We assume that, in reality, it would be difficult (if not impossible) to obtain an exhaustive set of error-free *CCV*s. Therefore, instead of flagging a *CCV* as abnormal if it is outside the hyper-sphere, we only flag it if the distance of the *CCV* from the hyper-sphere is far more than a predefined threshold. The use of a threshold allows the possibility of having imperfections in the metric measurements and in the workloads that are seen when normal-behavior traces are collected—in practice, it is possible that only a subset of all possible workloads that are observable in production are used during the training.

Finding *abnormal* data points (i.e., *CCV*s) based on a set of only *normal* data points (i.e., a hyper-sphere) can be seen as an outlier or novelty detection problem. Nearest neighbor (NN) is a powerful and widely-used technique for outlier detection in machine learning. Given a set of training data points, it simply computes the distances between a test data point x and all the training examples and uses the highest similarity score (lowest distance score) and compares it to a threshold to determine whether x is normal or abnormal. AUGURY makes use of NN to classify a *CCV* as normal or abnormal given training examples of *CCV*s. Figure 7.3 shows a sample hyper-sphere for 3 metrics. Points in the space are three-dimensional *CCV*s where dimensions are correlation-coefficients cc(1,2), cc(1,3), and cc(2,3). Nearest-neighbor (NN) is used to determine whether a *CCV* is normal or abnormal. Note that even when a *CCV* is outside the hyper-sphere (e.g. p_1), it is considered normal if its NN distance is less than a threshold.

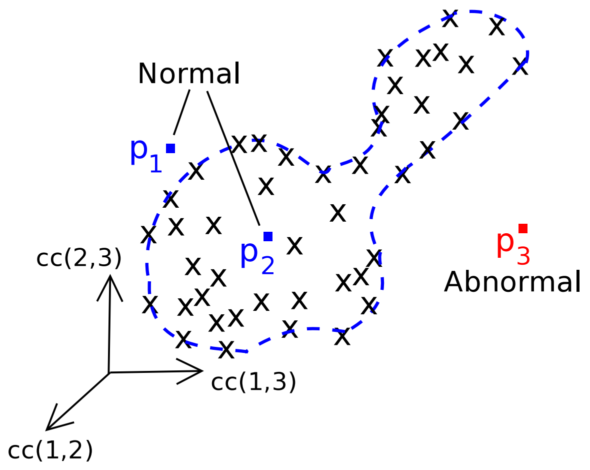


Fig. 7.3.: Hyper-sphere example for a 3-metrics system.

The complexity of classifying a new CCV using NN is $O(N \cdot m)$, where N is the number of training points (or normal-behavior CCVs), and m is the dimensionality of a CCV, which is equal to n^2 for n metrics. For a large application, m will dominate the complexity term because a large number of different metrics have to be used to detect a wide variety of errors. On the other hand N will be constant in the limit if the set of application workloads is constant, giving a complexity of $O(m)$. We believe that the number of distinct workloads for an application either stays constant or scales slowly as the application adds more functionality or scales to run on a larger number of nodes. This is because while specific request patterns may differ with added functionality, the high level of abstraction at which a workload class is considered will mean that the number of such classes stays relatively constant. Consider as an example that for the database benchmarks, as the database gets more tables (corresponding to new functionality), the workload classes defined as browse and browse-plus-buy remain unchanged. If the number of points in the hyper-sphere is large, one can use clustering techniques to group similar CCVs and to build the hyper-sphere based on the cluster centroids. In our experiments we did not use clustering because the NN distance was sufficiently fast to compute (in the order of milliseconds). In cases where additional speed is required when using NN, its algorithm can be parallelized using data decomposition—the NN distance calculation can be performed by decomposing it into independent sums (of independent correlation coefficients) followed by a reduction (sum) operation. Since in our experiments with 143 metrics we do not find a bottleneck when using NN, and because of its simplicity, we chose it as the outlier detection method for AUGURY.

In practice the training runs may have some erroneous executions as well (though they did not in our experiments). In such cases, the above scheme for classifying anomalous behavior will suffer from some missed alerts, proportional to the fraction of erroneous CCV points. In such cases, we can use a k-nearest neighbor metric to classify anomalous behavior, which will be less susceptible to erroneous points in the training data.

7.3.4 Metrics

Software systems are composed of multiple layers, each of which has metrics that can be measured using existing monitoring tools. Application-level metrics are the most specific to the services the application provides, and can be very useful in detecting and diagnosing errors that affect end users directly. For example, by observing processing time of web components (e.g., Java servlets or EJBs), monitoring tools can pinpoint buggy components that cause high response time to end users. Middleware-level metrics are less fine grained than application-level metrics, and, as a result, can be used to detect anomalies that affect multiple applications sharing the same middleware. Examples of metrics obtained at middleware-level are garbage collection statistics (e.g., in the Java Virtual Machine), thread-related metrics, and web container metrics. Operating system metrics are in the bottom layer, therefore they can be used to isolate faults that are related to a particular machine—in distributed systems, for example, administrators can measure operating system metrics to detect nodes that experience memory leaks, process deadlocks and packet losses.

It is important to monitor metrics from all layers in the system because faults can be manifested in metrics from any layer. As a consequence, it is difficult to exclude metrics from the error detection system since a fault that is manifested in one of the excluded metrics will not be detected. Since the total number of metrics to analyze is large, error detection algorithms need to scale well with the number of metrics in the system. As the functionality of applications increases, the number of metrics to analyze also increases because metrics from more application components and nodes need to be considered. Augury has a quadratic computational complexity in terms of the number of metrics, which is simply the complexity of calculating correlations between all the metrics.

7.4 Design of Augury

In this section, we explain the design of Augury: the practical approach to collect system metrics, methods to train statistical models, and the steps in the failure prediction process.

7.4.1 Data Gathering

We gather 143 metrics from different layers in the system—OS, middleware and end-user application—to have a full knowledge of the application's behavior. We do not rely on the system administrator knowing *a priori* which metrics will be affected by the bugs or performance faults. Therefore our approach considers a wide range of metrics and can automatically zoom into the pairwise correlations of metrics that are relevant to the error. The Table 6.1 in Chapter 6 shows a description of the observed metrics.

Figure 7.4 shows the number of metrics that we collect in each layer. We monitor 143 metrics in total: 30 OS metrics (15 per process), 15 middleware metrics (Apache Tomcat), 6 MySQL metrics, and 92 servlet metrics (4×23 servlets). To collect metrics of the Linux OS, we use the `sysstat` utilities which are a collection of open-

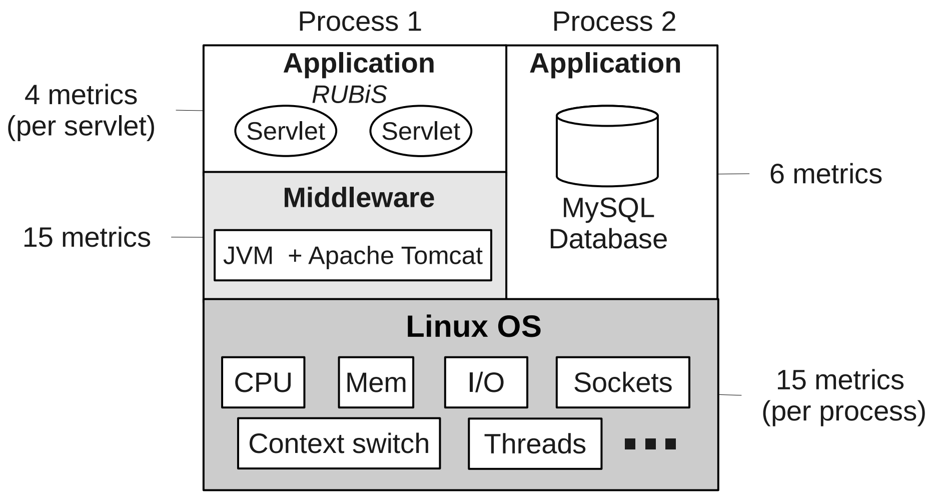


Fig. 7.4.: Monitored metrics per layer.

source performance monitoring tools for Linux [96]. In particular we used the `pidstat` tool that allows us to collect OS-related metrics on a per-process basis. The tools need very little CPU to run (they are written in C) so they do not perturb the monitored application. We collect 15 per-process OS-related metrics such as system- and user-CPU usage, virtual- and resident-memory usage, (minor and major) page faults, disk reads and writes, open sockets, file descriptors, and voluntary- and non-voluntary-context-switches.

To collect measurements in the Java middleware and web application, we use a `JMX` connector in the Apache Tomcat server version 5.5. The connector allows to evaluate metrics (at the middleware level) from the Tomcat server such as cache hits, `ThreadPool` usage, requests count and processing time. We also evaluate Java servlet metrics (at the application level) such as processing time and request counts. In the MySQL application layer, we use the `SHOW STATUS` command in MySQL to collect 6 metrics that represent the database status, such as memory usage, read/write hits, open tables, connections per second, and number of threads.

Augury filters out metrics that remain constant in the training traces. This helps reducing the dimensionality of the *CCV*s, allowing Augury detect anomalies faster.

AUGURY also filters metrics that are *almost* constant, i.e., metrics with a very small variance. It checks the values of removed metrics at runtime—an inexpensive task—and if one of these metrics vary, an alarm is flagged. We found that, after filtering out metrics, AUGURY only considers 46 metrics in the detection part. This allows AUGURY focus its power in the metrics that are more relevant for error detection.

The overhead of collecting the measurements of the 143 metrics is 36.91%, which is calculated by the increase of the server reply time from the baseline, i.e. when measurements are not collected. When measuring the overhead, server operates at 90% of its capacity. The choice of measurement tools is orthogonal to AUGURY, and AUGURY can easily accommodate additional metrics as long as the measurement tool can communicate with AUGURY through IPC calls.

7.4.2 Training Time Series Models

The training phase in AUGURY involves finding appropriate ARIMA models for each metric based on traces obtained when imposing normal-behavior workloads. Section 7.5.2 explains how workloads are generated. In this section we give details of time series models that are generated to resemble temporal patterns observed during training workloads.

We divide traces of each workload into multiple *segments* of equal size and fit an ARIMA model for each segment. The size of segments determines the overall time spent in training ARIMA models because, for each segment, we have to try several ARIMA models as to select the one that best fits the behavior in the segment. Therefore, in order to make this task manageable, we used segment sizes that are large enough that training can be done in a reasonable amount of time and but that are small enough that are to produce accurate models. In practice we found that segments of 5,000 observations produce very accurate ARIMA models and allow us to infer all the models in about 24 hours. For each workload, we have a total of around 27,000 observations, hence 6 ARIMA models are produced. Since we have 3 workloads and 143 metrics, a total number of $6 \times 3 \times 143 = 2,574$ models are created.

Training ARIMA models involves finding the parameters of the model that best fit the data. The algorithm for training ARIMA models is as follows: (i) for each metric and workload segment, we build an ARIMA(p,d,q) model, where p, d and q are varied over $\{1, 2, 3\}$ (it has been found that a maximum order of 3 is good enough to capture the majority of the temporal patterns that are encountered in practice [95]); (ii) we calculate the *Akaike information criterion* (AIC) for each ARIMA(p,d,q) model; (iii) the preferred model is the one with the minimum AIC value. This model is saved in a database to be used when forecasting measurements. The AIC method finds the model, among a candidate set of models, that best explains the data with the fewest parameters p, d, and q. The *Bayesian information criterion* (BIC) method can also be used and it should provide similar results to the AIC method. We use the `GNU R` program [97] to train ARIMA models.

7.4.3 Online Failure Prediction

Next we describe the steps involved in the failure prediction procedure in Augury: (1) maintaining a multi-metric sliding window of observations, (2) forecasting measurements for each metric, and (3) detecting abnormal deviations using the observed and forecasted information.

Sliding Observation-Window

Augury keeps a per-metric *observation window* that is used to forecast future measurements of that metric using a selected ARIMA model. The size of the observation window depends on the requirements of the time series models in terms of the amount of information needed to perform accurate forecasts. For ARIMA models typically it is required to have moderately long series in order to have reliable forecasts; some authors recommend a minimum of 100 observations [98]. In Augury the observation window size is set to $N = 100$ with observations taken every 5 seconds. An implication of this requirement is that we are not able to catch problems before the first 100 samples—or before the first 500 seconds after the application starts–since Augury needs at least this amount of samples to perform forecasts. However, most of the bugs and performance problems found in practice manifest themselves after the service has been running for more than this amount of time.

The observation window is sliding so, for every sample, the window is moved one step forward. We found that this provides enough time for us to collect measurements from the metrics at all layers of the system without interfering with the application's operation—collecting all the metric values takes about 1.5 seconds on average.

Forecasting of Metric Values

To perform forecasts for each of the monitored metrics, we need to have two components: a window of past measurements (i.e., the observation window) and a model that allows us estimate future values (an ARIMA model). To avoid the overhead of computing forecasting models at runtime, Augury uses pre-computed ARIMA models to perform forecasts. The models are selected based on data in the observation window, and out of a database of models that is created offline as described in Section 7.4.2. The model selection is fast as it only tests a few possible models and each test takes just a few operations.

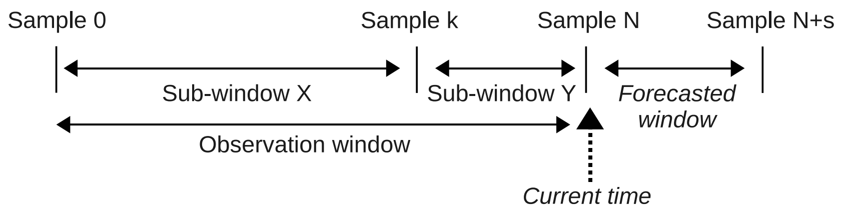


Fig. 7.5.: Windows that are used in the selection of the best model.

The idea of the model selection algorithm is that the best model to do forecasts should be the model that best fits the current observation window. The algorithm selects the best fit based on the model that best forecasts a sub-section of the current window. The algorithm first divides the current observation windows into two subwindows (see Figure 7.5): `subWinX` and `subWinY`. The algorithm iterates over all the possible models, and for each model, `subWinX` is used to forecasts $N - k$ steps ahead. The forecasts are stored in the temporal variable `testWin`. Then the root-mean-square (RMS) error of the difference between `testWin` and `subWinY` (the ground truth) is calculated. The best model is then the one with the minimum RMS error. A final check is done to determine whether the RMS error is too large for the best model which might indicate the possibility of an anomaly (Section 7.4.3 gives the details of this sanity check). The algorithm's complexity is $\mathrm{O}(M \cdot N)$, where M is the number of models, N is the number of samples in the observation window. The term N can be considered constant since in our case $N = 100$ because this is enough number of samples for ARIMA models to capture most of the temporal patterns that are found in practice. Therefore, over the limit, the overall complexity can be expressed as $\mathrm{O}(M)$.

Once a model has been selected, Augury performs s-step ahead forecasts. This process is done for each monitored metric. Forecasts using ARIMA models are iterative; i.e. if at time t we want to forecast the value of a variable at time $t + n$, we have to forecast the values for times $t + 1, t + 2, \ldots, t + n - 1$ first. Moreover, as we increase the number of steps ahead, forecast error variance (and the forecast value) converges to a constant value if the time series is stationary but the variance increases at each step if the time series is non-stationary. For this reason, a disadvantage of ARIMA models is that they cannot be used for long-term forecasts. We have found that in areas such as economics and meteorology, forecasts from 15 to 20 steps ahead can be performed (with metrics such as stocks and temperature) with reasonable accuracy [95]. For the metrics we are analyzing, we have found that, forecasting more than 15 steps ahead provides too much variance for the forecast error. For this reason,

$N - k$ and s are both set to 15 observations. Note that this does not mean that the look-ahead for a failure that AUGURY can achieve is at most 15×5 seconds $= 75$ seconds. The failure prediction can happen with current metric values, i.e., without needing to forecast any value.

Instantaneous Anomaly Detection

AUGURY can detect anomalies that manifest suddenly in the system by comparing the RMS error of the selected model to a specified threshold. If the RMS error is greater than the threshold, this could be explained by two reasons: either (1) there is no good model in the database to represent the current observation window, or (2) an anomaly is being manifested already so that none of the pre-estimated models could fit the data in the observation window because erroneous values are already part of it. If an exhaustive set of normal-behavior traces is used to fit ARIMA models (our assumption as in much previous work [7, 8, 92]), the only possibility is case (2). However, there exist the possibility that, for a particular application, one cannot obtain an exhaustive set of normal-behavior traces, for example, because the application it is not yet in a production environment or because it is infeasible to mimic the entire set of workloads that the application may receive. In this case, a threshold-surpassing alarm could be a false positive, for example if we have an abrupt workload change.

Failure Detection and Failure Prediction

The final step in AUGURY is to *detect* or to *predict* a failure. We say that a failure is predicted if an alarm is flagged before an error becomes visible to the users. On the other hand, we say that a failure is detected if AUGURY flags an alarm after a failure has been seen. This also has value because there is some bug localization information implicit in AUGURY's detection.

Once forecasts are generated for each metric, AUGURY uses the observation window plus the forecasted window to calculate a CCV. Next, it determines whether the CCV is normal or abnormal by calculating its nearest-neighbor distance using the pre-computed set of normal-behavior CCVs (which conform to the hyper-sphere). If the distance is larger than a pre-computed threshold, an alarm is flagged. The reason we use the observation window *plus* the forecasted window to calculate correlation coefficients in the CCV, rather than only using the forecasted window, is that the already observed values are more certain than the forecasted values, therefore the confidence of the results is increased by using the former window as well, while the latter window increases AUGURY's failure lookahead.

Correlation coefficients are calculated using the Pearson correlation-coefficient given in Equation (7.2). To evaluate this equation, the mean is calculated from the observations in the window (i.e., each X_k and Y_k) but the standard deviations (i.e., s_X and s_Y) are pre-calculated from the training runs. This allows the correlation coefficients to change when a the temporal pattern of a metric changes quickly, say due to a change in the nature of incoming requests.

Pearson correlation-coefficient, however, assumes a linear dependence between variables and may not be appropriate for non-linear correlations between variables. For such cases, the Spearman's rank correlation-coefficient can be computed which measures how well the relationship between two variables can be described using a monotonic function. Spearman's rank is considered a non-parametric method because it does not assume any relationship between variables. We have found that for the temporal patterns that our metrics expose, Pearson correlation-coefficient performs well enough to capture the majority of them. However, as observed in [8], some correlations of system-level metrics follow a non-linear pattern.

7.5 Testbed

7.5.1 Testbed Application

We use RUBiS, an auction site prototype modeled after eBay, as our application testbed. RUBiS has been widely used as a testbed for server performance scalability studies [99] and for dependability studies [5, 100]. RUBiS implements the core functionality of an auction site (selling, browsing, and bidding) through 21 application components and several servlets—it is composed of more than 25,000 lines of code. We use the servlets-version of RUBiS which runs under Apache Tomcat, using MySQL as the back-end database server.

7.5.2 Workload Generation

We implemented our own multi-threaded client emulator for RUBiS. Clients visit URLs based on a state-transition matrix (where states represent RUBiS URLs) that comes with the RUBiS package. A client starts in the `Home` state, and when it reaches the `End_of_session` state, it waits for 10 seconds and transitions back to the `Home` state. Clients think-time follow a negative exponential distribution with mean of 7 seconds, truncated at 70 seconds, as specified by TPC-W [101].

We designed workloads that covered a wide range of load levels, but that never exceeded the server's maximum capacity. First, we measured the maximum capacity of the server by varying the number of concurrent users and observing the number of replies per second. The maximum capacity was set as the point with the lowest number of concurrent users that caused the number of replies per second to plateau. Then, we scaled each workload so that the maximum load level was between 90% and 100% of the maximum capacity. We used three different types of workloads: RAMP, STEP, and BURSTY. RAMP and STEP were built as described in [9]. The BURSTY workload was created using the method described in [102] with index of dispersion of 2,025. All three workloads are 36-hours long.

7.5.3 Baseline Approach: Polynomial Regression

We use polynomial regression as a comparison point for Augury. This method fits an nth order polynomial to each metric independently using time as the predictor variable. Twenty future samples are forecast for each metric. Using the forecasted values, an alarm is flagged if these values are not within the threshold interval $(\mu - 5\sigma, \mu + 5\sigma)$, where μ (the mean) and σ (the standard deviation) are computed from the training data. The detection strategy is to flag an alert if even a single metric goes beyond the threshold interval. In the implementation, a sliding window of 100 samples is taken (as in Augury) for each metric and an nth order least squares polynomial is fitted to the sample window. The algorithm begins with a 1st order polynomial regression (linear model) and iterate through higher order polynomials until either the coefficient of determination is above a set value or the order of the polynomial is equal to the maximum order allowed. These upper bounds both limit the execution time of the program as well as limit the error of the forecasted values. During experimentation, we found that high order polynomials were less effective at forcasting future values of a metric (a 2nd order regression worked well for most metrics). The polynomial-regression baseline program is implemented in Java and has a runtime of 415 ms (model generation and forecasting) with 143 metrics at every sample, which are 5 seconds apart in our experiments.

7.6 Experiments and Results

7.6.1 Fault Injection

To evaluate Augury, we inject six types of faults that emulate common real-world problems. Table 7.1 shows a description of the injected faults and their categories. Three categories of faults that are typical in production environments were injected: configuration (CONFIG), performance (PERF) and programming (PROG) faults.

Fault Name	Description	Fault Type		
		CONFIG	PERF	PROG
`memory_leak`	Unused objects are created and added to a list		X	X
`file_desc_leak`	Unused file descriptors are created and left open		X	X
`busy_loop`	A computationally-expensive loop is executed in a servlet		X	X
`db_lock`	A table in RUBiS database is locked	X		
`deadlock`	A servlet blocks forever waiting for a message			X
`cpu_hog`	An external program consumes 80% of CPU gradually		X	

Table 7.1: Injected faults in RUBiS.

We run 30 experiments for each fault injection. Every time an experiment begins, we restart the server and the database to cleanup the environment and to avoid fault contamination from the previous experiment. The duration of each experiment is one hour and faults are injected between 18-40 minutes after the experiment begins. This avoids the burn-in time for the application, which we empirically observed to be 15 minutes. In addition, we allow AUGURY to obtain the first 100 samples that are needed for ARIMA models to be able to perform forecasts; this takes another 500 seconds.

7.6.2 Fault Injection Results

When a fault is injected, it can become an error; an error may then become a failure. We follow the traditional definition of failures which is that a failure occurs when the delivered service deviates from the specified service. In our experiments, the ground truth for failure is when the client emulator observes either: (i) a generic error report from the server, e.g., HTTP 500 internal server error, or (ii) the response time of a request exceeds a threshold, where the threshold is set to the mean plus 3 times the standard-deviation of the reply times seen in the collected normal-behavior traces.

We introduce the metrics that we use to evaluate failure detection quality. Let us define *injection time* (IT) as the point in time when a fault is injected, and *failure time* (FT) as the point in time when a failure is detected in the client emulator, with

$IT \leq FT$. Then, when AUGURY flags an alarm, we count it as one of the following cases:

- **True positive** (TP): (i) if the alarm is flagged after IT but before FT. In this case we say that AUGURY *predicts* the failure, and we also measure the *failure look-ahead time* as the difference between FT and the alarm time. (ii) if the alarm is flagged within a short period of time Δ after FT, where Δ is set as half the size of the observation window. In this case, the failure lookahead time is zero.

- **False positive** (FP): (1) if the alarm is flagged before IT; (ii) if the alarm is flagged after IT but no failure was found by the client emulator.

- **False negative** (FN): if the alarm is flagged after FT $+\Delta$.

Based on these variables, we calculate two metrics:

$$Recall = \frac{TP}{TP + FN}, Precision = \frac{TP}{TP + FP}, \tag{7.3}$$

where *recall* (also referred to as accuracy) expresses how well AUGURY is able to either predict a failure or detect a failure (a short time after it happens), while *precision* has an inverse relation to false alarms.

We evaluate AUGURY in two operational modes: (1) *Non-Forecasting*, in which forecasting values is not performed for any of the metrics and CCVs are calculated using only the values in the observation window. (2) *Forecasting*, in which forecasting values is done for each metric. The second mode has the advantage that a failure can be predicted with a larger look-ahead and hence a proactive recovery mechanism (such as rejuvenation) can be initiated earlier. However, this is also expected to be less accurate than AUGURY in the first mode because of the imperfection of metric forecasting.

Figure 7.6 shows the results of the fault injection campaign for non-forecasting and forecasting modes of AUGURY (Forecasting is not applicable for regression and

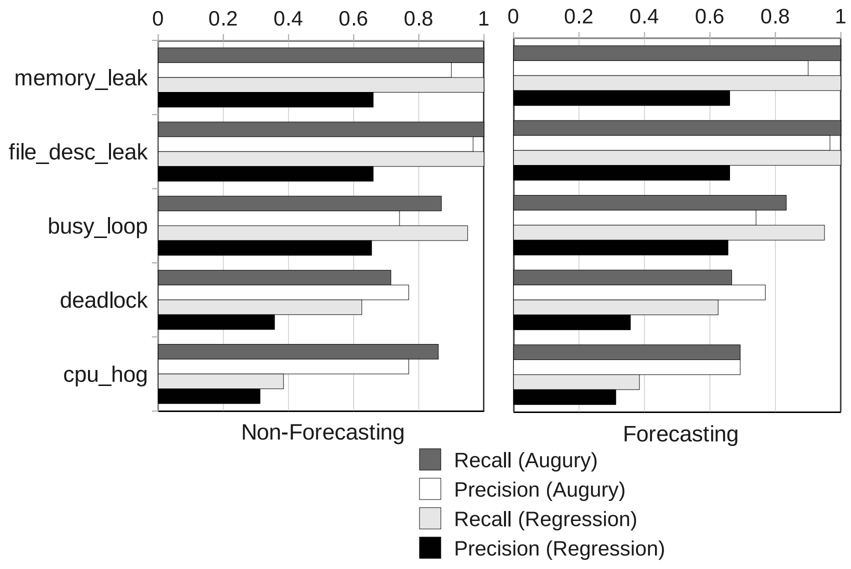


Fig. 7.6.: Recall and precision results for *non-forecasting* and *forecasting* operational modes of AUGURY, and for regression.

hence the same data is plotted for regression in the two sub-figures.). We do not include results for the db_lock fault because this injection causes the RUBiS application to lock up all the threads of the Tomcat server (waiting for the database operation to complete) and hence Tomcat becomes unresponsive. For the memory_leak, file_-dec_leak, busy_loop, and cpu_hog faults the results for recall and precision are similar between the two modes. However, in the cpu_hog fault, both recall and precision rates are less in forecasting mode than in non-forecasting mode. This is because in many cases AUGURY incorrectly flags an alarm before the injection. For regression, these results were obtained after manually removing three metrics from the analysis—MySQL sockets, read hits, and write hits. These were erratic in their behavior and were therefore throwing too many false alarms with regression. In contrast, with AUGURY the erratic nature of these individual metrics were not breaking any correlations and hence were not being flagged, pointing to some resilience in AUGURY's detection mechanism.

NF = *no-forecasting*, **F** = *forecasting*

	Failure Look-Ahead Time (minutes)		**F** anticipates failure before **NF**	
Fault	**NF**	**F**	% cases this occurs	% time reduction
memory_leak	8.69	10.19	96.3	22.72
file_desc_leak	16.32	17.61	96.55	8.47
cpu_hog	1.81	2.42	75	44.19

Table 7.2: Failure look-ahead time results for faults where Augury can predict a failure.

Figure 7.2 shows results of failure look-ahead time for the fault injections where Augury was able to predict failures. Expectedly, the forecasting mode improves Augury's capacity of anticipating failures compared to the non-forecasting mode. However, this comes at the cost of decreased accuracy for some kinds of faults. Thus, if it is a critical fault that needs to be avoided at all cost, the forecasting mode is suggested. Regression achieved failure look-ahead for two of the six fault classes: memory_leak with 31.33 minutes and file_desc_leak with 43.89 minutes. Thus, the look-ahead performance is even better than in Augury. However, it comes at the cost of much lower accuracy and no look-ahead for the cpu_hog case.

7.6.3 Performance Results

We perform experiments to evaluate the time it takes to execute all the steps that are performed online in Augury. Table 7.3 shows times and contribution percent for each step. The initialization steps correspond to those executed when Augury starts. The detection steps are those executed every time a measurement vector is taken from the application, and therefore every time the observation window is moved one step forward. Note that, in the initialization phase, the dominant factor is loading the hyper-sphere (about 9 sec) which is a file that contains 3,183 CCVs. However, this phase is done only once (when Augury starts). In the detection phase, the dominant factor is finding the distance of a CCV from the hyper-sphere (about 5.2 msec). All the steps in the detection part take less than 10 ms, which gives us enough time to perform all of them before the next sample of the measurements vector (which occur every 5 sec).

Initialization Steps

	Time (sec)	%
Load ARIMA models	9.53E-003	0.10
Load hyper-sphere	9.18755	99.90
Total	9.19708	100.00

Detection Steps

	Time (usec)	%
Get observations vector	14.28	0.16
Find best ARIMA model	3430.71	37.54
Perform forecasts	228.32	2.50
Join observations and forecasts	81.36	0.89
Calculate CCV	190.10	2.08
Find distance of CCV	5194.40	56.84
Total	9139.17	100.00

Table 7.3: Times for the initialization and detection steps in AUGURY.

We vary the number of metrics being considered and measure the time for all the detection steps. We observe that the overall time grows almost quadratically with a trend line $f(n) = (2.19 \times 10^{-5})n^{1.6}$, where $f(n)$ is the detection time in seconds, and n is the number of metrics. This is in line with our expectations since the overall complexity of the pair-wise correlation calculation is $\mathrm{O}(n^2)$ for n analyzed metrics. With this complexity, it is possible to perform analysis of more than 800 metrics in less than a second.

7.6.4 Android Case 1: File Descriptor Leak

We use bug cases in the `Android` OS to show how AUGURY detects an anomaly in a real-world scenario. `Android` is an open-source software stack for mobile devices that includes an operating system, middleware and user applications [103]. `Android` has a large community of developers and has over 200,000 applications available. The `Android` emulator lets application developers to develop and test `Android` applications without using a physical device.

Our first case is a bug that is described in issue 4825 in `Android` issues database [93]. The bug manifests itself with a `java.lang.NullPointerException` error preceded by this system message: *OSNetworkSystem: unclassified errno 24 (Too many open files).* The bug is in the implementation of the `Selector.close()` method which does not close a pipe that is created by the `Selector.open()` method, leading to a file-descriptor leak in the system. Figure 7.7 shows sample code that triggers the bug. The bug is in the implementation of `Selector.close()`

```
Selector selector;
for (int i=0; i < LIMIT; i++) {
  try {
    selector = Selector.open();
    // User performs tasks here...
    selector.close();
    // The pipe is never
    // closed because of a bug
  } catch (Exception e) {
    // Handle exception here...
  }
}
```

Fig. 7.7.: Sample code that triggers bug 4825 in Android.

To show the possibility of false alarms using a naïve detector (that looks only at instantaneous values of a single metric), we wrote a multi-threaded application that generates different workloads (Figure 7.8 shows the patterns of the workloads.): `NORMAL`: a random thread opens a file and performs a CPU-intensive task before closing it. This workload simulates the typical behavior of servers that open a socket, handle a request, and close the socket. The number of file descriptors that are used in the workload is controlled by a (range) parameter which is set to 50-100. This workload is used to train models of normal behavior. `NORMAL_INCREASED`: the same behavior as in the `NORMAL` workload, but the range of number of file descriptors is increased to 50-200. `BUGGY`: the same behavior and number of file descriptors range than in the `NORMAL_INCREASED` workload but the buggy `Selector` class is used to open files. Samples are taken every 5 seconds, and the failure (in the BUGGY workload) occurs at sample 740. In Figure 7.8, notice how AUGURY can detect the problem before the naïve approach.

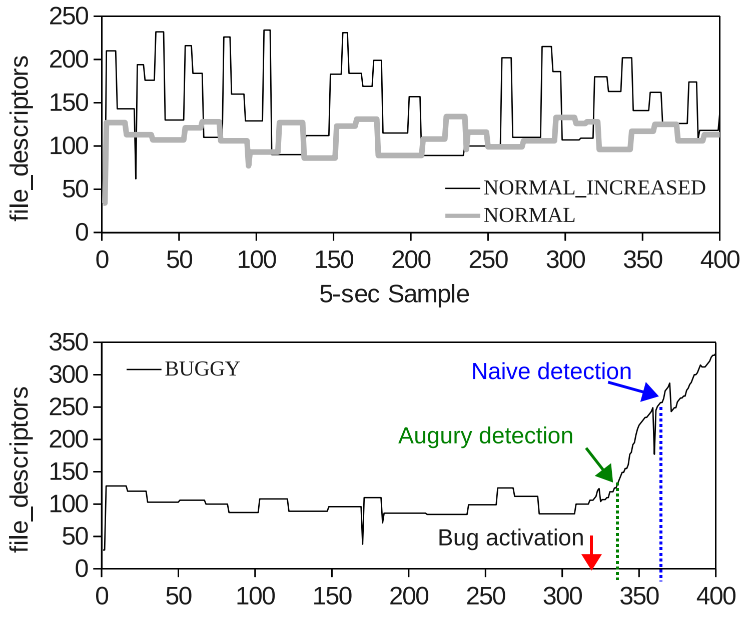


Fig. 7.8.: Number of file descriptors for the three workloads used in Android case study 1.

We wrote a program that measures 10 system-level metrics from the Android emulator every 5 seconds. Next we applied `NORMAL` workload and got metric measurements for about two hours, and trained models in AUGURY using the collected observations. We also trained a baseline naïve technique using this data set. The naïve technique simply learns a *threshold* for each of the metrics and it flags an alarm when the value of a metric goes beyond the threshold envelope of $\mu \pm 3\sigma$ of the observed values in the `NORMAL` workload.

Next, we applied AUGURY and the naïve approach to the `NORMAL_INCREASED` workload. In `BUGGY`, a failure happens when users see a `java.lang.NullPointerException` error, which happens at the 740th sample. The naïve approach correctly detects the file-descriptor leak in the `BUGGY` workload but incorrectly flags an alarm in `NORMAL_-INCREASED`. The reason is that the maximum number of file descriptors according to the threshold of the naïve approach is around 100, but, in the `NORMAL_INCREASED` workload, the number of file descriptors goes up to 200. Any threshold-based approach will have the same false-positive problem because metric thresholds are not necessarily invariant across different workloads. AUGURY, however, does not flag an alarm in the `NORMAL_INCREASED` workload because it notices that correlations between metrics are not destroyed. As a final experiment, we trained the naïve approach with the `NORMAL_INCREASED` workload, and tested both techniques with the `BUGGY` workload only. The result is that both techniques are able to detect the bug but AUGURY detects it 135 seconds before the naïve approach (see Figure 7.8).

Figure 7.9 shows *CCV* distances for this experiment. (τ represents the normal-behavior threshold for *CCV* distances. The failure point is at the 740-th sample.) Notice how the *CCV* distance starts increasing some time after the bug is activated allowing AUGURY to detect accurately the problem in `BUGGY`. Turning on forecasting mode gave us AUGURY's best performance in terms of anticipating the failure with a look-ahead time of 33 minutes. Figure 7.10 shows an example of a correlation that changes when the bug is activated, *file descriptors* with *non-voluntary context switches*, while there is one example of a correlation that is not affected by the bug, *non-voluntary context switches* and *user-level CPU time.* Notice how some file-descriptor CC's change after the bug manifests (the first plot) while other correlation coefficients maintain the same temporal behavior.

7.6.5 Android Case 2: HTTPS Request Hang

Our second case study is manifested as an application hang in Android Froyo 2.2. The hang occurs when HTTPS connections are established using the Java `HttpURLConnection` class in a very slow GPRS connection on an actual phone. This

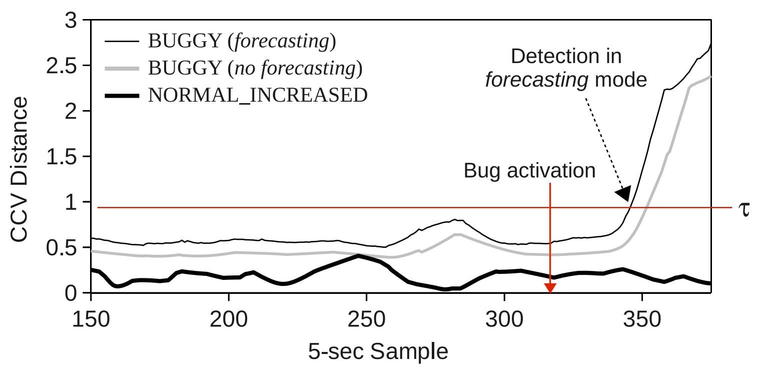


Fig. 7.9.: CCV distances for the the applied workloads.

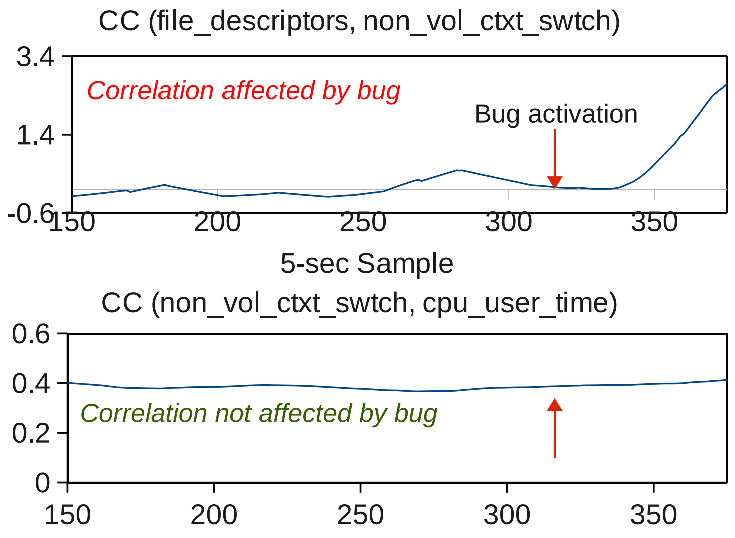


Fig. 7.10.: Examples of correlation-coefficients (CC) that are broken by the Android's bug.

bug has been reported in [104]. According to the bug report, when byte values are read from the `ChunkedInputStream` class, the `read()` method takes too long. The following code shows how the bug can be activated:

```
HttpURLConnection httpUrlConn =
  (HttpURLConnection)uri.openConnection();
// Set standard headers ...
InputStream input =
  httpUrlConn.getInputStream();
```

```
BufferedInputStream buffInput =
  new BufferedInputStream(input);

// this call hangs...
int data = buffInput.read();
```

This bug can be fixed by careful application development where a timeout is declared on the read. However, many applications do not do this careful setting.

To show how AUGURY would detect this bug in a production environment, we wrote a program that mimics two workloads that represent common behavior in web applications: `NORMAL`: HTTPS connections are opened and large files are downloaded using multiple URLs. The application spawns a separate thread for each connection. The arrival of these connections follows a Poisson process. `BUGGY`: Same pattern of HTTPS connections as in `NORMAL` but, because of the bug, connections block forever. Since more connections keep coming but none finish, the number of sockets keeps increasing till the resource limit is reached when the phone becomes unresponsive.

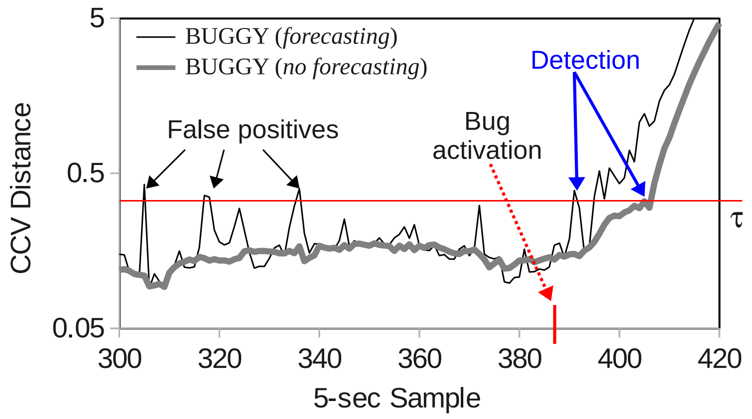


Fig. 7.11.: CCV distances for the the tested workloads in the second case study.

In `BUGGY`, a failure is flagged when reading the data from the connection takes more time than a fixed timeout. We use 300 seconds for this timeout, the default value in `Firefox`. We trained AUGURY with `NORMAL` and tested it with `BUGGY`. Figure 7.11 shows distances of CCVs in forecasting and non-forecasting mode. (τ represents the normal-behavior threshold for *CCV* distances.) The failure point is at sample 445. The failure look-ahead time is 3.16 minutes for the non-forecasting mode and 4.58 minutes for the forecasting mode. However, Augury produces some false alarms before the bug is actually activated. This is due to imperfections of the ARIMA model forecasts for metrics that experience random natural fluctuations before the bug manifests itself.

7.6.6 StationsStat Case Study

We evaluate Augury's ability to detect and predict failures in StationsStat, a multi-tier application that is used on Purdue's campus to check the availability of workstations. Students across the campus use StationsStat on a daily basis to check the number of available workstations for each lab on campus. More details of this application can be found in section 6.4.3 in Chapter 6. StationsStat's administrators tracked 495 metrics of the system-to-application stack at 1 minute intervals for more than two months from Jun 7 – Aug 19 in 2010 which we analyze in this case study.

Failure Model

The application monitoring system, Nagios, provided three types of alerts: (1) `warning`: HTTP GET requests that failed to respond within a timeout; (2) `critical`: three consecutive `warning` alerts; (3) `recovery`: the service returned to an OK state. `Recovery` occurs normally when a service-restart occurs but it could also occur because the problem simply went away itself. During instability periods (in the proximity of failures), the metric-collection script (which runs in the same application machine) could not obtain measurements from the middleware- and application-layer. These events are labeled as `missing-data` alerts.

For the rest of the study, we say that a failure occurs in StationsStat if a `warning`, `critical` or `missing-data` alert is seen. The reason we catalog `missing-data` alerts as failures is to cover problems that Nagios fails to detect and that are a clear symptom of a failure in the system. HTTP requests from Nagios and JMX requests (from the metric-collection script) are handled by different threads in Apache Tomcat—a performance problem can be reflected in only one of the threads (in the case of the `missing-data` alerts, the JMX thread).

Phase 1: Training for Metric Selection

StationsStat case is a challenging scenario for Augury, not only because the problem's root-cause is unknown, but also because there is no error-free data available to train and create the normal-behavior hyper-sphere. Fortunately, Augury can still work in this scenario by using *almost* error-free data. The administrators noticed that, after restarting StationsStat servers, the next failure was often seen only after a week or more—symptoms seemed to suggest that the problem (possibly a resource exhaustion bug) grew progressively from a service restart to a failure. Augury therefore used a data segment collected right after a restart to build the hyper-sphere representing normality. Augury also filtered out constant metrics in this phase which resulted in 70 non-constant metrics that it used in the rest of the analysis.

Even with 70 non-constant metrics—a significant reduction from 495—Augury did not provide definitive results when it was applied to the set of failures in the collected data. The bug in StationsStat seemed to only break a few correlations, possibly because only a small set of metrics were affected by the bug. As a consequence, our nearest-neighbor approach suffered from the *curse of dimensionality* [60] Due to a high dimensional space, deviations from normality in a few dimensions are not as significant, making distances between all pairs of points in high-dimensional data tend to become almost equal. We needed to eliminate *unimportant* metrics from the analysis and to use only the *important* metrics. The vexing question to us was how do we select the important metrics?

A metric set that detects (and predicts) known failures with sufficient accuracy and few false alarms can tautologically be considered the important metrics. The

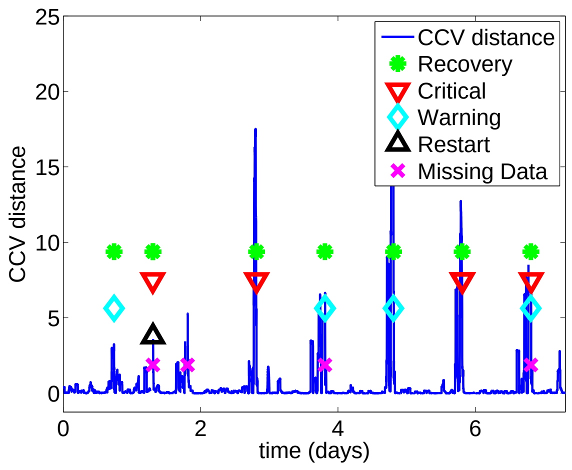


Fig. 7.12.: *CCV* distance for a segment of known failures.

reasoning we follow is that, if a metric set works well with known (already seen) failures, it should also work well with unknown (not yet seen) failures, if failures are caused by the same *unresolved* bug. The following procedure is used by the system administrators to select a small set of important metrics:

1. *Sampling*: the administrator randomly selects several sets of a small number of metrics and applies Augury individually on one set of metrics from the above combinations to a data segment that contains known failures. For this step we use 10 metrics since it was a reasonably smaller number compared to 70. We generated randomly 98 combinations and made sure that all the 70 original metrics appeared in at least one combination.

2. *Labeling*: For each combination, the system administrator determines visually what combinations worked well with the known failures and labels them as "good". Here, she looks at combinations that give a high detection accuracy—failures are proceeded or followed shortly by an increase of the CCV-distance and a small number of false alarms (i.e., the CCV-distance did not surpass a threshold when failures were not present). Figure 7.12 shows an example of a combination of 10 metrics that works well in this step and that is labeled as

"good". (In the figure, `Warning`, `Critical` and `Missing-data` labels correspond to the failures.) Notice how high CCV distance peaks are observed along with all the failures.

3. *Metric selection*: After the administrator has identified the good combinations, for each metric, AUGURY calculates the ratio of number of times it appears in a good combination to the number of times it appears in all the combinations (all the 98 combinations in our evaluation). The metrics are ranked in decreasing order according to the above ratio and the top K-metrics are considered for the next stage ($K = 20$ for our evaluation). Now, for each of these K metrics, AUGURY calculates the contribution of the metric to the high CCV distance before a failure, the intuition being that a metric that is associated with the root cause of the problem will break its correlations to a numerically large degree. Finally, the top 10 metrics are selected for further use in AUGURY, i.e., for predicting failures during the production run of the application. Thus, there is a two-step filtering process which finally leads to the set of metrics to be considered.

Phase 2: Results with Unknown Failures

AUGURY is applied to unknown failures using the reduced set of metrics that are selected from the previous phase. We run AUGURY in no-forecasting mode to eliminate too many noisy peaks in the CCV distance. The result is shown in Figure 7.13 (a). (Notice how distance peaks occur when failures occur. Thresholds are marked with dashed lines. Failures are critical, warning alerts, missing-data and no CCV distance.) Two different thresholds—a *low* and a *high* threshold—for the CCV distance are selected to show the tradeoff between recall and precision. Thresholds are determined visually by the system administrator in the metric-selection phase. To measure true- and false-positives, we cluster AUGURY's alarms (i.e. CCV distances that surpass a threshold) in windows of 2 hours. If multiple AUGURY alarms fit into one window, we consider this as a single positive. A true-positive can be either a *failure prediction*, i.e. an AUGURY alarm before a failure, or a *failure detection*, i.e. an AUGURY alarm at or after the failure point but before a Nagios `recovery` alert. If failure prediction occurs, we measure failure look-ahead time based on the first positive of a window. If a positive is seen after a Nagios `recovery` alert, this is counted as a false positive.

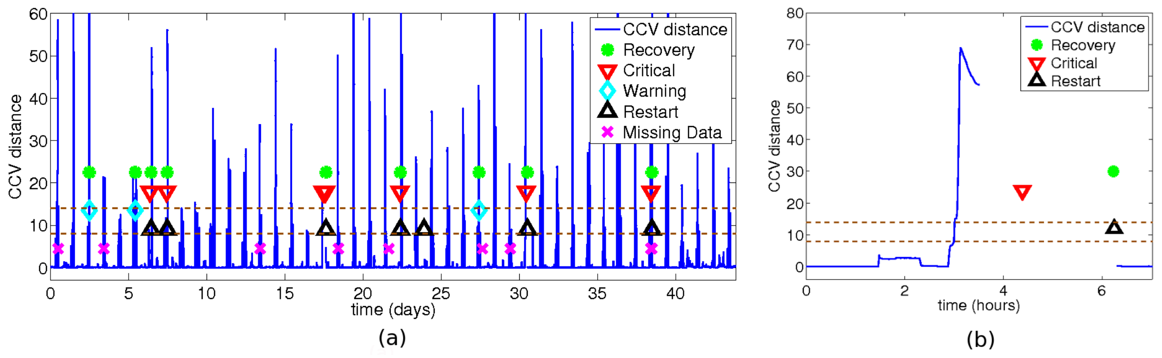


Fig. 7.13.: (a) *CCV* distance for unknown failures of StationsStat; (b) Zoomed-in plot of one of the failures.

Out of 17 failures, AUGURY is able to catch 13 with the low threshold and 11 with the high threshold, giving recalls of 76% and 65% respectively. For all the failures that are caught, *all* of them are predicted, i.e. the CCV distance increases *before* a failure. Look-ahead time is on average 51.01 minutes for the low threshold, and 30.94 minutes for the high threshold.

AUGURY throws a moderate number of false positives (when the CCV distance surpass a threshold when no failures are seen); however they typically occur with a separation of a day. For the low threshold the number of false positives is 74, and for the high threshold it is 43. Since our test period is 44 days, this corresponds to 1-2 false alarms per day on average, so the false alarms arguably do not annoy system administrators to mayhem.

Figure 7.13 (b) shows a zoomed-in plot of the CCV distance in the vicinity of a failure. Notice that the CCV distance increases before the failure. For this particular failure, the metric-collector script is not able to get any measurements from the system (from around time 3.5 to time 6.2). When this happens, the failure is detected from the metric-collector script at around time 3.5. Notice how AUGURY anticipates the failure with a look-ahead time of around 30 minutes when the CCV starts increasing at around time 3. Nagios, however, perceives the failure some time after the metric-collector script (when the critical alert occurs). If the failure is counted from Nagios' perspective, AUGURY has a look-ahead time of more than an hour.

7.7 Discussion

AUGURY uses nearest neighbor distance for outlier detection. There are other techniques that may be considered, such as Support Vector Domain Description (SVDD), a variant of Support Vector Machines that tries to find a hyper-sphere in the feature space which can enclose all the training data (already seen examples of CCVs). Once the hyper-sphere has been built with SVDD, one can test whether a new point x is inside or outside the hyper-sphere. An advantage of using a method like SVDD is that the hyper-sphere can be represented only by the points in the surface, therefore the decision of classifying a point as normal or abnormal can be made more efficiently. A disadvantage is that training SVDD can be quite slow in a high dimensional space (in the order of days for our applications).

Currently AUGURY performs forecasting of each metric independently, using a separate model for each metric. It may be possible to speed up the forecasting process by using multi-variate models through which forecasting of multiple metrics can be done in one step. The balancing factor will be that the multi-variate model should not become so complicated that creating the model or obtaining forecasts from the model is too slow. Thus, we would want to carefully group metrics that we will forecast using multi-variate methods, if at all.

A desirable use case of AUGURY is that a developer is creating slightly changing versions of a system (say, for nightly builds) and realizes that her latest version is buggy. Then, she can run the earlier version to generate the training data points. If the versions differ significantly (such as, two major releases), then the behavior of the metrics will likely be so different that one cannot be used as training data in Augury for detecting the failure point in the other.

Comparison to Related Work. We conducted a literature survey from 2004-2011 from the following conferences and workshops: NSDI, OSDI, SOSP, DSN, ICDCS, Usenix ATC, WASL, SLAML, SysML. For each paper we answered the following questions: Are multiple metrics used simultaneously in the analysis? Are sequences of observations (or temporal patterns) considered? Does it leverage on correlation of multiple metrics? Is failure prediction performed? Does it consider a restricted set of metrics? A summary of the survey is shown in Table 7.4.

	Not considered in the technique:				
Paper and Conference	**multiple metrics** [1]	**observation sequences**	**metric correlations**	**failure prediction**	**broad set of metrics** [2]
Anomaly? App. Change? - Cherkasova (**DSN '08**), *Detect Perf. Anomalies* - Ozonat (**DSN '08**)	X		X	X	X
Perf. Mod. Online Services - Stewart (**NSDI '05**)		X	X	X	X
Spectroscope - Sambasivan (**NSDI '11**), *ConfAid* - Attariyan (**OSDI '10**), *Analyzing Web Logs REBA* - Li (**SLAML '10**)			X	X	X
Measur. Correlations - Gao (**ICDCS '09**), *Tracking Prob. Corr.* - Guo (**DSN '06**), *Profiling Net. Perf.* - Yu (**NSDI '11**)				X	X
Failure Pred. - Salfner (**WASL '08**), *Assurance Soft. Rejuv*. - Avritzer (**DSN '06**)			X		X
Failure Pred. BG/P - Yu (**DSN '11**)					X
Chopstix - Bhatia (**OSDI '08**), *Corr. Failures* - Pertet (**SysML '07**)				X	
Retrieving Sys. History - Cohen (**SOSP '05**), *Ensembles Models* - Zhang (**DSN '05**)		X		X	
Tiresias - Williams (**IPDPS '07**), *Adaptive Aging* - Alonso (**DSN '10**), *Failure Mgmt. DSPS* - Gu (**ICDCS '08**)			X		
Corr. Instru. Data - Cohen (**OSDI '04**)		X			

[1] Multiple metrics are not used “simultaneously”

[2] The set of metrics that is used is restricted (e.g. from one layer), or metrics are handpicked in advance

Table 7.4: Literature survey of NSDI, OSDI, SOSP, DSN, ICDCS, Usenix ATC, WASL, SLAML, SysML from 2004-2011.

8. CONCLUSION

Detection and localization of software bugs and performance anomalies is becoming increasingly challenging as distributed systems grow increasingly larger and complex. Most of the steps in the debugging process of large scale applications are performed manually by the developer, often with tools that do not cope with the massive number of parallel tasks in these systems. An impediment of problem-localization and debugging tools that perform poorly at scale is that many faults only manifest (or manifest more frequently) with a large number of process or with a large amount of input data, in which case these tools are of no practical use—an example of such faults is the the bug that we analyzed in section 5.5.1 which only manifested with around 8,000 MPI tasks or more.

Our statistical modeling of parallel tasks (i.e., Semi-Markov Models), based on control flow and timing information, describes local and global behavior with a convenient granularity to detect a variety of common faults. Further, it requires very small amount of memory as compared to traditional debugging techniques that need to store large traces, e.g., because every program statement is intercepted. *AutomaDeD* creates application states based on executed regions of code rather than based on every instruction or program statement. *AutomaDeD* is a first step towards automating the detection and localization of errors in parallel applications at massive scales.

We have implemented techniques in *AutomaDeD* that enables it to achieve scalability by optimizing it at different levels of its procedures. First, we minimize the time to compare elements of task models by using efficient data structures and approximation methods. Second, we reduce the sizes of the models to an appropriate magnitude, which eliminates noisy dimensions when finding the task affected by a fault. Finally, we use sampling-based techniques such as CAPEK's clustering and scalable nearest neighbor to deal with the increasing number of parallel tasks that are present in today's largest systems. Our implementation scales easily to thousands of tasks and it can identify erroneous tasks and code regions in a few seconds—all the algorithms and data structures in its design allow us to achieve fault detection and diagnosis with a logarithmic complexity in terms of the number of parallel tasks.

By compressing historic control-flow behavior of MPI tasks using Markov models, *AutomaDeD* can identify the least progressed task of a parallel program by inferring probabilistically a progress-dependence graph (or PDG). The PDG is useful in identifying the origin of a variety of performance and correctness faults such as an application's hang or a slow code region. *AutomaDeD* uses backward slicing to pinpoint code that could have led to the unsafe state. Our analysis of a hard-to-diagnose bug in a molecular dynamics simulation code (i.e., ddcMD) and fault injections in two representative large-scale HPC applications demonstrate that *AutomaDeD* identifies these problems with high accuracy, where manual analysis and traditional debugging tools have been unsuccessful. The distributed part of the analysis is performed in a fraction of a second with over 32 thousand tasks. The low analysis cost allows its use online, i.e., during program execution, to automatically localize the origin of performance faults at a granularity of a parallel task and code region.

It has been known, and operationally used, that when problems occur in complex distributed systems, one or more metrics are affected, either in their instantaneous values or in their trends. Here we have shown that faults, which may be missed by mechanisms that analyze individual metrics, affect pair-wise correlations between metrics and this can be used as a trigger for detection. Further, this can enable a system to predict failures by observing the trends in the correlations. Our implementation of AUGURY shows that it is possible to perform the detection and the prediction at runtime even while considering a comprehensive set of metrics, from all layers of the system.

ORION shares some of the design objectives of AUGURY—to monitor a comprehensive set of metrics from all layers of the system—but its purpose is to localize the origin of failures at a granularity of regions of code, rather than to predict failures. ORION pinpoints the metric and a window that is most highly affected by a failure and subsequently highlights the code region that is associated with the origin of the problem. ORION's algorithm models the application behavior through pair-wise correlations of multiple metrics (as in AUGURY), and when failure occurs, it finds the correlations (and hence the metrics) that deviate from normality. Our case studies with four large-scale distributed applications show the utility of the tool — ORION can localize the origin of real-world failures at a granularity of metrics and code regions in the matter of minutes.

9. FUTURE WORK

Our insights from this work open up the possibility of several lines of work:

- ***Are more complex dependencies between metrics (non-linear correlations) more powerful in detection?*** In our current work we assumed linear correlations of metrics, so that, if they are broken, errors or failures can be detected or predicted. Non-linear correlations could fit better several cases of pairs of metrics. As a future work, we could evaluate whether or not using more complex correlation models would improve the accuracy in catching problems.

- ***How should a detection system deal with changing workload patterns, and corresponding discontinuous, but legitimate, changes in the correlation patterns?*** Workload changes could cause our techniques to flag alarms incorrectly because new patterns could make models to deviate from the normal-behavior models (that are constructed previously observed workloads). It could be studied, as a future work, how the detection system can handle changes in the workload by incorporating new "normal" knowledge into its models.

- ***When building SMM graphs for parallel processes, what is the optimal level of compression to obtain maximum accuracy in task isolation?*** We have shown that, as the SMM graph is compressed, accuracy in detecting an abnormal task is increased due dimensionality reduction. However, how much compression is optimal? The trade-off is that compressing the graph too much make us lose the program control-flow structure which can be useful in debugging, however having a too complex graph make us suffer from the curse-of-dimensionality problem—to many unimportant dimensions drown the important dimensions.

- ***Can the strategy in AutomaDeD to create states of the application be generalized?*** *AutomaDeD* creates states that represent regions of code. A state is created when an MPI routine is called by intercepting the call using MPI wrappers. However, the application also executes user-level function that could be used to create these states, and subsequently, to divide the application's execution in finer code regions. One could use sampling strategies to select user-level function calls to instrument. The challenge here is to select the right sampling strategy so that the generated state sizes do not incur high use of memory but at the same time provide sufficient granularity to localize faults accurately.

- ***Can failure prediction using analysis of system metrics be done for HPC applications (as we did it for commercial applications)?*** In principle, we could apply Augury to HPC applications to predict impending failures. However, a fundamental difference between commercial and HPC applications is that it is easier to obtain a large amount of metric samples from a commercial system since they typically operate in a 24/7 basis—our approach for failure prediction requires a large number of metric values to train the statistical models that are used for forecasting. In contrast, runs of HPC applications are shorter than those of commercial systems—in the range of hours (or a few days). A solution could be to sample multiple jobs in an HPC cluster to collect sufficient data to train the models and to use these models for future jobs. The challenge would be to use light weight instrumentation to collect the required metrics without interfering too much with the application.

- ***Can AutomaDeD's approach be applied in large-scale commercial systems such as MapReduce?*** In the MapReduce [12] model for parallel computation, the application workload is divided into several worker tasks by a master task. Modeling the behavior of tasks would follow as in the HPC applications case—using SMMs for each task—however other questions remain unanswered: how are application phases denoted? what is the optimal behavioral clustering configuration? what is the optimal instrumentation level?

LIST OF REFERENCES

LIST OF REFERENCES

[1] "GDB: The GNU Project Debugger." `http://www.gnu.org/software/gdb/`.

[2] Rogue Wave Software, "TotalView Debugger." `http://www.roguewave.com/products/totalview.aspx`.

[3] Allinea Software, "Allinea DDT the Distributed Debugging Tool." `http://www.allinea.com/products/ddt/`.

[4] M. Grottke, L. Li, K. Vaidyanathan, and K. Trivedi, "Analysis of software aging in a web server," *IEEE Transactions on Reliability*, vol. 55, pp. 411 –420, Sep 2006.

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot: A technique for cheap recovery," in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2004.

[6] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira, "Tracking probabilistic correlation of monitoring data for fault detection in complex systems," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 259–268, 2006.

[7] K. Ozonat, "An information-theoretic approach to detecting performance anomalies and changes for large-scale distributed web services," in *IEEE International Conference on Dependable Systems and Networks (DSN)*, june 2008.

[8] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 623 –630, june 2009.

[9] 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 *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pp. 16–16, 2004.

[10] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pp. 105–118, 2005.

[11] Java Community Process Program, "Jsr-000316 java (tm) platform, enterprise edition (java ee) 6." `http://jcp.org/aboutJava/communityprocess/pr/jsr316/`, Feb 2009.

[12] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation (OSDI)*, USENIX Association, 2004.

[13] Message Passing Interface Forum, "Mpi: A message-passing interface standard, version 3.0." `http://www.mpi-forum.org/docs/`, Sep 2012.

[14] Q. Gao, W. Zhang, and F. Qin, "FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking," in *ACM/IEEE Supercomputing Conference (SC)*, 2010.

[15] Q. Gao, F. Qin, and D. K. Panda, "DMTracker: Finding Bugs in Large-scale Parallel Programs by Detecting Anomaly in Data Movements," in *ACM/IEEE Supercomputing Conference (SC)*, 2007.

[16] A. Mirgorodskiy, N. Maruyama, and B. Miller, "Problem Diagnosis in Large-Scale Computing Environments," in *ACM/IEEE Supercomputing Conference (SC)*, pp. 11–23, 2006.

[17] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, "Scalable temporal order analysis for large scale debugging," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp. 44:1–44:11, 2009.

[18] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit, "Lessons Learned at 208K: Towards Debugging Millions of Cores," in *ACM/IEEE Supercomputing Conference (SC)*, pp. 1–9, IEEE Press, 2008.

[19] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. de Supinski, B. P. Miller, and M. Schulz, "Benchmarking the Stack Trace Analysis Tool for BlueGene/L," in *International Conference on Parallel Computing: Architectures, Algorithms and Applications (ParCo)*, 2007.

[20] P. C. Roth, D. C. Arnold, and B. P. Miller, "Mrnet: A software-based multicast/reduction network for scalable tools," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC)*, SC '03, 2003.

[21] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems," in *International Conference on Dependable Systems and Networks (DSN)*, june-1 july 2005.

[22] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pp. 111–124, 2010.

[23] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: finding liveness bugs in systems code," in *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, (Berkeley, CA, USA), pp. 18–18, USENIX Association, 2007.

[24] M. S. Musuvathi, D. Park, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "Cmc: A pragmatic approach to model checking real code," in *In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, OSDI '02, 2002.

[25] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 265–278, ACM, 2011.

[26] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "Modist: transparent model checking of unmodified distributed systems," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pp. 213–228, 2009.

[27] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3s: debugging deployed distributed systems," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, (Berkeley, CA, USA), pp. 423–437, USENIX Association, 2008.

[28] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, (Washington, DC, USA), pp. 149–158, IEEE Computer Society, 2009.

[29] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 26–26, USENIX Association, 2012.

[30] S. Sabato, E. Yom-Tov, A. Tsherniak, and S. Rosset, "Analyzing system logs: a new view of what's important," in *Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, SYSML'07, (Berkeley, CA, USA), pp. 6:1–6:7, USENIX Association, 2007.

[31] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, (New York, NY, USA), pp. 117–132, ACM, 2009.

[32] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 18–18, USENIX Association, 2004.

[33] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based faliure and evolution management," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI)*, NSDI'04, 2004.

[34] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2011.

[35] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: automated classification of performance crises," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pp. 111–124, 2010.

[36] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, (New York, NY, USA), pp. 105–118, ACM, 2005.

[37] J. Gao, G. Jiang, H. Chen, and J. Han, "Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems," in *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, (Washington, DC, USA), pp. 623–630, IEEE Computer Society, 2009.

[38] S. Zhang, I. Cohen, J. Symons, and A. Fox, "Ensembles of models for automated diagnosis of system performance problems," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, (Washington, DC, USA), pp. 644–653, IEEE Computer Society, 2005.

[39] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure Trends in a Large Disk Drive Population," in *5th USENIX Conference on File and Storage Technologies (FAST '06)*, p. 1728, 2007.

[40] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "Bluegene/l failure analysis and prediction models," in *International Conference on Dependable Systems and Networks (DSN '06)*, pp. 425–434, 2006.

[41] Y. Zhao, X. Liu, S. Gan, and W. Zheng, "Predicting disk failures with hmm- and hsmm-based approaches," in *Proceedings of the 10th industrial conference on Advances in data mining: applications and theoretical aspects*, ICDM'10, pp. 390–404, 2010.

[42] R. W. Featherstun and E. W. Fulp, "Using syslog message sequences for predicting disk failures," in *Proceedings of the 24th USENIX international conference on Large installation system administration (LISA)*, LISA'10, pp. 1–10, USENIX Association, 2010.

[43] A. Williams, S. Pertet, and P. Narasimhan, "Tiresias: Black-box failure prediction in distributed systems," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, march 2007.

[44] N. Laranjeiro, M. Vieira, and H. Madeira, "Database systems for advanced applications," ch. Predicting Timing Failures in Web Services, pp. 182–196, 2009.

[45] K. Lindekugel, A. DiGirolamo, and D. Stanzione, "Architecture for an Offline Parallel Debugger," in *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 227–235, Dec 2008.

[46] J. Lourenço and J. C. Cunha, "Fiddle: A Flexible Distributed Debugging Architecture," in *International Conference on Computational Science (ICCS)-Part II*, pp. 821–830, Springer-Verlag, 2001.

[47] MPIPlugin, "MPI Plugin for KDevelop." `http://sourceforge.net/projects/mpiplugin/`.

[48] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS Parallel Benchmarks," RNR-91-002, NASA Ames Research Center, Aug. 1991.

[49] M. Schulz and B. R. de Supinski, "$P^N MPI$ Tools: A Whole Lot Greater than the Sum of Their Parts," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), pp. 1–10, ACM, 2007.

[50] B. W. Silverman, *Density Estimation for Statistics and Data Analysis.* Chapman & Hall, 1986.

[51] C. D. Manning and H. Schtze, *Foundations of Statistical Natural Language Processing.* Cambridge, Mass: MIT Press, 1999.

[52] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data Clustering: A Review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264–323, 1999.

[53] B. G. Mirkin, *Mathematical Classification and Clustering.* Kluwer Academic Press, 1996.

[54] MVAPICH Project, "MVAPICH Discussion List." `http://mail.cse.ohio-state.edu/pipermail/mvapich-discuss/2007-July/0009%32.html`.

[55] Rogue Wave Software, "Totalview achieves massive milestone towards exascale debugging - totalview debugs 786,432 processor cores as part of scalability initiative." `http://www.roguewave.com/company/news-events/press-releases/2012/scalability%-milestone-for-totalview-debugger.aspx`, Nov 2012.

[56] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem Diagnosis in Large-Scale Computing Environments," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006.

[57] T. Gamblin, B. R. de Supinski, M. Schulz, R. Fowler, and D. A. Reed, "Clustering Performance Data Efficiently at Massive Scales," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, (New York, NY, USA), pp. 243–252, ACM, 2010.

[58] V. E. Henson and U. M. Yang, "BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner," *Appl. Numer. Math.*, vol. 41, no. 1, pp. 155–177, 2002.

[59] J. M. Linacre, "Overlapping Normal Distributions," *Rasch Measurement Transactions*, vol. 10, no. 1, pp. 487–8, 1996.

[60] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics).* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[61] D. Pelleg and A. Moore, "X-Means: Extending K-Means with Efficient Estimation of the Number of Clusters," in *Proceedings of the 17th International Conf. on Machine Learning*, pp. 727–734, 2000.

[62] E. W. Forgy, "Cluster Analysis of Multivariate Data: Efficiency *vs.* Interpretability of Classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.

[63] S. P. Lloyd, "Least Squares Quantization in PCM. Technical Note, Bell Laboratories," *IEEE Transactions on Information Theory*, vol. 28, pp. 128–137, 1967, 1982.

[64] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations," in *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* (L. M. Le Cam and J. Neyman, eds.), vol. 1, pp. 281–297, Univeristy of California Press, June 21-July 18 1967.

[65] G. Bronevetsky, I. Laguna, S. Bagchi, B. de Supinski, D. Ahn, and M. Schulz, "AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks," in *2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 231 –240, 2010.

[66] "Algebraic MultiGrid (AMG) 2006 Benchmark." `https://asc.llnl.gov/sequoia/benchmarks`.

[67] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449, 1981.

[68] F. H. Streitz, J. N. Glosli, M. V. Patel, B. Chan, R. K. Yates, B. R. de Supinski, J. Sexton, and J. A. Gunnels, "Simulating solidification in metals at high pressure: The drive to petascale computing," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 254, 2006.

[69] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, pp. 49–66, 2005.

[70] W. Haque, "Concurrent deadlock detection in parallel programs," *International Journal of Computers and Applications*, vol. 28, pp. 19–25, January 2006.

[71] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, "A graph based approach for mpi deadlock detection," in *International conference on Supercomputing (ICS)*, pp. 296–305, 2009.

[72] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of mpi applications with umpire," in *ACM/IEEE Supercomputing Conference (SC)*, 2000.

[73] "DynInst - An Application Program Interface (API) for Runtime Code Generation." `http://www.dyninst.org/`.

[74] M. Kamkar and P. Krajina, "Dynamic slicing of distributed programs," in *International Conference on Software Maintenance*, pp. 222 –229, oct 1995.

[75] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987.

[76] "Boost C++ libraries." `http://www.boost.org/`.

[77] B. Korel and J. Laski, "Dynamic slicing of computer programs," *Journal of Systems and Software*, vol. 13, pp. 187–195, Dec. 1990.

[78] M. Kamkar, P. Krajina, and P. Fritzson, "Dynamic slicing of parallel message-passing programs," in *Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing, 1996. PDP '96.*, pp. 170 –177, jan 1996.

[79] J. Rilling, H. Li, and D. Goswami, "Predicate-based dynamic slicing of message passing programs," in *Second IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 133 – 142, 2002.

[80] G. Shanmuganathan, K. Zhang, E. Wong, and Y. Qi, "Analyzing message-passing programs through visual slicing," in *International Conference on Information Technology: Coding and Computing (ITCC)*, vol. 2, pp. 341 – 346 Vol. 2, april 2005.

[81] M. Strout, B. Kreaseck, and P. Hovland, "Data-flow analysis for mpi programs," in *International Conference on Parallel Processing (ICPP)*, pp. 175 –184, aug. 2006.

[82] "ASC Sequoia Benchmark Codes." `https://asc.llnl.gov/sequoia/benchmarks/`.

[83] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: a pervasive network tracing framework," in *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, pp. 20–20, 2007.

[84] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: global comprehension for distributed replay," in *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI'07, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2007.

[85] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, (Berkeley, CA, USA), pp. 27–27, USENIX Association, 2006.

[86] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006.

[87] J. L. Hellerstein, F. Zhang, and P. Shahabuddin, "A statistical approach to predictive detection," *Computer Networks*, vol. 35, no. 1, pp. 77–95, 2001.

[88] K. Ozonat, "An information-theoretic approach to detecting performance anomalies and changes for large-scale distributed web services," in *IEEE International Conference on Dependable Systems and Networks (DSN)*, june 2008.

[89] "Apache HBase." `http://hbase.apache.org/`.

[90] "Apache Hadoop Project." `http://hadoop.apache.org/`.

[91] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? application change? or workload change? towards automated detection of application performance anomaly and change," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 452 –461, 2008.

[92] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI)*, NSDI'04, 2004.

[93] "Android issue 4825." `http://code.google.com/p/android/issues/detail?id=4825`.

[94] A. T. Tai and K. S. Tso, "A performability-oriented software rejuvenation framework for distributed applications," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, (Washington, DC, USA), 2005.

[95] C. Chatfield, *The Analysis of Time Series: An Introduction, Sixth Edition.* Chapman and Hall/CRC, 2003.

[96] "Sysstat utilities. http://sebastien.godard.pagesperso-orange.fr/."

[97] "The gnu r project. http://www.r-project.org."

[98] "Nist engineering statistics handbook (box-jenkins models). http://www.itl.nist.gov/div898/handbook/."

[99] "Rubis auction site prototype . http://rubis.ow2.org/."

[100] C. Stewart and K. Shen, "Performance modeling and system management for multi-component online services," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, NSDI'05, pp. 71–84, 2005.

[101] D. A. Menasce, "Tpc-w: a benchmark for e-commerce," *IEEE Internet Computing*, vol. 6, no. 3, pp. 83–87, 2002.

[102] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Injecting realistic burstiness to a traditional client-server benchmark," in *Proceedings of the 6th international conference on Autonomic computing*, ICAC '09, pp. 149–158, 2009.

[103] "Android emulator. http://developer.android.com."

[104] "Android issue 15554." `http://code.google.com/p/android/issues/detail?id=15554`.

VITA

VITA

Ignacio Laguna received his BSc degree in Electronics and Communication Engineering from the Universidad de Panama in August 2002. He obtained his MSc degree from Purdue University in the School of Electrical and Computer Engineering on August 2008. He began working on his PhD on September 2008 under the supervision of Professor Saurabh Bagchi in the same school. His research interests include fault detection, problem localization and machine learning for anomaly detection. He spent two terms (Spring 2009 and Fall 2010) at the Lawrence Livermore National Laboratory in California working in fault detection and diagnosis techniques for high-performance computing (HPC) applications. He received the ACM & IEEE George Michael Memorial HPC Fellowship in 2011; this award honors exceptional PhD students throughout the world whose research focus area is HPC.

PUBLICATIONS

Conference Papers

1. I. Laguna, D. H. Ahn, B. R. de Supinski, S. Bagchi, T. Gamblin, *Probabilistic Diagnosis of Performance Faults in Large-Scale Parallel Applications*, International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, Sep, 2012.

2. G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, *Automatic Fault Characterization via Abnormality-Enhanced Classification*, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Boston, Massachusetts, Jun, 2012.

3. I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, Barry Rountree, *Large Scale Debugging of Parallel Tasks with AutomaDeD*, ACM/IEEE Conference on Supercomputing 2011 (SC), Seattle, WA, Nov 2011.

4. G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, and M. Schulz, *Statistical Fault Detection for Parallel Applications with AutomaDeD*, 6th IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE), Stanford, CA, Mar 23-24, 2010.

5. G. Bronevetsky*, I. Laguna*, S. Bagchi, B. R. de Supinski, D. H. Ahn, M. Schulz, *AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks*, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Chicago Illinois, 2010. (* co-first authors)

6. I. Laguna, F. A. Arshad, D. M. Grothe, S. Bagchi, *How To Keep Your Head Above Water While Detecting Errors*, ACM/IFIP/USENIX 10th International Middleware Conference (Middleware), UIUC Illinois, Nov-Dec 2009.

7. D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz, *Scalable Temporal Order Analysis for Large Scale Debugging*, ACM/IEEE Conference on Supercomputing 2009 (SC), Portland, OR, Nov 2009.
8. G. Khanna, I. Laguna, F. A. Arshad, S. Bagchi, *Distributed Diagnosis of Failures in a Three Tier E-Commerce System*, 26th IEEE Symposium on Reliable Distributed Systems (SRDS), Beijing, China, Oct 2007.
9. G. Khanna, I. Laguna, F. A. Arshad, S. Bagchi, *Stateful Detection in High Throughput Distributed Systems*, 26th IEEE Symposium on Reliable Distributed Systems (SRDS), Beijing, China, Oct 2007.

Posters / Short Abstracts

1. Scalable Detection of Anomalous Parallel Tasks with AutomaDeD, Poster abstract at the Conference of Dependable Systems and Networks (DSN), Boston, Jun, 2012.
2. Scalable Error Detection and Failure Prediction in Large-Scale Applications, Poster abstract at Postdoc Research Symposium, Argonne National Laboratory, Chicago, Oct 27, 2011.
3. Stateful error detection in high throughput applications, Poster abstract at the ACM/IFIP/USENIX 10th International Middleware Conference, UIUC Illinois, Dec 2, 2009.