Sandboxed, Online Debugging of Production Bugs with No Overhead

Nipun Arora

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2016

ABSTRACT

Sandboxed, Online Debugging of Production Bugs with No Overhead

Nipun Arora

Software debugging is the process of localizing, and finding defects that were observed in a system. In a world of increasingly complex and large scale distributed systems, errors can be very subtle, and can have a compounded impact on the application. Unfortunately, replicating production bugs of complex large scale systems in an offline environment continues to be a substantial challenge. In particular, production systems bugs can be a result of complex interactions between multiple system components, and can cause faults either in the kernel, middleware or the application itself. Hence it is important, to be able to gain insight in the entire workflow of the system, both breadth-wise (across application tiers and network boundaries), and depth wise (across the execution stack from application to kernel).

In addition to the inherent complexity in software debugging, it is also essential to have a short time to bug diagnosis to reduce the financial impact of any error. Recent trends towards DevOps [OReilly, DevOps,], and Agile [Martin, 2003] software engineering paradigms further emphasize the need of having shorter debug cycles. DevOps stresses on close coupling between software developers and operators, and to merge the operations of both. Most companies which have adopted DevOps have very frequent releases and hence require a very short time to a bug fix, test, patch and release in order to realize continuous delivery (Facebook mobile has 2 releases a day, and Flickr has 10 deployment cycles per day). Similarly agile programming has shorter development cycles called *sprints*, which focus on faster releases, and quick debugging.

Existing debugging mechanisms provide light-weight instrumentation which can track execution flow in the application by instrumenting important points in the application code. These are followed by inference based mechanisms to find the root-cause of the problem. While such techniques are useful in getting a clue about the bug, they are limited in their ability to discover the root-

cause. Another body of work uses record-and-replay infrastructures, which record the execution and then replay the execution offline. While record and replay infrastructures generate a high fidelity representative framework for bug diagnosis, they suffer a heavy overhead generally not acceptable in user-facing production systems.

Therefore, to meet the demands of a low-latency distributed computing environment of modern service oriented systems, it is important to have debugging tools which have *minimal to negligible impact* on the application, and can provide a fast update to the operator to allow for *shorter time to debug*. To this end, we introduce a new debugging paradigm called *live debugging*. There are two goals that any *live debugging* infrastructure must meet: Firstly, it must offer real-time insight and bug diagnosis and localization, which is paramount when errors happen in service-oriented-application. Several modern day 24*7 applications have developers serving as operators who are available in *shifts* at all times to tackle any problems that occur in the system. Having a shorter debug cycles and quicker patches is essential to ensure application quality, reliability and reduce the financial impact on the application. Secondly, *live debugging* should not impact user-facing performance for non bug triggering events. In large distributed applications, non-crashing bugs which impact only a small percentage of users are common. In such scenarios, debugging a small part of the application should not impact the entire system.

With the above stated goals in mind, we have designed a platform called *Parikshan*, which leverages user-space containers (OpenVZ/ LXC) to launch application instances for the express purpose of *live debugging*. *Parikshan* is driven by a live-cloning process, which generates a replica (*debug container*) of production services for debugging or testing, cloned from a *production container* which provides the real output to the user. The *debug container* provides a sandbox environment, for safe execution of test-cases/debugging done by the users without any perturbation to the execution environment. As a part of this framework, we have designed customized-network proxy agents, which replicate inputs from clients to both the production and test-container, as well safely discard all outputs from the test-container. Together the network proxy, and the debug container ensure both compute and network isolation of the debugging environment, while at the same time allowing the user to debug the application. This thesis also looks into *light-weight instrumentation techniques*, which can complement our *live debugging* environment. Additionally, we will demonstrate a *statistical debugging mechanism* that can be applied in the debug-container to gain insight and localize the

error in real-time. We believe that this piece of work provides the first of it's kind practical real-time debugging of large multi-tier and cloud applications, without requiring any application down-time, and minimal performance impact.

The principal hypothesis of this dissertation is that, for large-scale service-oriented-applications(SOA) it is possible to provide a *live debugging* environment, which allows the developer to debug the target application without impacting the performance of the production system. Primarily, we will present an approach for *live debugging* of production systems. This involves discussion of *Parikshan* framework which forms the backbone of this dissertation. We will discuss how to clone the containers, split and isolate network traffic, and aggregate it for communication to both upstream and downstream tiers, in a multi-tier SOA infrastructure. As a part of this description, we will also show case-studies demonstrating how network replay is enough for triggering most bugs in real-world application. To show this, we have presented 16 real-world bugs, which were triggered using our network duplication techniques. Additionally, we present a survey of 270+ bugs from bug reports of SOA applications which were found to be similar to the 16 mentioned above.

Secondly, we will present `iProbe` a new type of instrumentation framework, which uses a combination of static and dynamic instrumentation, to have an order-of-magnitude better performance than existing instrumentation techniques. While *live debugging* does not put any performance impact on the production, it is still important to have the debug-container as much in sync with the production container as possible. `iProbe` makes applications live debugging friendly, and provides an easy way for the debuggers to apply probes in the debugging sandboxed environment.

Lastly, while *Parikshan* is a platform to quickly attack bugs, in itself it's a debugging platform. For the last section of this dissertation we look at how various existing debugging techniques can be adapted to *live debugging*, making them more effective. We first enumerate scenarios in which debugging can take place: *post-facto* - turning livedebugging on after a bug has occured, *proactive* - having debugging on before a bug has happened. We then introduce two new approaches: *active debugging*, which will allow developers to apply patches/fix or do testing in parallel to the production container, and *budget-limited instrumentation* which will allow longer continuous debugging with a controlled overhead.

Table of Contents

1 Introduction **1**

1.1 Definitions . . . 4

1.2 Problem Statement . . . 6

1.3 Requirements . . . 7

1.4 Scope . . . 7

1.4.1 Service Oriented Applications . . . 8

1.4.2 Non-Crashing Bugs . . . 8

1.4.3 Native Applications . . . 8

1.5 Proposed Approach . . . 9

1.6 Hypothesis . . . 10

1.7 Assumptions . . . 11

1.7.1 Resource Availability . . . 11

1.8 Outline . . . 11

2 Background and Motivation **13**

2.1 Recent Trends . . . 13

2.1.1 Software development trends . . . 13

2.1.2 Microservice Architecture . . . 15

2.1.3 Virtualization, Scalability and the Cloud . . . 16

2.2 Current debugging of production systems . . . 17

2.3 Motivating Scenario . . . 18

2.4 Summary . . . 19

I *Parikshan*: A framework for sandboxed, online debugging of production bugs with no overhead **21**

3 Parikshan **22**

3.1 Introduction . . . 22

3.2 *Parikshan* . . . 25

3.2.1 Clone Manager . . . 25

3.2.2 Network Proxy Design Description . . . 27

3.2.3 Debug Window . . . 30

3.2.4 Divergence Checking . . . 32

3.2.5 Implementation . . . 32

3.3 Evaluation . . . 33

3.3.1 Live Cloning Performance . . . 33

3.3.2 Debug Window Size . . . 37

3.3.3 A survey of real-world bugs . . . 40

3.4 Applications of Live Debugging . . . 41

3.5 Discussion and Limitations . . . 42

3.5.1 Non-determinism: . . . 42

3.5.2 Distributed Services: . . . 43

3.6 Summary . . . 43

4 Is network replay enough? **45**

4.1 Overview . . . 45

4.2 Case Studies . . . 46

4.2.1 Semantic Bugs . . . 46

4.2.2 Performance Bugs . . . 50

4.2.3 Resource Leaks . . . 52

4.2.4 Concurrency Bugs . . . 54

4.2.5 Configuration Bugs . . . 57

4.3 Summary . . . 58

II `iProbe`: Creating live debugging friendly applications **59**

5 iProbe **60**

5.1 Introduction . . . 60
5.2 Design . . . 63
5.2.1 ColdPatching Phase . . . 63
5.2.2 HotPatching Phase . . . 64
5.2.3 Safety Checks for iProbe . . . 67
5.2.4 Extended iProbe Mode . . . 68
5.2.5 Extended iProbe Mode . . . 69
5.3 Trampoline vs. Hybrid Approach . . . 71
5.4 Implementation . . . 73
5.4.1 iProbe Framework . . . 73
5.4.2 FPerf: An iProbe Application for Hardware Event Profiling . . . 75
5.4.3 Safety Checks for iProbe . . . 76
5.5 Evaluation . . . 77
5.5.1 Overhead of ColdPatch . . . 77
5.5.2 Overhead of HotPatching and Scalability Analysis . . . 78
5.5.3 Case Study: Hardware Event Profiling . . . 79
5.6 Summary . . . 83

III Active Debugging and Applications of Live Debugging **84**

6 Active Debugging and Applications of Live Debugging **85**

6.1 Overview . . . 85
6.2 Live Debugging using Parikshan . . . 87
6.3 Debugging Scenarios . . . 90
6.3.1 Scenario 1: Post-Facto Analysis . . . 90
6.3.2 Scenario 2: Proactive Analysis . . . 91
6.4 Existing Debugging Mechanisms . . . 92
6.4.1 Execution Tracing . . . 92

6.4.2 Statistical Debugging . . . 93
6.4.3 Staging Record and Replay . . . 93
6.4.4 A-B Testing . . . 95
6.4.5 Interactive Debugging . . . 95
6.5 Budget Limited, Adaptive Instrumentation . . . 96
6.5.1 Proactive: Modeling Budgets . . . 97
6.5.2 Extended Load-balanced duplicate clones . . . 100
6.5.3 Simulation . . . 100
6.5.4 Reactive: Adaptive Instrumentation . . . 100
6.5.5 Automated Reactive Scores . . . 102
6.5.6 Feasibility . . . 102
6.6 Active Debugging . . . 103
6.7 Summary . . . 105

IV Related Work **106**

7 Related Work **107**
7.1 Related Work for Parikshan . . . 107
7.1.1 Record and Replay Systems: . . . 107
7.1.2 Decoupled or Online Analysis . . . 107
7.1.3 Real-Time techniques: . . . 108
7.1.4 Live Migration & Cloning . . . 108
7.1.5 Large Scale Software Debugging . . . 109
7.1.6 Software Programming Paradigms . . . 109
7.1.7 Virtualization . . . 109
7.2 Related Work for iProbe . . . 109

V Conclusions **112**

8 Conclusions **113**
8.1 Contributions . . . 113

8.2 Future Work 113
8.2.1 Immediate Future Work 114
8.2.2 Possibilities for Long Term 115
8.3 Conclusion 115

VI Bibliography **116**

Bibliography **117**

List of Figures

2.1 Devops is a new programming paradigm 14

2.2 An example of a microservice architecture for a car renting agency website 16

2.3 Live Debugging aims to move debugging part of the lifecycle to be done in parallel to the running application, as currently modeling, analytics, and monitoring is done 18

2.4 Workflow of *Parikshan* in a live multi-tier production system with several interacting services. When the administrator of the system observes errors in two of it's tiers, he can create a sandboxed clone of these tiers and observe/debug them in a sandbox environment without impacting the production system. 19

3.1 **High level architecture of *Parikshan***, showing the main components: Network Duplicator, Network Aggregator, and Cloning Manager. The replica (debug container) is kept in sync with the master (production container) through network-level record and replay. In our evaluation, we found that this light-weight procedure was sufficient to reproduce many real bugs. 25

3.2 External and Internal Mode for live cloning: P1 is the production, and D1 is the debug container, the clone manager interacts with an agent which has drivers to implement live cloning. 26

3.3 Suspend time for live cloning, when running a representative benchmark 34

3.4 Live Cloning suspend time with increasing amounts of I/O operations 35

3.5 Simulation results for debug-window size. Each series has a constant arrival rate, and the buffer is kept at 64GB. 38

5.1 The Process of ColdPatching. 63

5.2 Native Binary, the State Transition of ColdPatching and HotPatching. 64
5.3 HotPatching Workflow. 65
5.4 Traditional Trampoline based Dynamic Instrumentation Mechanisms. 71
5.5 Overview of *FPerf*: Hardware Event Profiler based on iProbe. 74
5.6 Overhead of iProbe "ColdPatch Stage" on SPEC CPU 2006 Benchmarks. 78
5.7 Overhead and Scalability Comparison of iProbe HotPatching vs. SystemTap vs. DynInst using a Micro-benchmark. 80
5.8 The number of different functions that have been profiled in one execution. 81
5.9 Overhead Control and Number of Captured Functions Comparison. 82

6.1 Staged Record and Replay using *Parikshan* 94
6.2 Traditional A-B Testing 95
6.3 *Parikshan* applied to a mid-tier service 97
6.4 External and Internal Mode for live cloning: P1 is the production, and D1 is the debug container. 97
6.5 This figure shows how queueing theory can be extended to a load balanced debugging scenario. Here each of the debug container receive the requests at rate λ, and the total instrumentation is balanced across multiple debug containers. 99
6.6 Reactive Instrumentation 101
6.7 Offline Debugging 103
6.8 Debugging strategies for offline debugging 103
6.9 Active Debugging 104
6.10 Debugging Strategies for Active Debugging 104

7.1 Advantages of iProbe over existing monitoring frameworks DTrace/SystemTap and DynInst 109

List of Tables

3.1 Approximate debug window sizes for a MySQL request workload 37

3.2 Survey and classification of bugs 41

4.1 List of real-world production bugs studied with *Parikshan* 46

5.1 Experiment Platform. 80

Acknowledgments

First and foremost, I would like to thank my advisor, Gail Kaiser, who has provided me invaluable guidance and wisdom in all matters related to my research and publishing. I truly appreciate the support, and am thankful for the flexibility and patience over the years in providing guidance in my research efforts.

I would also like to thank Franjo Ivancic, who has been a constant source of support and provided valuable feedback to my half-baked ideas. A special thank you to Jonathan Bell, who provided insightful critique, and ideas which helped in re-shaping and significantly improving the work presented in this thesis. I would also like to thank Swapneel Sheth, and Chris Murphy who provided valuable support in terms of direction and practical experience during my initial years of the PhD.

The work described here has been supported by the Programming Systems Laboratory, and several members and associates of the PSL lab. I would like to thank Mike Su, Leon Wu, Simha Sethumadhavan, Ravindra Babu Ganapathi, and Jonathan Demme, and many others whom I had the pleasure to work with. I also had the pleasure of supervising a number of graduate students who have helped in testing out several early stage prototypes of different projects I was involved in.

I would also like to acknowledge my co-workers at NEC Labs America, I have had the pleasure to work with several extremely intelligent academicians, and have gained a lot from my experience in working with them. In particular I would like to thank - Hui Zhang, Junghwan Rhee, Cristian Lumezanu, Abhishek Sharma, Vishal Singh, Qiang Xu and several interns. I would also like to thank Kai Ma for his work towards the implementation of *FPerf*, an application of `iProbe`.

Last but most important, I would like to thank my wife, my parents and my sister for sticking

with me, and their emotional support and encouragement.

To my parents, and my sister for their unwavering support, and to my wife for her constant encouragement and the final push.

Chapter 1

Introduction

Although software bugs are nothing new, the complexities of virtualized environments coupled with large distributed systems have made bug localization harder. The large size of distributed systems means that any downtime has significant financial penalties for all parties involved. Hence, it is increasingly important to localize and fix bugs in a very short period of time.

Existing state-of-art techniques for monitoring production systems [McDougall *et al.*, 2006; Park and Buch, 2004; Prasad *et al.*, 2005] rely on light-weight dynamic instrumentation to capture execution traces. Operators then feed these traces to analytic tools [Barham *et al.*, 2004; Zhang *et al.*, 2014] to find the root-cause of the error. However, dynamic instrumentation has a trade-off between granularity of tracing and the performance overhead. Operators keep instrumentation granularity low, to avoid higher overheads in the production environment. This often leads to multiple iterations between the debugger and the operator, to increase instrumentation in specific modules, in order to diagnose the root-cause of the bug.

Another body of work has looked into record-and-replay [Altekar and Stoica, 2009; Dunlap *et al.*, 2002; Laadan *et al.*, 2010; Geels *et al.*, 2007] systems which capture the log of the system, in order to faithfully replay the trace in an offline environment. Replay systems try and capture system level information, user-input, as well as all possible sources of non-determinism, to allow for in-depth *post-facto* analysis of the error. However, owing to the amount of instrumentation required, record-and-replay tools deal with an even heavier overhead, making them impractical for real-world production systems.

This thesis is centered around the idea of a new debugging paradigm called *"live debugging"*,

whereby developers can do in-situ debugging without impacting the performance of the application. The key idea behind this approach is to give faster time-to-bug localization, deeper insight into the health and activity within the system, and to allow operators to dynamically debug applications without fear of changing application behavior. We leverage existing work in live migration and light-weight user-space container virtualization, to provide an end-to-end workflow for real-time debugging.

Our work is inspired by three key observations in modern service oriented applications. Firstly, we observe that most service oriented applications(SOA) are launched on cloud based infrastructures. These applications use virtualization to share physical resources, maintained by third-party vendors like Amazon EC2 [Amazon, 2010], or google compute [gco,] platforms. Furthermore, there is an increasing trend towards light-weight user-space container virtualization, which is less resource hungry, and makes sharing physical resources easier. Frameworks like docker [Merkel, 2014] allow for scaled out application deployment, by allowing each application service instance to be launched in it's own container. For instance, an application server, and a database server making up a web-service, can be hosted on their own containers, thereby sandboxing each service, and making it easier to scale out.

Secondly, we observe a trend towards DevOps [OReilly, DevOps,] by the software engineering industry. DevOps stresses on close coupling between software developers and operators, in order to have shorter release cycles (Facebook mobile has 2 releases a day, and Flickr has 10 deployment cycles per day [Allspaw J., 2009]). This re-emphasizes the need to have a very short time to diagnose and fix a bug especially in service oriented application. We believe by providing a means to observe the application when the bug is active, we will significantly reduce the time to bug localization.

Lastly, our key insight is that for most service-oriented applications (SOA), a failure can be reproduced simply by replaying the network inputs passed on to the application. For these failures, capturing very low-level sources of non-determinism (e.g. thread scheduling or general system calls, often with high overhead) is unnecessary to successfully and automatically reproduce the buggy execution in a development environment. We have evaluated this insight by studying 16 real-world bugs, which we were able to trigger by only duplicating and replaying network packets. Furthermore we categorized 217 bugs from three real-world applications, finding that most were similar in nature to the 16 that were reproduced, suggesting that our approach would be applicable to them as well.

This thesis will make the following contributions:

First, in Chapter 3 we will present a framework for "live debugging" applications while they are running in the production environment. This will involve a description of our system called *Parikshan*[1], which allows **real-time debugging** without any performance impact on the production service. We provide a facility to sandbox the production and debug environments so that any modifications in the debug environment do not impact user-facing operations. *Parikshan* avoids the need of large test-clusters, and can target specific sections of a large scale distributed application. In particular, *Parikshan* allows debuggers to apply debugging techniques with deeper granularity instrumentation, and profiling, without impacting application performance.

This chapter will also present details of our case-study presenting real-world bugs which were triggered by network input alone, and which show why using *Parikshan* would be enough to capture most real-world bugs. Each case study presents a different variety of bugs from the following classes: performance, semantic, non-deterministic, configuration and resource leak. We believe that these bugs form the most common classification of bugs in service oriented applications.

Second, in Chapter 5 we will present a dynamic instrumentation mechanism called `iProbe`, which can assist developers in making applications *Parikshan* ready. iProbe uses a novel two-stage design, and offloads much of the dynamic instrumentation complexity to an offline compilation stage. It leverages standard compiler flags to introduce "place-holders" for hooks in the program executable. Then it utilizes an efficient user-space "HotPatching" mechanism which modifies the functions to be traced and enables execution of instrumented code in a safe and secure manner.

In the final chapter 6 of this thesis we focus on applications of live debuggging. In particular we discuss several existing techniques and how they can be coupled with live debugging. We discuss step-by-step scenarios where debugging on the fly can be helpful, and how it can be applied.

We also briefly introduce two new techniques: Firstly, a **budget limited instrumentation** technique for live debugging. This technique leverages existing work on statistical debugging, and queuing theory to lay a statistical foundation for allocating buffer sizes and various configuration parameters. It proposes a reactive mechanism to adapt to the overhead of instrumentation bounds using sampling techniques. Secondly, we introduce the concept of active debugging. This allows for debuggers to add a patch/or a fix in the debug-container and check if it fixes the error. Similarly

[1] *Parikshan* is the Sanskrit word for testing

tests can be performed while ensuring the production container output is not changed, and forward progress is ensured. We isolate tests in *debug container* to ensure forward progress, and no impact on the production-container.

The rest of this section is organized as follows. Firstly in section 1.1 we define terms and terminologies used in the rest of this thesis. Section 1.2 further defines the scope of our problem statement, definitions, and classifications of the bugs. Section 1.3 illustrates the requirements this thesis must meet. Next, in section 1.4 we define the scope of the techniques presented in this thesis. Section 1.5 briefly goes over the proposed approach presented in this thesis. In section 1.6 we give the hypothesis of this thesis. Section 1.7 lists some of the assumptions made in this thesis, and section 1.8 gives an outline of the organization of the rest of this document.

1.1 Definitions

Before we further discuss the problem statement, requirements, and approach,this section first formalizes some of the terms used throughout this thesis.

- **Live Debugging** For the purpose of this thesis, we define *live debugging* as a mechanism to debug applications on the fly while the production services are running and serving end-users.
- The **development environment** refers to a setting (physical location, group of human developers, development tools, and production and test facilities) in which software is created and tested by software developers and is not made available to end users. The debugging process in the development environment can be interactive, and can have a high overhead.
- A **production environment**, or use environment, refers to a setting in which software is no longer being modified by software developers and is being actively being used by users. Applications in production cannot have a high instrumentation/debugging overhead, as it is detrimental to the users.
- An **error**, also referred to as a defect or bug, is the deviation of system external state from correct service state.
- A **fault** is the adjudged or hypothesized cause of an error.

- A **failure** is an event that occurs when the delivered functionality deviates from correct functionality. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function.

- **DevOps** is a software development method that stresses communication, collaboration (information sharing and web service usage), integration, automation and measurement of cooperation between software developers and other information-technology (IT) professionals. DevOps acknowledges the interdependence of software development and IT operations. It aims to help an organization rapidly produce software products and services and to improve operations performance quality assurance.

- **Development/Operational Phase** Development phase is the phase where the application is being developed. The process involves testing, and debugging and iterative development such as adding bug fixes etc. Operational phase is where the application is being operated and used by active users

- **Downstream Servers** For a given application or service, the downstream server is the server which sends it a request.

- **Upstream Servers** For a given application or service, the upstream servers are servers which process it's requests and send it responses.

- **Production Container** This is the container in which the original production service is hosted and where all incoming requests are routed.

- **Debug Container** This is a replica of the production container, where a copy of the production service is running. The debug container is used for debugging purposes, and provides the *live debugging* service.

- **Replica** A replica is a clone of a container, with an exact clone of the file system and the processes running in the container. For the purpose of this thesis *debug container* and replica refer to the same thing.

- **Service Oriented Applications** Service oriented applications are applications which offer transactional services via network input, and provide responses on the network as well.

1.2 Problem Statement

Despite advances in software engineering bugs in applications are inevitable. The complexity of distributed and large scale applications, with an increased emphasis on shorter development cycles has made debugging more difficult. The key challenge of debugging modern applications is twofold: firstly, the complexity due to a combination of distributed components interacting together, and secondly fast debugging of applications to assure a short-time-to debug.

We have observed that while several debugging techniques exist, most of them focus on localizing errors in the development phase. Production level debugging techniques are ad-hoc in nature, and generally rely on unstructured logs printed as exceptions or transaction events using print outs from within the application. While such logs are good, and can often give contextual information to the developer or the operator, they are meant to provide an indication to only expected errors. Furthermore, they do not provide a systematic way to localize such bugs.

More systematic approaches such as record-and-replay systems offer a complete picture of the running production systems. These tools capture the exact state, and execution of the system, and allow for it to be faithfully replayed offline. This saves the debugger hours of effort in re-creating the bug, it's input and application state. However, in order to capture such detailed information, there is a high performance penalty on the production systems. This is often unacceptable in real-world scenarios, which is why such techniques have only found limited use.

We further observe that debugging is an iterative process. While systematic approaches can provide a complete picture, developer insight is paramount. The debugging process usually involves several iterations where the debugger uses clues present in error logs, system logs, execution traces etc. to understand and capture the source of the error. This process can have an impact on real-world applications, hence traditionally the debugging and the production phase are kept completely separate.

Production level dynamic program instrumentation tools [McDougall *et al.*, 2006; Prasad *et al.*, 2005; Park and Buch, 2004] enable application debugging, and live insights of the application. However, these are executed inline with the program execution, thereby incurring an overhead. The perturbations and overhead because of the instrumentation could restrict the tools from being used in production environments. Thus we require a solution which allows operators/developers to observe, instrument, test or fix service oriented applications in parallel with the production. The techniques

and mechanisms in this thesis will aim to provide a *live debugging* environment, which allows debuggers a free reign to debug, without impacting the user-facing application.

1.3 Requirements

Our solution should meet the following requirements.

1. **Real-Time Insights:** Observing application behavior as the bug presents itself will allow for a quick insight and shorter time to debug. Any solution should allow the debugger to capture system status as well as observe, whatever points he wishes in the execution flow.
2. **Sanity and Correctness**: If the debugging is to be done in a running application with real users, it should be done without impacting the outcome of the program. The framework must ensure that any changes to the application's state or to the environment does not impact the user-facing production application.
3. **Language/Application Agnostic**: The mechanisms presented should be applicable to any language, and any service oriented application (our scope is limited to SOA architectures).
4. **Have negligible performance impact** The user of a system that is conducting tests on itself during execution should not observe any noticeable performance degradation. The tests must be unobtrusive to the end user, both in terms of functionality and any configuration or setup, in addition to performance.
5. **No service interruption**: Since we are focusing our efforts on service oriented systems, any solution should ensure that there is not impact on the service, and the user facing service should not be interrupted.

1.4 Scope

Although we present a solution that is designed to be general purpose and applicable to a variety of applications, in this thesis we specifically limit our scope to the following:

1.4.1 Service Oriented Applications

The traditional batch-processing single node applications are fast disappearing. Modern day devices like computers, IOT's, mobile's and web-browsers rely on interactive and responsive applications, which provide a rich interface to it's end-users. Behind the scenes of these applications are several *SOA* applications working in concert to provide the final service. Such services include storage, compute, queuing, synchronization, application layer services. One common aspect of all of these services is the fact that they get input from network sources. Multiple services can be hosted on multiple machines(many-to-many deployment), and each of them communicates with the other as well as the user using the network. The work presented in this thesis leverages duplication of network based input to generate a parallel debugging environment. In this sense, the scope of the applications targeted in this thesis are limited to service oriented applications, which gather input through the network.

1.4.2 Non-Crashing Bugs

In this thesis, we have primarily focused on continuous debugging in parallel with the production application. We have looked at a variety of bugs - performance, resource leak, concurrency, semantic, configuration etc. However, we also try to debug an active problem in the application.

Hence, although a bug which immediately crashes, can still be investigated using *Parikshan*, it would not be an ideal use-case scenario. On the other hand non-crashing bugs such as performance slow-downs, resource leaks which stay in the application long enough, fault tolerant bugs, which do not crash the entire system or similar non-crashing concurrency, semantic and configuration bugs, can be investigated in parallel to the original applications thereby reducing the investigation time, and the time to fix the bug.

1.4.3 Native Applications

One of the tools presented in this thesis is `iProbe`- an intelligent hybrid instrumentation tool. `iProbe` uses place-holders inserted at compile time in the binary, and leverages them to dynamically patch them at the run-time. In it's current implementation `iProbe`'s techniques can be only applied on native applications.

Managed and run-time interpreted languages such as Java, and .NET can also theoretically have a similar approach built in, but that is out of the scope of this thesis.

1.5 Proposed Approach

Analyzing the executions of a buggy software program is essentially a data mining process. Although several interesting methods have been developed to trace crashing bugs (such as memory violations and core dumps), it is still difficult to analyze non-crashing bugs. Studies have shown that several bugs in large-scale systems lead to either a changed/inconsistent output, or impact the performance of the application. Examples of this are slow memory leaks, configuration, or performance bugs, which do not necessarily stop all services, but need to be fixed quickly so as to avoid degradation in the QoS.

Existing approaches towards debugging production bugs mostly rely on application logs, and transaction logs which are inserted within the application by the developer himself, to give an idea of the progress of the application, and to guide the debugger towards errors. While these logs provide valuable contextual information, they can only be used for expected bug scenarios. Furthermore, often they provide incomplete information, or are just triggered as exceptions without providing a complete trace. Modern applications also contain a level of fault tolerance, which means that applications are likely to continue to spawn worker threads and provide service despite faults which happen at run-time. This often means that the debugger loses the context of the application.

Other more systematic debugging techniques have been used in record-and-replay techniques which allow operators to faithfully capture the entire execution as well as the status of the operating system as well as the application. This allows the debuggers to carefully debug the application offline and understand the root-cause of the bug. However, an obvious disadvantage of such techniques is that the recording overhead can be relatively high, especially in unpredictable worst-case scenarios (for e.g. spikes in user requests etc.). This makes the use of such techniques impractical for most real-world production systems.

Researchers have also studied non-systematic inference based techniques, which allow for lightweight tracing or capturing application logs in distributed applications, and then threading them together to form distributed execution flows. These inference techniques do not add much overhead to

the production system, as they typically use production instrumentation tools, or existing application logs. However, owing to the low amount of instrumentation and data captured, it is often not possible to recreate the entire trace. This means that the root-cause of the error may be lost.

We propose a **paradigm shift in debugging service oriented applications, with a focus on debugging applications running in the production environment**. We call this technique **"live debugging"**: this technique will provide real-time insights into running systems, and allow developers to debug applications without fearing crashes in the production application. We believe that this will in turn lead to much shorter time to bug resolution, hence improving application reliability, and reducing financial costs in case of errors. In this thesis we present an **end-to-end work-flow of localizing production bugs, which includes a framework for live debugging, new live debugging techniques, and mechanisms to make applications live debugging friendly**.

1.6 Hypothesis

The principal hypothesis we test in this thesis is as follows:

For user-facing application where time-to-bug resolution is critical, network replay alone is enough to trigger most bugs in service-oriented applications, and on-the-fly debugging parallel to the production application can be enabled without impacting the production environment. This can be achieved in a robust manner, with no impact on the production system, the debugging instrumentation can be adaptive and adhere to pre-defined budgets, and the production applications can be made debugging friendly to ensure low-cost instrumentation.

In order to test this, we have developed the following technologies:

1. Framework for debugging in production environment (Parikshan)
2. Packaging applications to allow low-cost instrumentation (iProbe)
3. Applications of live on-the-fly debugging

1.7 Assumptions

The work presented in this thesis is designed so that it can be applied in the most generic cases. However, the implementation and some of the design motivation make some key assumptions which are presented in this section:

1.7.1 Resource Availability

One of the core insight driving our live debugging technology is the increasing availability of compute resources. With more and more applications being deployed on cloud infrastructure, in order to ease scaling out of resources and sharing of compute power across multiple services - The amount of computing power available is flexible and plentiful. Several existing services like Amazon EC2 [Amazon, 2010] and Google Compute [gco,] provide infrastructure-as-a-service and form the backbone of several well known cloud services.

Parikshan assumes cheap and ample resource availability for most modern day services, and ease of scalability. We leverage this abundance of resources, to utilize unused resources for debugging purposes. As mentioned earlier, *Parikshan* uses unused containers to run a replica of the original production service, solely for the purpose of debugging. While it is difficult to quantify, we believe that the advantage of on-the-fly debugging and quick bug isolation outweighs the cost of these extra resources.

1.8 Outline

The rest of this thesis is organized as follows:

- Chapter 3 discusses the design and implementation of the *Parikshan* framework which enables live debugging. In this chapter we will first give a brief motivation, and discuss the overall design, and how our framework fits into service-oriented applications. We then go into a detailed explanation of the design of each of the components of network request duplication as well as our live cloning algorithm. We follow this up with implementation details, and evaluation scenarios using both simulation results and real-world experiments which show the performance of our framework. The chapter also includes a case study of 16 real-world

bugs, and a survey of 150+ bugs real-world bugs which we use to show that network replay is enough for triggering most bugs in service-oriented applications.

- Chapter 5 introduces `iProbe` a novel hybrid instrumentation technique, which can help in pre-packaging SOA application binaries for instrumentation - thereby helping in live-debugging. We first begin with an explanation of `iProbe`'s design, which is split in a two phase process - ColdPatching and HotPatching. This is explained in stateful diagrams to show how the code is modified at different states in the binary. We then show safety considerations of `iProbe` and this is followed by an extended design which shows how `iProbe` can be applied to applications without compile time modifications as well. Next we compare `iProbe`'s approach with traditional trampoline executions. We then follow this with the implementation, and a short description of *fperf* which is a application of `iProbe` for hardware monitoring. We follow this up with evaluation of `iProbe` which shows `iProbe`'s overhead in coldpatching and hotpatching phase, and it's comparison with traditional tools.

- While the previous two chapters build the base for *live debugging*, Chapter 6 discusses how these tools can be leveraged to do real-world debugging. In the first part of this chapter, we discuss several important advantages and limitations, which must be kept in mind when using *Parikshan* to debug applications. Then we discusss existing debugging techniques which can be used in tandem with *live debugging* to provide a more effective means for localizing the bug. In this chapter we introduce two key techniques which can be used for effective debugging with *Parikshan*. In the first called budget limited adaptive debugging we introduce how statistical debugging can be used in *Parikshan* to increase or decrease the degree of instrumentation in order to improve the statistical odds of localizing the bug. Next we look at *active debugging* which allows debuggers to apply fixes, modify running binaries and verify the bug fix on a parallel execution of an unmodified application.

- In chapter 8, we conclude this thesis, highlighting the contributions of our techniques. Additionally, this chapter also includes several future work possibilities that can arise from this thesis including some short-term future work and long-term possibilities.

- Appendix

Chapter 2

Background and Motivation

2.1 Recent Trends

Parikshan is driven by some recent trends in the industry towards faster bug resolution and quicker development, and scaled deployment. In this section we discuss three such trends in the industry which are of particular relevance to *Parikshan*.

2.1.1 Software development trends

Software development paradigms have evolved over the years from a more documentation oriented process to quicker and faster releases. The software development industry is working towards faster evolving softwares, rather than building monolithic softwares for long term uses. Similarly software development no longer follows strict regimented roles of developer, administrator/operator, tester etc, instead new paradigms are being developed which encourage cross-functionalities.

One recent trend in software development processes is *agile* [Martin, 2003] and *extreme* [Beck, 2000] programming development paradigms. Compared to traditional *waterfall model* [Petersen *et al.*, 2009], both *agile* and *extreme* programming focus on faster response to changing customer demands, and a quicker delivery time. Agile programming for instance works on the principle of very short development cycles called -*scrums*. At the end of each scrum, there should be a working software product that can be readily deployed. The work-items are generally short, and goal oriented, and a scrum will usually last at most 2 weeks.

Agile development focuses on shorter development cycle, to apply patches, bug-fixes and having

a leaner team/operations. *Parikshan*'s live-debugging capability is yet another tool to facilitate faster software development and debugging, by allowing developers to debug their applications in parallel to the one deployed in production. We believe agile development can be tied up with *Parikshan* to have an end-to-end quick test, debug, and deploy strategy and make application development an even more lean process.

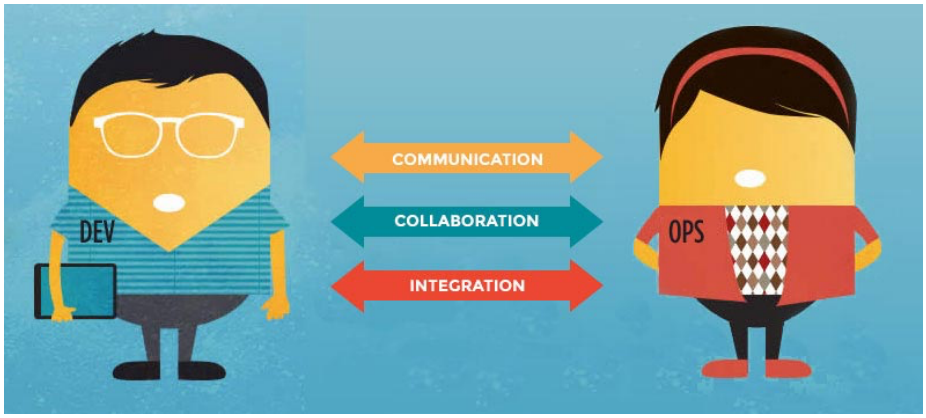


Figure 2.1: Devops is a new programming paradigm

Another trend in software development is cross-functional development and production application management called *Devops* [Allspaw J., 2009]. *Devops* is a term used to refer to a set of practices that emphasizes the collaboration and communication of both software developers and other information-technology (IT) professionals (operators/administrators) while automating the process of software delivery and infrastructure changes. The key in devops is the close collaboration of developers and operators, and an interchangable role (i.e. developers are also operators for real-time critical systems), or alternatively having developers and operators being active in the entire software cycle (including QA and operations). The old view of operations tended towards the Dev side being the makers and the Ops side being the people that deal with the creation after its birth the realization of the harm that has been done in the industry of those two being treated as siloed concerns is the core driver behind DevOps.

The driving force behind this change, where expensive resources(developers), are applied on what is traditionally managed by operators(with lower expertise or understanding of the software) - is to have faster responses and a shorter time to debug. This necessity of having a shorter time to

debug, and the availability of developers in the operation stage is one of the trend which motivates live debugging. Clearly developers who have much better understanding of the source code (having written it themselves), will be able to debug the application faster as long as they have some degree of visibility and debug-capability within the application. We believe that *Parikshan*'s livedebugging framework will allow such developers to debug their application in an isolated yet parallel environment, which clones in real-time the behavior without impacting the production. This will greatly reduce development overhead by giving crucial insight and make the feedback cycle shorter. *This will shorten the time to debug, and will easily fit into a debugging paradigm in an already increasing trend of devops.*.

2.1.2 Microservice Architecture

As applications grow in size they grow more and more complex with several interacting modules. With iterative improvements in every release applications tend to grow in code-size with large obsolete code-bases, un-productive technology, and which is difficult to maintain or modify owing to it's size and complexity. Many organizations, such as Amazon, eBay, and Netflix, have solved this problem by adopting what is now known as the Microservices Architecture pattern. Instead of building a single monstrous, monolithic application, the idea is to split your application into set of smaller, interconnected services.

A service typically implements a set of distinct features or functionality, such as order management, customer management, etc. Each microservice is a mini-application that has its own hexagonal architecture consisting of business logic along with various adapters. Some microservices would expose an API thats consumed by other microservices or by the applications clients. Other microservices might implement a web UI. At runtime, each instance is often a cloud VM or a Docker container.

Figure 2.2 shows the micro-service architecture of a car renting agency website. Each functional area is implemented as it's own independent service. Moreover, the web application is split into a set of simpler web applications (such as one for passengers and one for drivers in our taxi-hailing example). This makes it easier to deploy distinct experiences for specific users, devices, or specialized use cases.

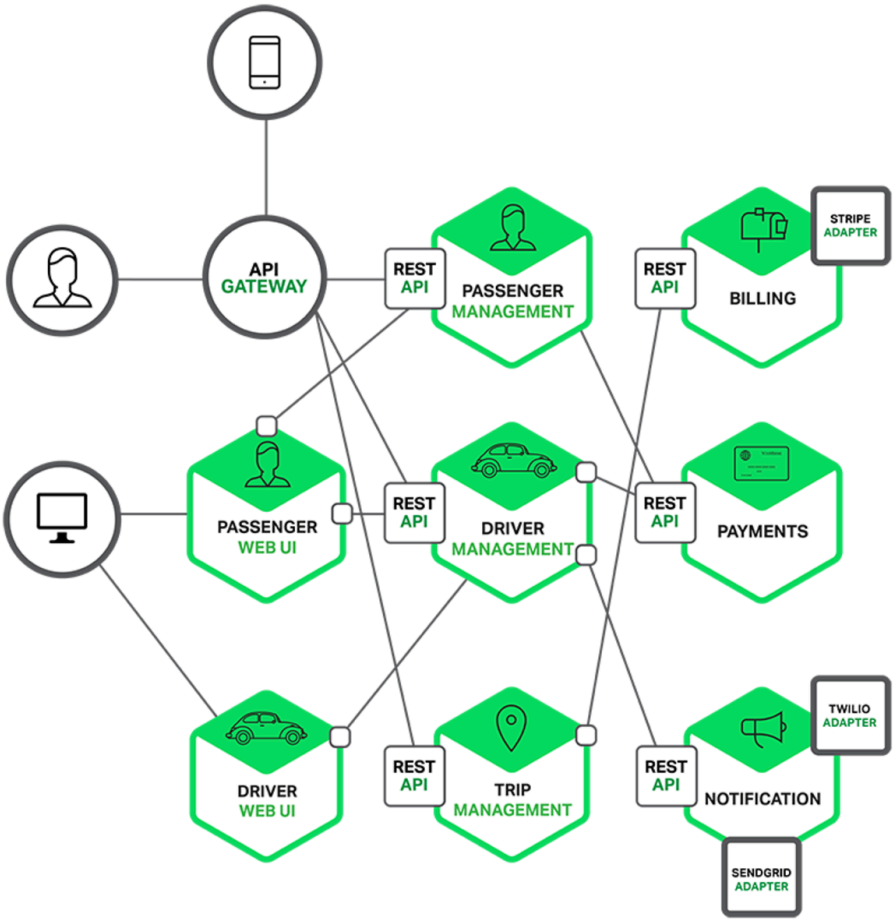


Figure 2.2: An example of a microservice architecture for a car renting agency website

2.1.3 Virtualization, Scalability and the Cloud

Modern day service oriented applications, are large and complex systems, which can serve billions of users. Facebook has 1.79 billion active users every month, and Google search has approximately 1.71 billion users, similarly twitter, netflix, instagram, and several other such websites have a huge base of users.

2.2 Current debugging of production systems

Before moving forward with a new software debugging paradigm, we want to discuss the current state-of-art debugging mechanisms followed in the industry. The software development cycle consists of the following four components - software development, monitoring, modeling & analytics, and software debugging.

Here monitoring involves getting periodic statistics or insight regarding the application, when deployed in the production environment, either using instrumentation within the application or using periodic sampling of resource usage in the system. Monitoring gives an indication regarding the general health of the system, and can alert the user incase anything has gone wrong. System level default tools provided by most commodity operating systems, like process monitors in linux, mac and windows, provide a high level view of real-time resource usage in the system. On the other hand, software event monitoring tools like nagios, ganglia, and rsyslog [Enterprises, 2012; Massie *et al.*, 2004; Matulis, 2009] aggregate logs and provide a consolidated view of application operations a cluster of machines to the administrator. On the other hand, tools like SystemTap [Prasad *et al.*, 2005], DTrace [McDougall *et al.*, 2006] allow operators to write customized instrumentation and dynamically patch them into applications to allow for a much deeper understanding of the system (albeit at higher overheads).

Modeling and analytics is generally a follow up step, which uses the output of monitoring and can provide useful insights using the monitoring data in real-time to highlight outliers and unexpected behavior. Tools like loggly [loggly,], ELK [ElasticSearch,], Splunk [splunk,], allow operators to search logs in real-time, as well as provide statistical analytics for different categories of logs. Academic tools like vpath [Tak *et al.*, 2009], magpie [Barham *et al.*, 2004], spectroscope [Sambasivan *et al.*, 2011], appinsight [Ravindranath *et al.*, 2012], amongst others can stitch events together to give a much more detailed transaction flow analysis.

As can be seen in figure 2.3, both monitoring and analytics happen in real-time in parallel to production applications. However, without any interaction with the running application these techniques are only limited to realizing that the production system has a bug, and potentially localizing the error. The actual root-cause extraction unfortunately currently relies on offline debugging. *Parikshan* aims to move the debugging process from an offline process to a completely or partially online (real-time) process in order to shorten time to debugging. In some cases our framework

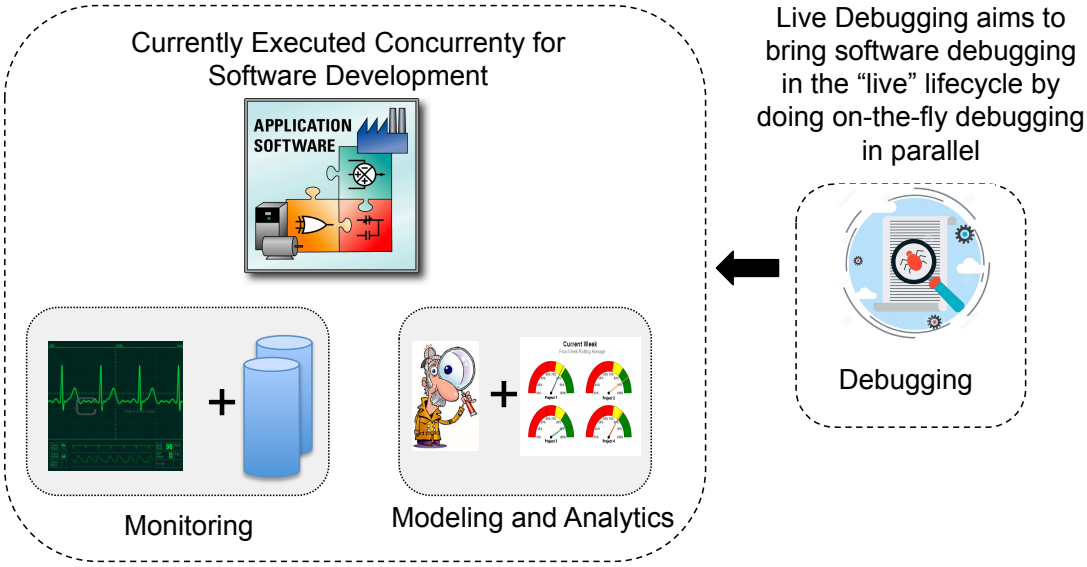


Figure 2.3: Live Debugging aims to move debugging part of the lifecycle to be done in parallel to the running application, as currently modeling, analytics, and monitoring is done

can also be used for patch testing and fix validation. In the next section we will see a real-world motivation scenario for *Parikshan*.

2.3 Motivating Scenario

Consider the complex multi-tier service-oriented system shown in Figure 2.4 that contains several interacting services (web servers, application servers, search and indexing, database, etc.). The system is maintained by operators who can observe the health of the system using lightweight monitoring that is attached to the deployed system. At some point, an unusual memory usage is observed in the glassfish application server, and some error logs are generated in the Nginx web server. Administrators can then surmise that there is a potential memory leak/allocation problem in the app-server or a problem in the web server. However, with a limited amount of monitoring information, they can only go so far.

Typically, trouble tickets are generated for such problems, and they are debugged offline. However using *Parikshan*, administrators can generate replicas of the Nginx and Glassfish containers as ***Nginx-debug*** and ***glassfish-debug***. *Parikshan*'s network duplication mechanism ensures that the debug

replicas receive the same inputs as the production containers and that the production containers continue to provide service without interruption. This separation of the production and debug environment allows the operator to use dynamic instrumentation tools to perform deeper diagnosis without fear of additional disruptions due to debugging. Since the replica is cloned from the original potentially "buggy" *production container*, it will also exhibit the same memory leaks/or logical errors. Additionally, *Parikshan* can focus on the "buggy" parts of the system, without needing to replicate the entire system in a test-cluster. This process will greatly reduce the time to bug resolution, and allow real-time bug diagnosis capability.

The replica can be created at any time: either from the start of execution, or at any point during execution that an operator deems necessary, allowing for post-facto analysis of the error, by observing execution traces of incoming requests (in the case of performance bugs and memory leaks, these will be persistent in the running system). Within the debug replica, the developer is free to employ any dynamic analysis tools to study the buggy execution, as long as the only side-effect those tools is on execution speed.

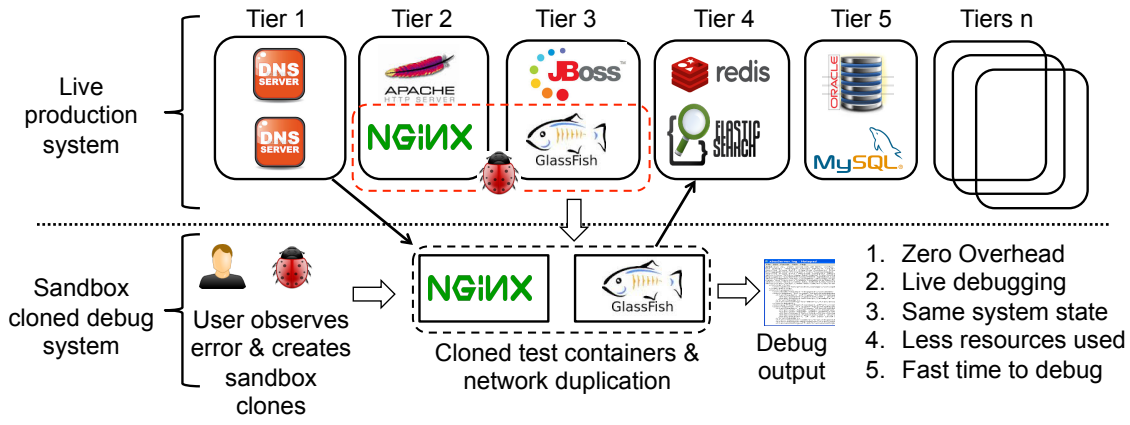


Figure 2.4: Workflow of *Parikshan* in a live multi-tier production system with several interacting services. When the administrator of the system observes errors in two of it's tiers, he can create a sandboxed clone of these tiers and observe/debug them in a sandbox environment without impacting the production system.

2.4 Summary

In this chapter we first discussed some recent software trends which motivated the development of *Parikshan*, and show that it complements as well as is driven by the current direction of industry. We then discussed the current state-of-art practices followed in the industry for most production applications, and showed the current limitation in doing real-time debugging. We then discussed a motivation scenario highlighting a real-world use-case for *Parikshan*, and how livedebugging could hypothetically take place.

Part I

Parikshan: A framework for sandboxed, online debugging of production bugs with no overhead

Chapter 3

Parikshan

3.1 Introduction

Rapid resolution of incident (error/alert) management [Lou *et al.*, 2013] in online service-oriented systems [Newman, 2015; Borthakur, 2008; Lakshman and Malik, 2010; Carlson, 2013] is extremely important. The large scale of such systems means that any downtime has significant financial penalties for all parties involved. However, the complexities of virtualized environments coupled with large distributed systems have made bug localization extremely difficult. Debugging such production systems requires careful re-creation of a similar environment and workload, so that developers can reproduce and identify the cause of the problem.

Existing state-of-art techniques for monitoring production systems rely on execution trace information. These traces can be replayed in a developer's environment, allowing them to use dynamic instrumentation and debugging tools to understand the fault that occurred in production. On one extreme, these monitoring systems may capture only very minimal, high level information, for instance, collecting existing log information and building a model of the system and its irregularities from it [Barham *et al.*, 2004; Erlingsson *et al.*, 2012; Kasikci *et al.*, 2015; Eigler and Hat, 2006]. While these systems impose almost no overhead on the production system being debugged (since they simply collect log information already being collected, or have light-weight monitoring), they are limited in their fault finding and redproduction power, hence limited in their utility to developers. On the other extreme, some monitoring systems capture complete execution traces, allowing the entire application execution to be exactly reproduced in a debugging environment [Altekar and Stoica, 2009;

Dunlap *et al.*, 2002; Laadan *et al.*, 2010; Geels *et al.*, 2007]. Despite much work towards minimizing the amount of such trace data captured, overheads imposed by such tracing can still be unacceptable for production use: in most cases, the overhead of tracing is at least 10%, and it can balloon up to 2-10x overhead. [Patil *et al.*, 2010; Wang *et al.*, 2014].

We seek to allow developers to diagnose and resolve crashing and non-crashing failures of production service-oriented systems *without suffering any performance overhead*. Our key insight is that for most service-oriented systems, a failure can be reproduced simply by replaying the network inputs passed to the application. For these failures, capturing very low-level sources of non-determinism (e.g. thread scheduling or general system calls, often with very high overhead) is unnecessary to successfully and automatically reproduce the buggy execution in a development environment. We evaluated this insight by studying 16 real-world bugs (see Section 4.2), which we were able to trigger by only duplicating and replaying network packets. Furthermore, we categorized 217 bugs from three real world applications, finding that most of these were similar in nature to the 16 that we reproduced. This suggests that our approach would be applicable to the bugs in our survey as well (see Section 3.3.3).

Guided by this insight, we have created *Parikshan*[1], which allows for real-time, online debugging of production services ***without imposing any performance penalty***. At a high level, *Parikshan* leverages live cloning technology to create a sandboxed replica environment. This replica is kept isolated from the real world so that developers can modify the running system in the sandbox to support their debugging efforts without fear of impacting the production system. Once the replica is executing, *Parikshan* replicates all network inputs flowing to the production system, buffering and feeding them (without blocking the production system) to the debug system. Within that debug system, developers are free to use heavy-weight instrumentation that would not be suitable in a production environment to diagnose the fault. Meanwhile, the production system can continue to service other requests. *Parikshan* can be seen as very similar to tools such as Aftersight [Chow *et al.*, 2008] that offload dynamic analysis tasks to replicas and VARAN [Hosek and Cadar, 2015] that support multi-version execution, but differs in that its high-level recording level (network inputs, rather than system calls) allows it to have significantly lower overhead.

Parikshan focuses on helping developers debug faults *online* — as they occur in production

[1] *Parikshan* is the *sanskrit* word for testing

systems. We expect *Parikshan* to be used in cases of tricky bugs that are highly sensitive to their environment, such as semantic bugs, performance bugs, resource-leak errors, configuration bugs, and concurrency bugs. Although in principle, *Parikshan* can be used to diagnose crashing bugs, we target primarily non-crashing bugs, where it is important for the production system to remain running even after a bug is triggered, for instance, to continue to process other requests. We present a more detailed explanation of these categories in Section 4.2.

We leverage container virtualization technology (e.g., Docker [Merkel, 2014], OpenVZ [Kolyshkin, 2006]), which can be used to pre-package services so as to make deployment of complex multi-tier systems easier (i.e. DockerHub [DockerHub, ; Boettiger, 2015] provides pre-packaged containers for storage, webserver, database services etc.). Container based virtualization is now increasingly being used in practice [Bernstein, 2014]. In contrast to VM's containers run natively on the physical host (i.e. there is no hypervisor layer in between), this means that there is no additional overhead, and near-native performance for containers [Felter *et al.*, 2015; Xavier *et al.*, 2013]. While *Parikshan* could also be deployed using VM's, container virtualization is much more light weight in terms of resource usage.

The key benefits of our system are:

- **Zero Overhead Monitoring:** While existing approaches have focused on minimizing the recording overhead. *Parikshan* uses novel non-blocking network duplication to avoid any overhead at all in the production environment.
- **Sandbox debugging:** *Parikshan* provides a cloned sandbox environment to debug the production application. This allows a safe mechanism to diagnose the error, without impacting the functionality of the application.
- **Capture large-scale context:** Allows capturing the context of large scale production systems, with long running applications. Under normal circumstances capturing such states is extremely difficult as they need a long running test input and large test-clusters.

The rest of the paper is organized as follows. Section 3.2 and 3.2.5 describe the design and implementation of *Parikshan* and each of it's internal components. We then present a case study of 16 real-world bugs successfully reproduced by *Parikshan* in Section 4.2. This is followed by the evaluation in Section 3.3. In Section 3.4, we discuss potential applications of *Parikshan*. Finally, we discuss some challenges in Section 3.5 and conclude.

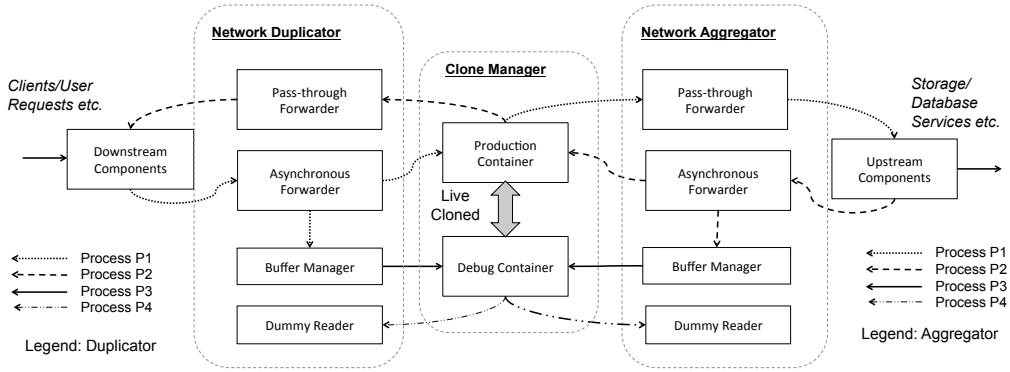


Figure 3.1: **High level architecture of *Parikshan***, showing the main components: Network Duplicator, Network Aggregator, and Cloning Manager. The replica (debug container) is kept in sync with the master (production container) through network-level record and replay. In our evaluation, we found that this light-weight procedure was sufficient to reproduce many real bugs.

3.2 *Parikshan*

In Figure 3.1, we show the architecture of *Parikshan* when applied to a single mid-tier application server. *Parikshan* consists of 3 modules: **Clone Manager**: manages "live cloning" between the production containers and the debug replicas, **Network Duplicator**: manages network traffic duplication from downstream servers to both the production and debug containers, and **Network Aggregator**: manages network communication from the production and debug containers to upstream servers. The network duplicator also performs the important task of ensuring that the production and debug container executions do not diverge. The duplicator and aggregator can be used to target multiple connected tiers of a system by duplicating traffic at the beginning and end of a workflow. Furthermore, the aggregator module is not required if the debug-container has no upstream services.

3.2.1 Clone Manager

Live migration [Mirkin *et al.*, 2008; Clark *et al.*, 2005; Gebhart and Bozak, 2009] refers to the process of moving a running virtual machine or container from one server to another, without disconnecting any client or process running within the machine (this usually incurs a short or negligible suspend time). In contrast to live migration where the original container is destroyed, the "Live Cloning" process used in *Parikshan* requires both containers to be actively running, and be still attached to

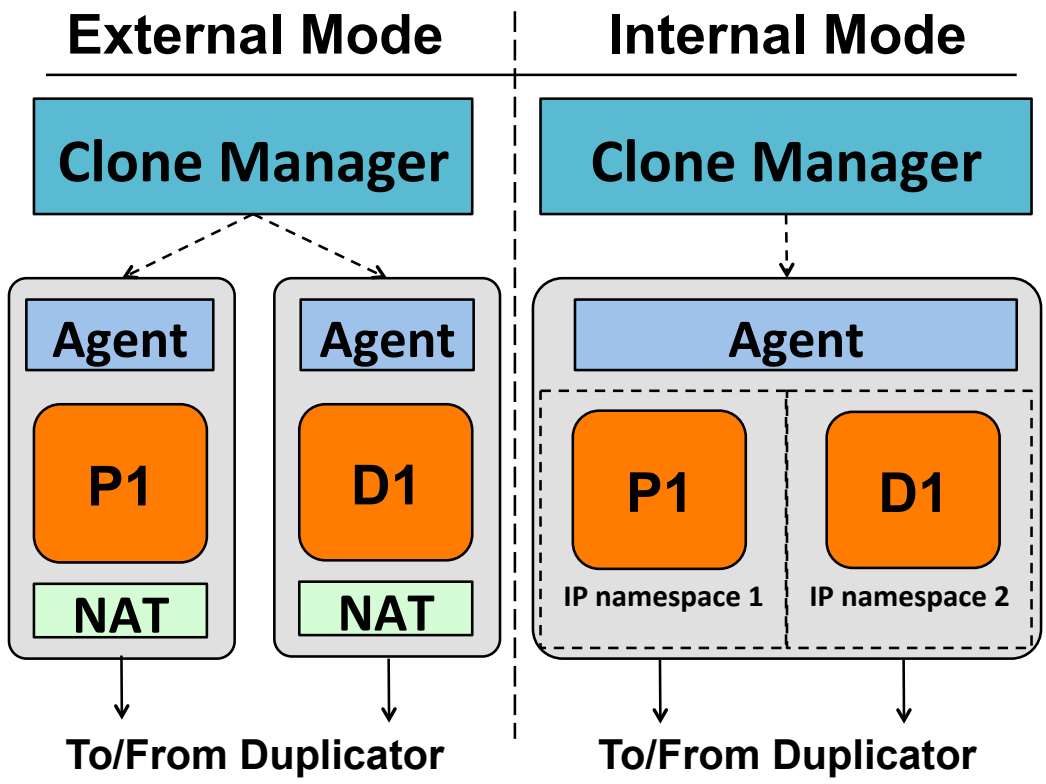


Figure 3.2: External and Internal Mode for live cloning: P1 is the production, and D1 is the debug container, the clone manager interacts with an agent which has drivers to implement live cloning.

the original network. The challenge here is to manage two containers with the same identities in the network and application domain. This is important, as the operating system and the application processes running in it may be configured with IP addresses, which cannot be changed on the fly. Hence, the same network identifier should map to two separate addresses, and enable communication with no problems or slowdowns.

We now describe two modes (see Figure 3.2) in which cloning has been applied, followed by the algorithm for live cloning:

- ***Internal Mode***: In this mode, we allocate the production and debug containers to the same host node. This would mean less suspend time, as the production container can be locally cloned (instead of streaming over the network). Additionally, it is more cost-effective since the number of servers remain the same. On the other hand, co-hosting the debug and production containers

could potentially have an adverse effect on the performance of the production container because of resource contention. Network identities in this mode are managed by encapsulating each container in separate network namespaces [net,]. This allows both containers to have the same IP address with different interfaces. The duplicator is then able to communicate to both these containers with no networking conflict.

- ***External Mode***: In this mode we provision an extra server as the host of our debug-container (this server can host more than one debug-container). While this mechanism can have a higher overhead in terms of suspend time (dependent on workload) and requires provisioning an extra host-node, the advantage of this mechanism is that once cloned, the debug-container is totally separate and will not impact the performance of the production-container. We believe that external mode will be more practical in comparison to internal mode, as cloning is likely to be transient, and high network bandwidth between physical hosts can offset the slowdown in cloning performance. Network identities in external mode are managed using NAT [nat,] (network address translator) in both host machines. Hence both containers can have the same address without any conflict.[2]

Algorithm 1 describes the specific process for cloning some production container P1 from Host H1 to replica D1 on Host H2.

The suspend time of cloning depends on the operations happening within the container between step 2 and step 4 (the first and the second rsync), as this will increase the number of dirty pages in the memory, which in turn will impact the amount of memory that needs to be copied during the suspend phase. This suspend time can be viewed as an amortized cost in lieu of instrumentation overhead. We evaluate the performance of live cloning in Section 3.3.1.

3.2.2 Network Proxy Design Description

The network proxy duplicator and aggregator are composed of the following internal components:

[2]Another additional mode can be *Scaled Mode*: This can be viewed as a variant of the external mode, where we can execute debug analysis in parallel on more than one debug-containers each having its own cloned connection. This will distribute the instrumentation load and allow us to do more analysis concurrently, without overflowing the buffer. We aim to explore this in the future.

Algorithm 1 Live cloning algorithm using OpenVZ

1. Safety checks and pre-processing (ssh-copy-id operation for password-less rsync, checking pre-existing container ID's, version number etc.)
2. Create and synchronize file system of P1 to D1
3. Set up port forwarding, duplicator, and aggregator
4. Suspend the production container P1
5. Checkpoint & dump the process state of P1
6. Since step 2 and 5 are non-atomic operations, some files may be outdated. A second sync is run when the container is suspended to ensure P1 and D1 have the same state
7. Resume both production and debug containers

- **Synchronous Passthrough**: The synchronous passthrough is a daemon that takes the input from a source port, and forwards it to a destination port. The passthrough is used for communication from the production container out to other components (which is not duplicated).
- **Asynchronous Forwarder**: The asynchronous forwarder is a daemon that takes the input from a source port, and forwards it to a destination port, and also to an internal buffer. The forwarding to the buffer is done in a non-blocking manner, so as to not block the network forwarding.
- **Buffer Manager**: Manages a FIFO queue for data kept internally in the proxy for the debug-container. It records the incoming data, and forwards it a destination port.
- **Dummy Reader**: This is a standalone daemon, which reads and drops packets from a source port

3.2.2.1 Proxy Network Duplicator:

To successfully perform online debugging in the replica to work, both production and debug containers must receive the same input. A major challenge in this process is that the production and debug container may execute at different speeds (debug will be slower than production): this will result in them being out of sync. Additionally, we need to accept responses from both servers and drop all the traffic coming from the debug-container, while still maintaining an active connection with the client. Hence simple port-mirroring and proxy mechanisms will not work for us.

TCP is a connection-oriented protocol and is designed for stateful delivery and acknowledgment that each packet has been delivered. Packet sending and receiving are blocking operations, and if either the sender or the receiver is faster than the other the send/receive operations are automatically blocked or throttled. This can be viewed as follows: Let us assume that the client was sending packets at $XMbps$ (link 1), and the production container was receiving/processing packets at $YMbps$ (link 2), where $Y < X$. Then automatically, the speed of link 1 and link 2 will be throttled to $YMbps$ per second, i.e the packet sending at the client will be throttled to accommodate the production server. Network throttling is a default TCP behavior to keep the sender and receiver synchronized. However, if we also send packets to the debug-container sequentially in link 3 the performance of the production container will be dependent on the debug-container. If the speed of link 3 is $Z\ Mbps$, where $Z < Y$, and $Z < X$, then the speed of link 1, and link 2 will also be throttled to $Z\ Mbps$. The speed of the debug container is likely to be slower than production: this may impact the performance of the production container.

Our solution is a customized TCP level proxy. This proxy duplicates network traffic to the debug container while maintaining the TCP session and state with the production container. Since it works at the TCP/IP layer, the applications are completely oblivious to it. To understand this better let us look at Figure 3.1: Here each incoming connection is forwarded to both the production container and the debug container . This is a multi-process job involving 4 parallel processes (P1-P4): In P1, the asynchronous forwarder sends data from client to the production service, while simultaneously sending it to the buffer manager in a non-blocking send. This ensures that there is no delay in the flow to the production container because of slow-down in the debug-container. In P2, the pass-through forwarder reads data from the production and sends it to the client (downstream component). Process P3, then sends data from Buffer Manager to the debug container, and Process P4 uses a dummy reader, to read from the production container and drops all the packets

The above strategy allows for non-blocking packet forwarding and enables a key feature of *Parikshan*, whereby it avoids slowdowns in the debug-container to impact the production container. We take the advantage of an in-memory buffer, which can hold requests for the debug-container, while the production container continues processing as normal. A side-effect of this strategy is that if the speed of the debug-container is too slow compared to the packet arrival rate in the buffer, it may eventually lead to an overflow. We call the time taken by a connection before which the buffer

overflows its *debug-window*. We discuss the implications of the *debug window* in Section 3.2.3.

3.2.2.2 Proxy Network Aggregator:

The proxy described in Section 3.2.2.1 is used to forward requests from downstream tiers to production and debug containers. While the network duplicator duplicates incoming requests, the network aggregator manages incoming "responses" for requests sent from the debug container. Imagine if you are trying to debug a mid-tier application container, the proxy network duplicator will replicate all incoming traffic from the client to both debug and the production container. Both the debug container and the production, will then try to communicate further to the backend containers. This means duplicate queries to backend servers (for instance, sending duplicate 'delete' messages to MySQL), thereby leading to an inconsistent state. Nevertheless, to have forward progress the debug-container must be able to communicate and get responses from upstream servers. The "proxy aggregator" module stubs the requests from a duplicate debug container by replaying the responses sent to the production container to the debug-container and dropping all packets sent from it to upstream servers.

As shown in Figure 3.1, when an incoming request comes to the aggregator, it first checks if the connection is from the production container or debug container. In process P1, the aggregator forwards the packets to the upstream component using the pass-through forwarder. In P2, the asynchronous forwarder sends the responses from the upstream component to the production container, and sends the response in a non-blocking manner to the internal queue in the buffer manager. Once again this ensures no slow-down in the responses sent to the production container. The buffer manager then forwards the responses to the debug container (Process P3). Finally, in process P4 a dummy reader reads all the responses from the debug container and discards them.

We assume that the production and the debug container are in the same state, and are sending the same requests. Hence, sending the corresponding responses from the FIFO queue instead of the backend ensures: (a) all communications to and from the debug container are isolated from the rest of the network, (b) the debug container gets a logical response for all it's outgoing requests, making forward progress possible, and (c). similar to the proxy duplicator, the communications from the proxy to internal buffer is non-blocking to ensure no overhead on the production-container.

3.2.3 Debug Window

Parikshan's asynchronous forwarder uses an internal buffer to ensure that incoming requests proceed directly to the production container without any latency, regardless of the speed at which the debug replica processes requests. The incoming request rate to the buffer is dependent on the user, and is limited by how fast the production container manages the requests (i.e. the production container is the rate-limiter). The outgoing rate from the buffer is dependent on how fast the debug-container processes the requests. Instrumentation overhead in the debug-container can potentially cause an increase in the transaction processing times in the debug-container. As the instrumentation overhead increases, the incoming rate of requests may eventually exceed the transaction processing rate in the debug container. If the debug container does not catch up, this in turn can lead to a buffer overflow. We call the time period until buffer overflow happens the *debug-window*. This depends on the size of the buffer, the incoming request rate, and the overhead induced in the debug-container. For the duration of the debugging-window, we assume that the debug-container faithfully represents the production container. Once the buffer has overflown, the debug-container may be out of sync with the production container. At this stage, the production container needs to be re-cloned, so that the replica is back in sync with the production and the buffer can be discarded. In case of frequent buffer-overflows, the buffer size needs to be increased or the instrumentation to be decreased in the replica, to allow for longer debug-windows.

The debug window size also depends on the application behavior, in particular how it launches TCP connections. *Parikshan* generates a pipe buffer for each TCP connect call, and the number of pipes are limited to the maximum number of connections allowed in the application. Hence, buffer overflows happen only if the requests being sent in the same connection overflow the queue. For webservers, and application servers, the debugging window size is generally not a problem, as each request is a new "connection." This enables *Parikshan* to tolerate significant instrumentation overhead without a buffer overflow. On the other hand, database and other session based services usually have small request sizes, but multiple requests can be sent in one session which is initiated by a user. In such cases, for a server receiving a heavy workload, the number of calls in a single session may eventually have a cumulative effect and cause overflows.

To further increase the *debug window*, we propose load balancing debugging instrumentation overhead across multiple debug-containers, which can each get a duplicate copy of the incoming

data. For instance, debug-container 1 could have 50% of the instrumentation, and the rest on debug-container 2. We believe such a strategy would significantly reduce the chance of a buffer overflow in cases where heavy instrumentation is needed. Section 3.3.2 explains in detail the behavior of the debug window, and how it is impacted by instrumentation.

3.2.4 Divergence Checking

In *Parikshan* it is possible that non-deterministic behavior (discussed in Section **??**) in the two containers or user instrumentation, causes the production and debug container to diverge with time. To understand and capture this divergence, we compare the corresponding network output received in the proxy. This is an optional component, which gives us a black-box mechanism to check the fidelity of the replica based on its communication with external components. In our current prototype, we use a hash on each data packet, which is collected and stored in memory for the duration that each packet's connection is active. The degree of acceptable divergence is dependent on the application behavior, and the operator's wishes. For example, an application that includes timestamps in each of its messages (i.e. is expected to have some non-determinism) could perhaps be expected to have a much higher degree of acceptable divergence than an application that should normally be returning deterministic results.

3.2.5 Implementation

The clone-manager and the live cloning utility are built on top of the user-space container virtualization software OpenVZ [Kolyshkin, 2006]. *Parikshan* extends *VZCTL* 4.8 [Furman, 2014] live migration facility [Mirkin *et al.*, 2008], to provide support for online cloning. To make **live cloning** easier and faster, we used OpenVZ's *ploop* devices [OpenVZ Ploop,] as the container disk layout. The network isolation for the production container was done using Linux network namespaces [net,] and NAT [nat,]. While *Parikshan* is based on light-weight containers, we believe that *Parikshan* can easily be applied to heavier-weight, traditional virtualization software where live migration has been further optimized [Svärd *et al.*, 2015; Deshpande and Keahey, 2016].

The network proxy duplicator and the network aggregator was implemented in C/C++. The forwarding in the proxy is done by forking off multiple processes each handling one send/or receive a connection in a loop from a source port to a destination port. Data from processes handling

communication with the production container, is transferred to those handling communication with the debug containers using *Linux Pipes* [lin,]. Pipe buffer size is a configurable input based on user-specifications.

3.3 Evaluation

To evaluate the performance of *Parikshan*, we pose and answer the following research questions:

- **RQ1:** How long does it take to create a live clone of a production container and what is it's impact on the performance of the production container?

- **RQ2:** What is the size of the debugging window, and how does it depend on resource constraints?

- **RQ3:** Can we generalize the results of our case study to see if *Parikshan* can target even more real bugs?

We evaluated the **internal mode** on two identical VM's with an Intel i7 CPU, with 4 Cores, and 16GB RAM each in the same physical host (one each for production and debug containers). We evaluated the **external mode** on two identical host nodes with Intel Core 2 Duo Processor, 8GB of RAM. All evaluations were performed on CentOS 6.5.

3.3.1 Live Cloning Performance

As explained in Section 5.2, a short suspend time during live cloning is necessary to ensure that both containers are in the exact same system state. The suspend time during live cloning can be divided in 4 parts: (1) Suspend & Dump: time taken to pause and dump the container, (2) Pcopy after suspend: time required to complete rsync operation (3) Copy Dump File: time taken to copy an initial dump file. (4) Undump & Resume: time taken to resume the containers. To evaluate "live cloning", we ran a micro-benchmark of I/O operations, and evaluated live-cloning on some real-world applications running real-workloads.

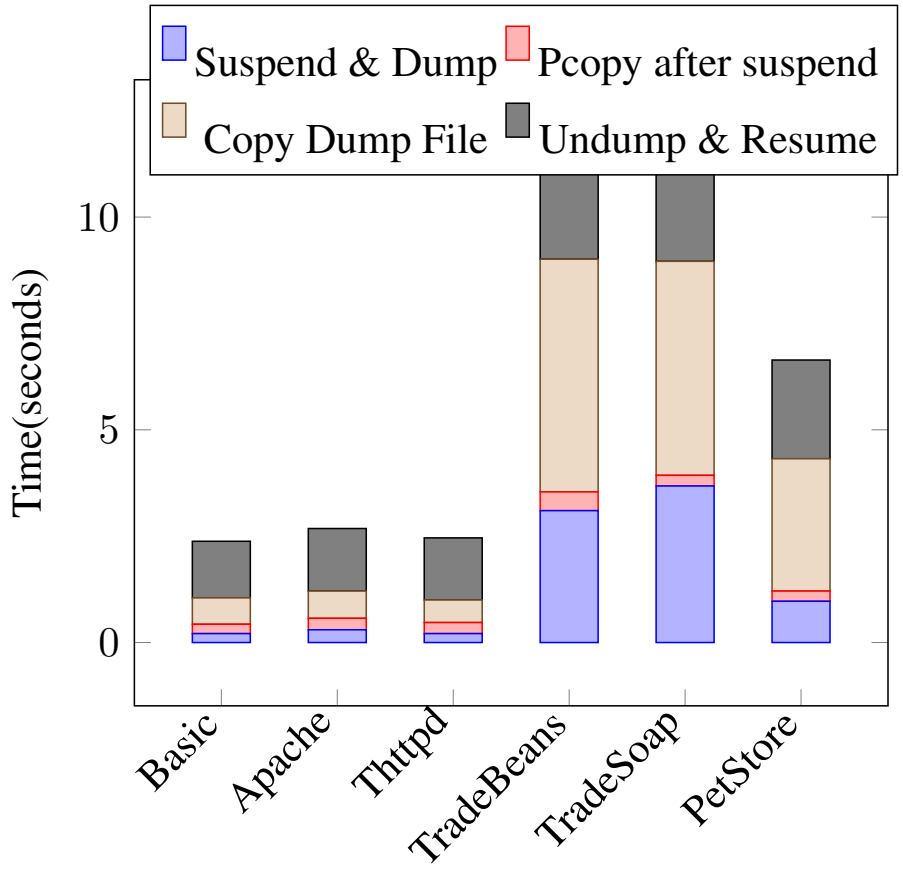


Figure 3.3: Suspend time for live cloning, when running a representative benchmark

3.3.1.1 Real world applications and workloads:

To begin to study the overhead of live cloning, we performed an evaluation using five well-known applications. Figure 3.3 presents the suspended times for five well-known applications when cloning a replica with *Parikshan*. We ran the httperf [Mosberger and Jin, 1998] benchmark on Apache and *thttpd* to compute max throughput of the web-servers, by sending a large number of concurrent requests. Tradebeans and Tradesoap are both part of the dacapo [Blackburn *et al.*, 2006] benchmark "DayTrader" application. These are realistic workloads, which run on a multi-tier trading application provided by IBM. PetStore [pet,] is also a well known J2EE reference application. We deployed PetStore in a 3-tier system with JBoss, MySQL and Apache servers, and cloned the app-server. The input workload was a random set of transactions which were repeated for the duration of the cloning process.

As shown in Figure 3.3, for Apache and Thttpd the container suspend time ranged between 2-3 seconds. However, in more memory intensive application servers such as PetStore and DayTrader, the total suspend time was higher (6-12 seconds). Nevertheless, we did not experience any timeouts or errors for the requests in the workload[3]. However, this did slowdown requests in the workload. This shows that short suspend times are largely not visible or have minimal performance impact to the user, as they are within the time out range of most applications. Further, a clean network migration process ensures that connections are not dropped, and are executed successfully. We felt that these relatively fast temporary app suspensions were a reasonable price to pay to launch an otherwise overhead-free debug replica. To further characterize the suspend time imposed by the live cloning phase of *Parikshan*, we created a synthetic micro-benchmark to push *Parikshan* towards its limit.

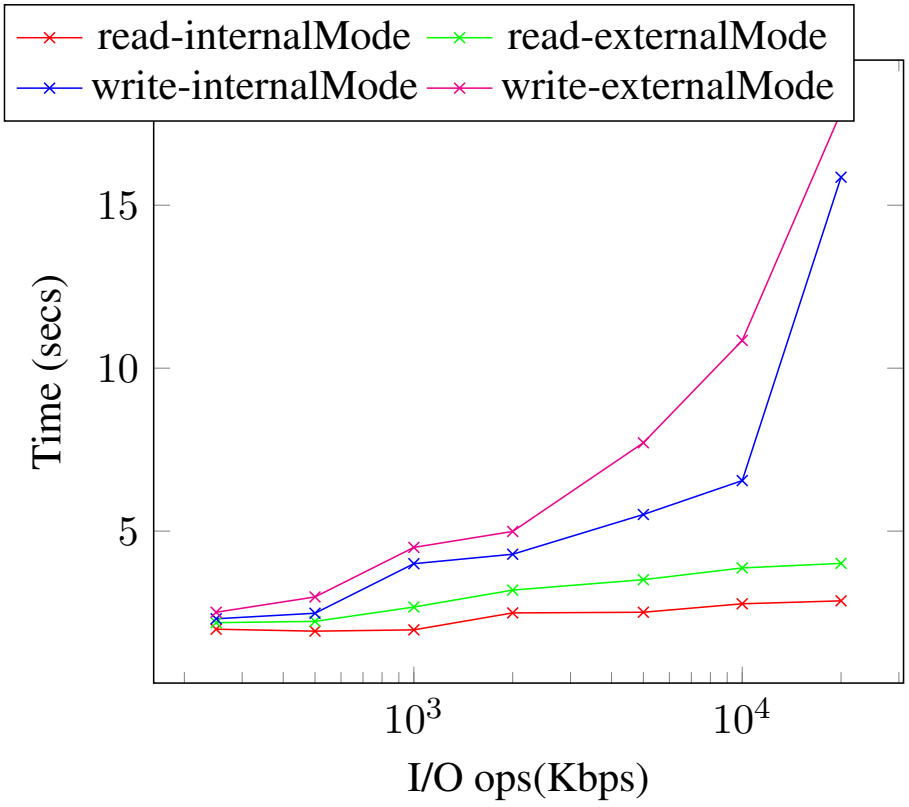


Figure 3.4: Live Cloning suspend time with increasing amounts of I/O operations

[3]In case of packet drops, requests are resent both at the TCP layer, and the application layer. This slows down the requests for the user, but does not drop them

3.3.1.2 Micro Benchmark using I/O operations:

The main factor that impacts suspend time is the number of "dirty pages" in the suspend phase, which have not been copied over in the pre-copy rsync operation (see section **??**). To understand this better, we use *fio* (flexible I/O tool for Linux) [Axboe, 2008], to gradually increase the number of I/O operations while doing live cloning. We run the *fio* tool to do read and writes of random values with a controlled I/O bandwidth. Additionally, we ensure that the I/O job being processed by *fio* is long enough to last through the cloning process.

As shown in figure 3.4, read operations have a much smaller impact on suspend time of live cloning compared to write operations. This can be attributed to the increase of "dirty pages" in write operations, whereas for read, the disk image remains largely the same. The internal mode is much faster than the external mode, as both the production and debug-container are hosted in the same physical device. We believe, that for higher I/O operations, with a large amount of "dirty-pages", network bandwidth becomes a bottleneck: leading to longer suspend times. Overall in our experiments, the internal mode is able to manage write operation up to 10 Mbps, with a total suspend-time of approx 5 seconds. Whereas, the external mode is only able to manage up to 5-6 Mbps, for a 5 sec suspend time.

To answer **RQ1**, live cloning introduces a short suspend time in the production container dependent on the workload. Write intensive workloads will lead to longer suspend times, while read intensive workloads will take much less. Suspend times in real workload on real-world systems vary from 2-3 seconds for webserver workloads to 10-11 seconds for application/database server workloads. Compared to external mode, internal mode had a shorter suspend time. A production-quality implementation could reduce suspend time further by rate-limiting incoming requests in the proxy, or using copy-on-write mechanisms and faster shared file system/storage devices already available in several existing live migration solutions.

3.3.2 Debug Window Size

To understand the size of the debug-window and it's dependence on resources, we did some experiments on real-world applications, by introducing a delay while duplicating the network input. This gave us some real-world idea of buffer overflow and it's relationship to the buffer size and input workload. Since it was difficult to observe systematic behavior in a live system to understand the decay rate of the debug-window, we also did some simulation experiments, to see how soon the buffer would overflow for different input criteria.

Input Rate	Debug Window	Pipe Size	Slowdown
530 bps, 27 rq/s	∞	4096	1.8x
530 bps, 27 rq/s	8 sec	4096	3x
530 bps, 27 rq/s	72 sec	16384	3x
Pois., λ = 17 rq/s	16 sec	4096	8x
Pois., λ = 17 rq/s	18 sec	4096	5x
Pois.,λ = 17 rq/s	∞	65536	3.2x
Pois.,λ = 17 rq/s	376 sec	16384	3.2x

Table 3.1: Approximate debug window sizes for a MySQL request workload

Experimental Results: We call the time taken to reach a buffer overflow the "debug-window". As explained earlier, the size of this debug-window depends on the overhead of the "instrumentation", the incoming workload distribution, and the size of the buffer. To evaluate the approximate size of the debug-window, we sent requests to both a production and debug MySQL container via our network duplicator. Each workload ran for about 7 minutes (10,000 "select * from table" queries), with varying request workloads. We also profiled the server, and found that is able to process a max of 27 req/s[4] in a single user connect session. For each of our experiments, we vary the buffer sizes to get an idea of debug-window. Additionally, we generated a slowdown by first modeling the time taken by MySQL to process requests (27 req/s or 17req/s), and then putting an approximate sleep in the request handler.

[4]Not the same as bandwidth, 27 req/s is the maximum rate of sequential requests MySQL server is able to handle for a user session

Initially, we created a connection and sent requests at the maximum request rate the server was able to handle (27 req/s). We found that for overheads up-to 1.8x (approx) we experienced no buffer overflows. For higher overheads the debug window rapidly decreased, primarily dependent on buffer-size, request size, and slowdown.

Next, we mimic user behavior, to generate a realistic workload. We send packets using a Poisson process with an average request rate of 17 requests per second to our proxy. This varies the inter-request arrival time, and let's the cloned debug-container catch up with the production container during idle time-periods in between request bursts. We observed, that compared to earlier experiments, there was more slack in the system. This meant that our system was able to tolerate a much higher overhead (3.2x) with no buffer overflows.

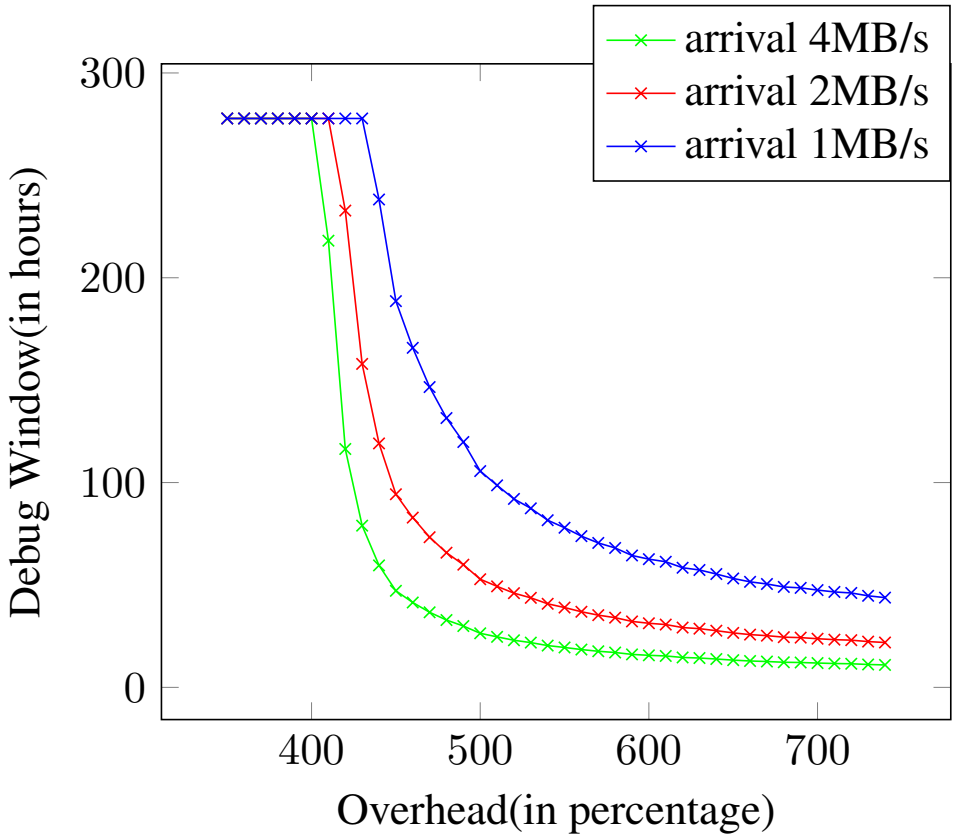


Figure 3.5: Simulation results for debug-window size. Each series has a constant arrival rate, and the buffer is kept at 64GB.

Simulation Results: In our next set of experiments, we simulate packet arrival and service process-

ing for a buffered queue in SOA applications. We use a discrete event simulation based on an MM1 queue, which is a classic queuing model based on Kendall's notation [Kendall, 1953], and is often used to model SOA applications with a single buffer based queue. Essentially, we are sending and processing requests based on a Poisson distribution with a finite buffer capacity. In our simulations (see Figure 3.5), we kept a constant buffer size of 64GB, and iteratively increased the overhead of instrumentation, thereby decreasing the service processing time. Each series (set of experiments), starts with an arrival rate approximately 5 times less than the service processing time. This means that at 400% overhead, the system would be running at full capacity (for stable systems SOA applications generally operate at much less than system capacity). Each simulation instance was run for 1000000 seconds or 277.7 hours. We gradually increased the instrumentation by 10% each time, and observed the *hitting-time* of the buffer (time it takes for the buffer to overflow for the first time). As shown there is no buffer overflow in any of the simulations until the overhead reaches around 420-470%, beyond this the debug-window decreases exponentially. Since beyond 400% overhead, the system is over-capacity, the queue will start filling up fairly quickly. This clarifies the behavior we observed in our experiments, where for lower overheads (1.8-3.2x) we did not observe any overflow, but beyond a certain point, we observed that the buffer would overflow fairly quickly. Also as shown in the system, since the buffer size is significantly larger than the packet arrival rate, it takes some time for the buffer to overflow (several hours). We believe that while most systems will run significantly under capacity, large buffer sizes can ensure that our debug-container may be able to handle short bursts in the workload. However, a system running continuously at capacity is unlikely to tolerate significant instrumentation overhead.

To answer **RQ2**, we found that the debug-container can stay in a stable state without any buffer overflows as long as the instrumentation does not cause the service times to become less than the request arrival rate. Furthermore, a large buffer will allow handling of short bursts in the workload until the system returns back to a stable state. The debug-window can allow for a significant slowdown, which means that many existing dynamic analysis techniques [Flanagan and Godefroid, 2005; Nethercote and Seward, 2007], as well as most

fine-grained tracing [Erlingsson *et al.*, 2012; Kasikci *et al.*, 2015] can be applied on the debug-container without leading to an incorrect state.

3.3.3 A survey of real-world bugs

In Table 3.2, we present the results of a survey of bug reports of three production SOA applications. In order to understand how we did the survey, let us look at MySQL as an example. We first searched for bugs which were tagged as "fixed" by developers and dumped them. We then chose a random time-line (2013-2014) and filtered out all bugs which belonged to non-production components - like documentation, installation failure, compilation failure. We then manually went through each of the bug-reports, filtering out the ones which were mislabeled or were reported based on code-analysis, or did not have a triggering test report (essentially we focused only on bugs that happened during production scenarios). We then classified these bugs into the categories shown in Table 3.2 based on the bug-report description, and the patch fix, to-do action item for the bug.

One of the core-insights provided by this survey was that most bugs (93%) triggered in production systems are deterministic in nature (everything but concurrency bugs), among which the most common ones are semantic bugs (80%). This is understandable, as they usually happen because of unexpected scenarios or edge cases, that were not thought of during testing. Recreation of these bugs depend only on the state of the machine, the running environment (other components connected when this bug was triggered), and network input requests, which trigger the bug scenario. *Parikshan* is a useful testing tool for testing these deterministic bugs in an exact clone of the production state, with replicated network input. The execution can then be traced at a much higher granularity than what would be allowed in production containers, to find the root cause of the bug.

On the other hand, concurrency errors, which are non-deterministic in nature make up for less than 7% of the production bugs. Owing to non-determinism, it is possible that the same execution is not triggered. However concurrent points can still be monitored and a post-facto search of different executions can be done to find the bug [Flanagan and Godefroid, 2005; Thomson and Donaldson, 2015] to capture these non-deterministic errors.

Category	Apache	MySQL	HDFS
Performance	3	10	6
Semantic	36	73	63
Concurrency	1	7	6
Resource Leak	5	6	1
Total	45	96	76

Table 3.2: Survey and classification of bugs

To answer **RQ3**, we found that almost 80% of bugs were semantic in nature, while less than 6% of the bugs are non-deterministic. About 13-14% of bugs are performance and resource-leak bugs, which are generally persistent in the system.

3.4 Applications of Live Debugging

Statistical Testing: One well-known technique for debugging production applications is statistical testing. This is achieved by having predicate profiles from both successful and failing runs of a program and applying statistical techniques to pinpoint the cause of the failure. The core advantage of statistical testing is that the sampling frequency of the instrumentation can be decreased to reduce the instrumentation overhead. However, the instrumentation frequency for such testing to be successful needs to be statistically significant. Unfortunately, overhead concerns in the production environment limit the frequency of instrumentation. In *Parikshan*, the buffer utilization can be used to control the frequency of such statistical instrumentation in the debug-container. This would allow the user to utilize the slack available in the debug-container for instrumentation to it's maximum, without leading to an overflow. Thereby improving the efficiency of statistical testing.

Record and Replay: Record and Replay techniques have been proposed to replay production site bugs. However, they are not yet used in practice as they can impose unacceptable overheads in the service processing time. *Parikshan* replicas can be used to do recording at a much finer granularity (higher overhead), allowing for easy and fast replays offline. Similar to existing mechanisms, the system can be replayed can then be used for offline debugging, without imposing any recording

overhead to the production container.

Patch Testing: Bug fixes and patches to resolve errors, often need to undergo testing in the offline environment and are not guaranteed to perform correctly. Patches can be made to the replica instead. The fix can be traced and observed if it is correctly working, before moving it to the production container. This is similar in nature to AB-Tesing, which is applied to find if a new fix is useful or works [Eisenberg and Quarto-vonTivadar, 2009]

3.5 Discussion and Limitations

Through our case studies and evaluation, we concluded that *Parikshan* can faithfully reproduce many real bugs in complex applications with no running-overhead. However, there may be several threats to the validity of our experiments. For instance, in our case study, the bugs that we selected to study may not be truly representative of a broad range of different faults. Perhaps, *Parikshan*'s low-overhead network record and replay approach is less suitable to some classes of bugs. To alleviate this concern, we selected bugs that represented a wide range of categories of bugs, and further, selected bugs that had already been studied in other literature, to alleviate a risk of selection bias. We further strengthened this studied with a follow-up categorization of 217 bugs in three real-world applications, finding that most of those bugs were semantic in nature, and very few were non-deterministic, and hence, having similar characteristics to those 16 that we reproduced.

There are also several underlying limitations and assumptions regarding *Parikshan*'s applicability:

3.5.1 Non-determinism:

Non-determinism can be attributed to three main sources (1) system configuration, (2) application input, and (3) ordering in concurrent threads. Live cloning of the application state ensures that both applications are in the same "system-state" and have the same configuration parameters for itself and all dependencies. *Parikshan*'s network proxy ensures that all inputs received in the production container are also forwarded to the debug container. However, any non-determinism from other sources (e.g. thread interleaving, random numbers, reliance on timing) may limit *Parikshan*'s ability to faithfully reproduce an execution. While our current prototype version does not handle these, we believe there are several existing techniques that can be applied to tackle this problem in the

context of live debugging. However, as can be seen in our case-studies above, unless there is significant non-determinism, the bugs will still be triggered in the replica, and can hence be debugged. Approaches like statistical debugging [Liblit, 2004], can be applied to localize bug. *Parikshan* allows debugger to do significant tracing of synchronization points, which is often required as an input for constraint solvers [Flanagan and Godefroid, 2005; Ganai *et al.*, 2011], which can go through all synchronization orderings to find concurrency errors. We have also tried to alleviate this problem using our divergence checker (Section **??**)

3.5.2 Distributed Services:

Large-scale distributed systems are often comprised of several interacting services such as storage, NTP, backup services, controllers and resource managers. *Parikshan* can be used on one or more containers and can be used to clone more than one communicating . Based on the nature of the service, it may be (a). Cloned, (b). Turned off or (c). Allowed without any modification. For example, storage services supporting a replica need to be cloned or turned off (depending on debugging environment) as they would propagate changes from the debug container to the production containers. Similarly, services such as NTP service can be allowed to continue without any cloning as they are publishsubscribe broadcast based systems and the debug container cannot impact it in anyway. Furthermore, instrumentation inserted in the replica, will not necessarily slowdown all services. For instance, instrumentation in a MySQL query handler will not slowdown file-sharing or NTP services running in the same container.

3.6 Summary

Parikshan is a novel framework that uses redundant cloud resources to debug production SOA applications in real-time. It can be combined with several existing bug diagnosis technique to localize errors. Compared to existing monitoring solutions, which have focused on reducing instrumentation overhead, our tool is able to avoid any performance slowdown at all, at the same time potentially allow significant monitoring for the debugger.

We have made *Parikshan* prototype tool available on GitHub for use by other researchers and practitioners. For each of the 16 faults studied in our case study, we have also created a docker

containers for most of our experiments, and in the rest we have left a detailed README of the install instructions, and the bug trigger mechanism.

Chapter 4

Is network replay enough?

4.1 Overview

Several existing record and replay have a much higher overhead as they record low level of non-determinism in order to capture and replay the exact state of execution. However, most bugs in service oriented application do not require such low level of recording. We hypothesize that network input with live cloning is enough to trigger most bugs in user-facing services.

To validate this insight, we selected sixteen real-world bugs, applied *Parikshan*, reproduced them in a production container, and observed whether they were also simultaneously reproduced in the replica. For each of the sixteen bugs that we triggered in the production environments, *Parikshan* faithfully reproduced them in the replica.

We selected our bugs from those examined in previous studies [Lu *et al.*, 2005; Yuan *et al.*, 2014], focusing on bugs that involved performance, resource-leaks, semantics, concurrency, and configuration. We have further categorized these bugs whether they lead to a crash or not, and if they can be deterministically reproduced. Table 4.1 presents an overview of our study.

In the rest of this section we discuss the bug-reproduction technique in each of these case-studies in further detail.

Bug Type	Bug ID	Application	Symptom/Cause	Determ-inistic	Crash	Trigger
Performance	MySQL #15811	mysql-5.0.15	Bug caused due to multiple calls in a loop	Yes	No	Repeated insert into table
	MySQL #26527	mysql-5.1.14	Load data is slow in a partitioned table	Yes	No	Create table with partition and load data
	MySQL #49491	mysql-5.1.38	calculation of hash values inefficient	Yes	No	MySql client select requests
Concurrency	Apache #25520	httpd-2.0.4	Per-child buffer management not thread safe	No	No	Continuous concurrent requests
	Apache #21287	httpd-2.0.48, php-4.4.1	Dangling pointer due to atomicity violation	No	Yes	Continuous concurrent request
	MySQL #644	mysql-4.1	data-race leading to crash	No	Yes	Concurrent select queries
	MySQL #169	mysql-3.23	Race condition leading to out-of-order logging	No	No	Delete and insert requests
	MySQL #791	mysql-4.0	Race - visible in logging	No	No	Concurrent flush log and insert requests
Semantic	Redis #487	redis-2.6.14	Keys* command duplicate or omits keys	Yes	No	Set keys to expire, execute specific reqs
	Cassandra #5225	cassandra-1.5.2	Missing columns from wide row	Yes	No	Fetch columns from cassandra
	Cassandra #1837	cassandra-0.7.0	Deleted columns become available after flush	Yes	No	Insert, delete, and flush columns
	Redis #761	redis-2.6.0	Crash with large integer input	Yes	Yes	Query for input of large integer
Resource Leak	Redis #614	redis-2.6.0	Master + slave, not replicated correctly	Yes	No	Setup replication, push and pop some elements
	Redis #417	redis-2.4.9	Memory leak in master	Yes	No	Concurrent key set requests
Configuration	Redis #957	redis-2.6.11	Slave cannot sync with master	Yes	No	Load a very large DB
	HDFS #1904	hdfs-0.23.0	Create a directory in wrong location	Yes	No	Create new directory

Table 4.1: List of real-world production bugs studied with *Parikshan*

4.2 Case Studies

4.2.1 Semantic Bugs

The majority of the bugs found in production SOA systems can be categorized as semantic bugs. These bugs often happen because an edge condition was not checked during the development stage or there was a logical error in the algorithm etc. Many such errors result in an unexpected output or possibly can crash the system. We recreated 4 real-world production bugs from Redis [Carlson, 2013] queuing system, and Cassandra [Lakshman and Malik, 2010] a NoSQL database.

4.2.1.1 Redis #761

In this subsection we describe the Redis#761 semantic bug

Cause of the error:

The Redis#761 is an integer overflow error. This error is triggered, when the client tries to insert and store a very large number. This leads to an unmanaged exception, which crashes the production system. Integer overflow, is a common error in several applications. We classify this as a semantic bug, which could have been fixed with an error checking condition for large integers.

Steps for reproduction:

1. Start a redis service with log level set to verbose. Setting loglevel to verbose ensures that we can view whatever is going on inside the service. This is our *production container*.
2. Create a live clone of the service mapped to a parallel *debug container* which will be used to visualize debugging
3. Start cloning the incoming traffic to both the *production container* and the *debug container* asynchronously using *Parikshan*'s network duplicator
4. Send the following request through the redis client:

   ```
   zinterstore out 9223372036854775807 zset zset2
   ```

   This tries to set the variable to the integer in the request, and leads to an integer overflow error. The integer overflow error is simultaneously triggered both in the production and the debug containers.

4.2.1.2 Redis #487

In this subsection we describe the Redis#487 semantic bug

Cause of the error:

Redis#487 resulted in expired keys still being retained in Redis, because of an unchecked edge condition. While this error does not lead to any exception or any error report in application logs, it gives the user a wrong output. In the case of such logical errors, the application keeps processing, but the internal state can stay incorrect. The bug impacts only clients who set keys, and then expire them.

Steps for reproduction:

1. Start a redis service with log level set to verbose. Setting loglevel to verbose ensures that we can view whatever is going on inside the service. This is our *production container*.
2. Create a live clone of the service mapped to a parallel *debug container* which will be used to visualize debugging
3. Start cloning the incoming traffic to both the *production container* and the *debug container* asynchronously using *Parikshan*'s network duplicator
4. flush all keys using `flushall` command
5. set multiple keys using `set key value` command
6. expire a key from one of them using `expire s 5` command
7. run the `keys *` command. This will list keys which should have expired

At the end it can be seen that the expired keys can be listed and accessed in both the production and debug container. This is a persistent error, which does not impact most other aspects of the service. The debug container can be used to look at the transactional logs, and have further instrumentation to understand the root cause of the error.

4.2.1.3 Cassandra #5225

In this subsection we describe the Cassandra#5225 semantic bug

Cause of the error:

Missing columns when requesting specific columns from a wide row. The data is still in the table, just it might not be returned to the user. Taking closer look, Cassandra is reading from the wrong col-

umn index. A problem was found with the index checking algorithm. In fact, it was written in reverse.

Steps for reproduction:

1. Start a cassandra service in the production container
2. Use *Parikshan*'s live cloning facility to create a clone of cassandra in the debug-container.
3. Connect to cassandra using a python client
4. Insert a large number of columns into cassandra (so that it is a wide row). For our testing we used `pycassa` python cassandra client.
5. Fetch the columns in a portion of ranges
6. At the end of this test case you can observe that some columns were dropped in the response to the client.

To facilitate re-creating this bug, we have provided dockerized containers, which install the correct version of Cassandra which has this bug as well as it's dependencies. The re-compilation of the version which has the bug is significantly difficult as some of the dependency libraries are no longer available using simply their Apache IVY build files. We also provide a python script for the cassandra client to trigger the bug conditions.

4.2.1.4 Cassandra #1837

In this subsection we describe the Cassandra#1837 semantic bug

Cause of the error:

The main symptom of this error was that deleted columns become available again after doing a flush. With some domain knowledge, a developer found the error. This happens because of a bug in a the way deleted rows are not interpreted once they leave the memtable in the CFS.getRangeSlice code i.e. the flush does not recognize the delete and the purged data does not contain the delete operation. Thus querying for the data shows content as well.

Steps for reproduction:

1. Start a cassandra service in the production container
2. Use *Parikshan*'s live cloning facility to create a clone of cassandra in the debug-container.
3. Using cassandra's command line client, insert columns into cassandra without flushing
4. delete the inserted column
5. flush the columns so that the deletion should be committed
6. query for the columns in the table
7. observe that the columns have not been deleted and are retained.

Once again we have provide dockerized containers for Cassandra, as well as execution scripts for the client.

4.2.2 Performance Bugs

These bugs do not lead to crashes but cause significant impact to user satisfaction. A casestudy [Jin *et al.*, 2012] showed that a large percentage of real-world performance bugs can be attributed to uncoordinated functions, executing functions that can be skipped, and inefficient synchronization among threads (for example locks held for too long etc.). Typically, such bugs can be caught by function level execution tracing and tracking the time taken in each execution function. Another key insight provided in [Jin *et al.*, 2012] was that two-thirds of the bugs manifested themselves when special input conditions were met, or execution was done at scale. Hence, it is difficult to capture these bugs with traditional offline white-box testing mechanisms.

4.2.2.1 MySQL #26257

In this subsection we describe the MySQL#15811 performance bug

Cause of the error:

Steps for reproduction:

1. Start an instance of the MySQL server in the production container
2. Using *Parikshan*'s live cloning capability create a clone of the *production container*. This is our *debug container*
3. Start network duplicator to duplicate network traffic to both the production and debug containers
4. connect to the production container using `mysqlclient`

4.2.2.2 MySQL #49491

In this subsection we describe the MySQL#15811 performance bug

Cause of the error:

Steps for reproduction:

1. Start an instance of the MySQL server in the production container
2. Using *Parikshan*'s live cloning capability create a clone of the *production container*. This is our *debug container*
3. Start network duplicator to duplicate network traffic to both the production and debug containers
4. connect to the production container using `mysqlclient`

4.2.2.3 MySQL #15811

In this subsection we describe the MySQL#15811 performance bug

Cause of the error:

For one of the bugs in MySQL#15811, it was reported that some of the user requests which were dealing with complex scripts (Chinese, Japanese), were running significantly slower than others. To evaluate *Parikshan*, we re-created a two-tier client-server setup with the server (container)

running a buggy MySQL server and sent queries to the production container with complex scripts (Chinese). These queries were asynchronously replicated, in the debug container. To further investigate the bug-diagnosis process, we also turned on execution tracing in the debug container using SystemTap [Prasad *et al.*, 2005]. This gives us the added advantage, of being able to profile and identify the functions responsible for the slow-down, without the tracing having any impact on production.

Steps for reproduction:

1. Start an instance of the MySQL server in the production container
2. Using *Parikshan*'s live cloning capability create a clone of the *production container*. This is our *debug container*
3. Start network duplicator to duplicate network traffic to both the production and debug containers
4. connect to the production container using `mysqlclient`
5. Create a table with default charset as latin1: `create table t1(c1 char(10)) engine=myisam default charset=latin1;`
6. Repeat the following line several times to generate a large dataset `insert into t1 select * from t1;`
7. Now create a mysqldump of the table
8. Load this table back again, and observe a significant slow response for large table insert requests. This is magnified several times when using complex scripts

4.2.3 Resource Leaks

Resource leaks can be either memory leak or un-necessary zombie processes. Memory leaks are common errors in service-oriented systems, especially in C/C++ based applications which allow low-level memory management by users. These leaks build up over time and can cause slowdowns because of resource shortage, or crash the system. Debugging leaks can be done either using systematic debugging tools like Valgrind, which use shadow memory to track all objects, or memory profiling tools like VisualVM, mTrace, or PIN, which track allocations, de-allocations, and heap size.

Although Valgrind is more complete, it has a very high overhead and needs to capture the execution from the beginning to the end (i.e., needs application restart). On the other hand, profiling tools are much lighter and can be dynamically patched to a running process.

Let us take Redis#417 for instance, here we had a redis master and slave set up for both production and debug container. We then triggered the bug by running concurrent requests through the client which can trigger the memory leak. The memory leak was easily visible in the debug container by turning on debug tracing, which showed a growing memory usage.

4.2.3.1 Redis #417

In this subsection we describe the Redis#417 performance bug

Cause of the error:

Steps for reproduction:

1. Start a redis service with log level set to verbose. Setting loglevel to verbose ensures that we can view whatever is going on inside the service. This is our *production container*.
2. Create a live clone of the service mapped to a parallel *debug container* which will be used to visualize debugging
3. Start cloning the incoming traffic to both the *production container* and the *debug container* asynchronously using *Parikshan*'s network duplicator

4.2.3.2 Redis #614

In this subsection we describe the Redis#614 performance bug

Cause of the error:

Steps for reproduction:

1. Start a redis service with log level set to verbose. Setting loglevel to verbose ensures that we can view whatever is going on inside the service. This is our *production container*.
2. Create a live clone of the service mapped to a parallel *debug container* which will be used to visualize debugging
3. Start cloning the incoming traffic to both the *production container* and the *debug container* asynchronously using *Parikshan*'s network duplicator

4.2.4 Concurrency Bugs

One of the most subtle bugs in production systems is caused due to concurrency errors. These bugs are hard to reproduce, as they are non-deterministic, and may or may not happen in a given execution. Unfortunately, *Parikshan* cannot guarantee that if a buggy execution is triggered in the production container, an identical execution will trigger the same error in the debug container. However, given that the debug container is a live-clone of the production container, and that it replicates the state of the production container entirely, we believe that the chances of the bug also being triggered in the debug container are quite high. Additionally, the debug container is a useful tracing utility to track thread lock and unlock sequences, to get an idea of the concurrency bug.

4.2.4.1 Apache #25520

In this subsection we describe the Apache#25520 performance bug

Cause of the error:

Steps for reproduction:

1. Start a redis service with log level set to verbose. Setting loglevel to verbose ensures that we can view whatever is going on inside the service. This is our *production container*.
2. Create a live clone of the service mapped to a parallel *debug container* which will be used to visualize debugging

3. Start cloning the incoming traffic to both the *production container* and the *debug container* asynchronously using *Parikshan*'s network duplicator

4.2.4.2 Apache #21287

In this subsection we describe the Apache#21287 performance bug

Cause of the error:

Steps for reproduction:

1. Start a redis service with log level set to verbose. Setting loglevel to verbose ensures that we can view whatever is going on inside the service. This is our *production container*.
2. Create a live clone of the service mapped to a parallel *debug container* which will be used to visualize debugging
3. Start cloning the incoming traffic to both the *production container* and the *debug container* asynchronously using *Parikshan*'s network duplicator

4.2.4.3 MySQL #644

In this subsection we describe the MySQL#644 performance bug

Cause of the error:

Steps for reproduction:

1. Start an instance of the MySQL server in the production container
2. Using *Parikshan*'s live cloning capability create a clone of the *production container*. This is our *debug container*
3. Start network duplicator to duplicate network traffic to both the production and debug containers

4. connect to the production container using `mysqlclient`

4.2.4.4 MySQL #169

In this subsection we describe the MySQL#169 performance bug

Cause of the error:

Steps for reproduction:

1. Start an instance of the MySQL server in the production container
2. Using *Parikshan*'s live cloning capability create a clone of the *production container*. This is our *debug container*
3. Start network duplicator to duplicate network traffic to both the production and debug containers
4. connect to the production container using `mysqlclient`

4.2.4.5 MySQL #791

In this subsection we describe the MySQL#791 performance bug

Cause of the error:

Steps for reproduction:

1. Start an instance of the MySQL server in the production container
2. Using *Parikshan*'s live cloning capability create a clone of the *production container*. This is our *debug container*
3. Start network duplicator to duplicate network traffic to both the production and debug containers
4. connect to the production container using `mysqlclient`

4.2.5 Configuration Bugs

Configuration errors are usually caused by wrongly configured parameters, i.e., they are not bugs in the application, but bugs in the input (configuration). These bugs usually get triggered at scale or for certain edge cases, making them extremely difficult to catch.

A simple example of such a bug is Redis#957, here the slave is unable to sync with the master. The connection with the slave times out and it's unable to sync because of the large data. While the bug is partially a semantic bug, as it could potentially have checks and balances in the code. The root cause itself is a lower output buffer limit. Once again, it can be easily observed in our debug-containers that the slave is not synced, and can be investigated further by the debugger.

4.2.5.1 Redis #957

In this subsection we describe the Redis#957 performance bug

Cause of the error:

Steps for reproduction:

1. Start a redis service with log level set to verbose. Setting loglevel to verbose ensures that we can view whatever is going on inside the service. This is our *production container*.
2. Create a live clone of the service mapped to a parallel *debug container* which will be used to visualize debugging
3. Start cloning the incoming traffic to both the *production container* and the *debug container* asynchronously using *Parikshan*'s network duplicator

4.2.5.2 HDFS #1904

In this subsection we describe the HDFS#1904 performance bug

Cause of the error:

Steps for reproduction:

1. Start a redis service with log level set to verbose. Setting loglevel to verbose ensures that we can view whatever is going on inside the service. This is our *production container*.
2. Create a live clone of the service mapped to a parallel *debug container* which will be used to visualize debugging
3. Start cloning the incoming traffic to both the *production container* and the *debug container* asynchronously using *Parikshan*'s network duplicator

4.3 Summary

Part II

`iProbe`: Creating live debugging friendly applications

Chapter 5

iProbe

5.1 Introduction

Ideally, a production system tracing tool should have zero-overhead when it is not activated and should have a low overhead when it is activated. In other words, its performance should not adversely effect the usage of the traced application. At the same time, it should be flexible enough so as to meet versatile instrumentation needs at run-time for management tasks such as trouble-shooting or performance analysis.

Over the years researchers have proposed many tools to assist in application performance analytics [Luk *et al.*, 2005; Stallman *et al.*, 2002; McDougall *et al.*, 2006; Prasad *et al.*, 2005; Desnoyers and Dagenais, 2006; McGrath, 2009; Linux Manual Ptrace, ; Buck and Hollingsworth, 2000]. While these techniques provide flexibility, and deep granularity in instrumenting applications, they often trade in considerable complexity in system design, implementation and overhead to profile the application. For example, binary instrumentation tools like Intel's PIN Instrumentation tool [Luk *et al.*, 2005], DynInst [Buck and Hollingsworth, 2000] and GNU debugger [Stallman *et al.*, 2002] allow complete blackbox analysis and instrumentation but incur a heavy overhead, which is unacceptable in production systems. Inherently, these tools have been developed for the development environment, hence are not meant for a production system tracer.

Production system tracers such as DTrace[McDougall *et al.*, 2006] and SystemTap[Prasad *et al.*, 2005] allow for low overhead kernel function tracing. These tools are optimized for inserting hooks in kernel function/system calls, and can monitor run-time application behavior over long time

periods. However, they have limited instrumentation capabilities for user-space instrumentation, and incur a high overhead due to frequent kernel context-switches and complex trampoline mechanisms.

Software developers often utilize program print statements, write their own loggers, or use tools like log4j [Gupta, 2003] or log4c [log,] to track the execution of their applications. Those manually instrumented probe points can easily be deployed without additional libraries or kernel support, and have a low overhead to run without impacting the application performance noticeably. However, they are inflexible and can only be turned on/off at compile-time or before starting the execution. Besides, usually only a small subset of functions is chosen to avoid larger overheads.

In this paper, we introduce a dynamic instrumentation framework called *iProbe*. iProbe has instrumentation overheads comparable to print/debug statements, while still giving users the flexibility to choose targets in the execution stage. We evaluated iProbe on micro-benchmark and SPEC CPU 2006 benchmarks. iProbe showed an order of magnitude performance improvement in comparison to SystemTap[McGrath, 2009] and DynInst[Buck and Hollingsworth, 2000] in terms of tracing overhead and scalability. Additionally, the instrumented applications incur negligible overhead when iProbe is not activated. We also present a new hardware event profiling tool called *FPerf* developed in the iProbe framework. FPerf leverages iProbe's flexibility and scalability to realize a fine-grained performance event profiling solution with overhead control. In the evaluation, FPerf was able to obtain function-level hardware event breakdown on SPEC CPU2006 applications while controlling performance overhead (e.g., under 5%)

The main idea in iProbe design is *decoupling the process of run-time instrumentation into offline and and online stages*, which avoids several complexities faced by current state-of-the-art mechanisms [McDougall *et al.*, 2006; Prasad *et al.*, 2005; Buck and Hollingsworth, 2000; Luk *et al.*, 2005] such as instruction overwriting, complex trampoline mechanisms, and code segment memory allocation, kernel context switches etc. Most existing dynamic instrumentation mechanisms rely on a trampoline based design, and generally have to make several jumps to get to the instrumentation function as they not only do instrumentation but also simulate the instructions that have been overwritten. Additionally, they have frequent context-switches as they use kernel traps to capture instrumentation points, and execute the instrumentation. The performance penalty imposed by these designs are unacceptable in a production environment.

Our design avoids any transitions to the kernel which generally causes higher overheads, and is

completely in user space. iProbe can be imagined as a framework which provides a seamless transition from an instrumented binary to a non-instrumented binary. We use a hybrid 2-step mechanism which offloads dynamic instrumentation complexities to an offline development stage, thereby giving us a much better performance. The following are the 2 stages of iProbe:

- **ColdPatch:** We first prepare the target executable by introducing dummy instructions as "place-holders" for hooks during the development stage of the application. This can be done in 3 different ways: Primarily, we can leverage compiler based instrumentation to introduce our "place-holders". Secondly we can allow users to insert macros for calls to instrumentation functions which can be turned on and off at run-time. Lastly we can use static binary rewriter to insert place-holders in the binary without any recompilation. iProbe uses binary parsers to capture all place-holders in the development stage and generates a meta-data file with all possible probe points created in the binary.

- **HotPatch:** We then leverage these place-holders during the execution of the process to safely replace them with calls to our instrumentation functions during run-time. iProbe uses existing tools, ptrace [Linux Manual Ptrace,], to modify the code segment of a running process, and does safety check to ensure correctness of the executing process. Using this approach in a systematic manner we reduce the overhead of iProbe while at the same time maintaining a relatively simple design.

We propose a new paradigm in development and packaging of applications, wherein developers can insert probe points in an application by using compiler flag options, and applying our ColdPatch. An iProbe-ready application can then be packaged along with the meta-data information and deployed in the production environment. iProbe has negligible effect on the application's performance when instrumentation is not activated, and low overhead when instrumentation is activated. We believe this is an useful feature as it requires minimal developer effort, and allows for low overhead production-stage tracing which can be switched on and off as required. This is desirable in long-running services for both debugging and profiling usages.

The rest of the paper is organized as following. Section 5.2 discusses the design of iProbe framework, explaining our ColdPatching, and HotPatching techniques; we also discuss how safety checks are enforced by iProbe to ensure correctness, and some extended options in iProbe for further

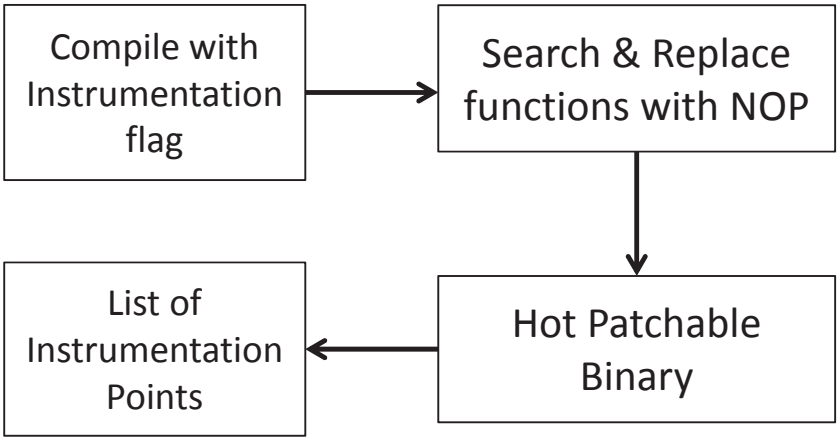


Figure 5.1: The Process of ColdPatching.

flexibility. Section 5.3 compares traditional trampoline based approaches with our hybrid approach and discusses why we perform, and scale better. Section 5.4 explains the implementation of iProbe, and describes *FPerf* a tool developed using iProbe framework. In section 5.5 we evaluate the iProbe prototype, and section 8.3 concludes this paper.

5.2 Design

In this section we present the design of iProbe. Additionally, we then explain some safety checks imposed by iProbe that ensure the correctness of our instrumentation scheme. Finally, we discuss extended iProbe modes, static binary rewriting and user written macros, which serve as alternatives to the default compiler-based scheme to insert instrumentation in the pre-processing stage of iProbe.

The first phase of our instrumentation is an offline pre-processing stage to make the binaries ready for runtime instrumentation. We call this phase *ColdPatching*. The second phase is the an online *HotPatching* stage which instruments the monitored program dynamically at runtime without shutting down and restarting the program. Next, we present the details of each phase.

5.2.1 ColdPatching Phase

ColdPatching is a pre-processing phase which generates the place-holders for hooks to be replaced with the calls for instrumentation. This operation is performed offline before any execution by

statically patching the binary file. This phase is composed of three internal steps that are demonstrated in Figure 5.1.

- Firstly, iProbe uses compiler techniques to insert instrumentation calls at the beginning and end of each function call. The instrumentation parameters, are decided on the basis of the design of the compiler pass. The current implementation by default passes callsite information and the base stack pointer as they can be used to inspect and form execution traces. Calls to the these instrumentation functions must be *cdecl* calls so that stack correctness can be maintained, this is discussed in further detail in Section 5.4.3.

- Secondly, iProbe parses the executable and replaces all instrumentation calls with a `NOP` instruction which is a no-operation or null instruction. This generates instructions in the binary which does no-operation, hence has a negligible overhead, and acts as an empty space for iProbe to be overwritten at run-time.

- Thirdly, iProbe parses the binary and gathers meta-data regarding all the target instrumentation points into a *probe-list*. Optionally, iProbe can strip away all debug and symbolic information in the binary making it more secure and light-weight. The probe-list is securely transferred to the run-time interface of iProbe and used to probe the instrumentation points. Hence iProbe does not have to rely on debug information at run-time to HotPatch the binaries.

5.2.2 HotPatching Phase

Once the application binary has been statically patched (i.e., ColdPatched), instrumentation can be applied at runtime. Compared to existing trampoline approaches, *iProbe does not overwrite any instructions in the original program, or allocate additional memory* when patching the binaries, and still ensures reliability. In order to have a *low overhead*, and *minimal intrusion* of the binary, iProbe avoids most of the complexities involved in HotPatching such as allocation of extra memory in the code segment or scanning code segments to find instrumentation targets in an offline stage. The process of HotPatching is as follows:

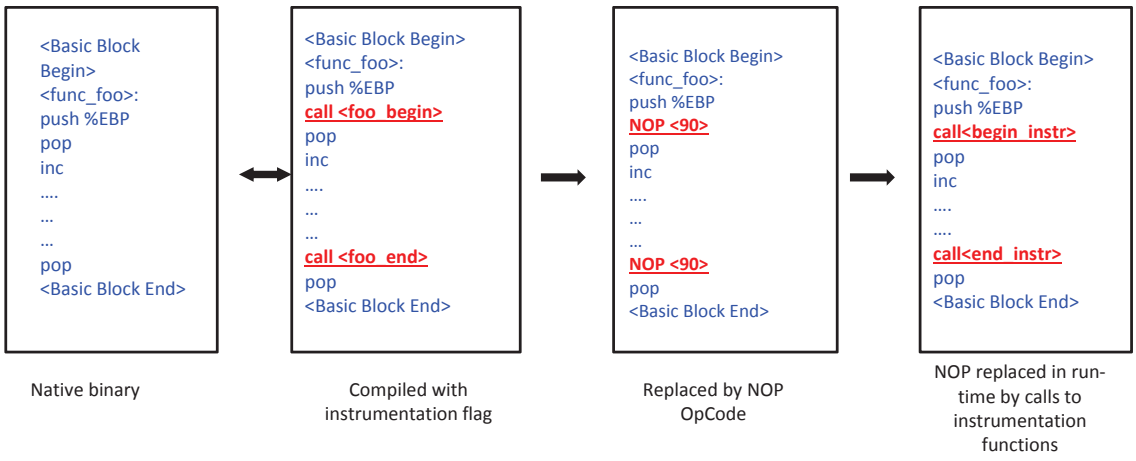


Figure 5.2: Native Binary, the State Transition of ColdPatching and HotPatching.

- Firstly, iProbe loads the relevant instrumentation functions in a shared library to the code-segment of the target process. This along with allocation of `NOP`s in the ColdPatching phase allows iProbe to avoid allocation of memory for instrumentation in the code segment.

- The probe-list generated in the ColdPatching phase is given to our run-time environment as a list of target probe points in the executable. iProbe can handle stripped binaries due to previous knowledge of the target instructions in the ColdPatching.

- As shown in Figure 5.3, in our instrumentation stage, our HotPatcher attaches itself to the target process and issues an interrupt (time T1). It then performs a reliability check (see Section 5.4.3), and subsequently replaces the `NOP` instructions in each of the target functions, with a call to our instrumentation function. This is a key step which enables iProbe to avoid the complexity of traditional trampoline [liv, ; Bratus *et al.*, 2010] by not overwriting any logical instructions (non-`NOP`) in the original code. Since the place-holders (`NOP` instructions) are already available, iProbe can seamlessly patch these applications without changing the size or the runtime footprint of the process. Once the calls have been added iProbe releases the interrupt and let normal execution proceed (time T2).

- At the un-instrumentation stage the same process is repeated, with the exception that the target functions are again replaced with a `NOP` instruction. The period between time T2 and time T3

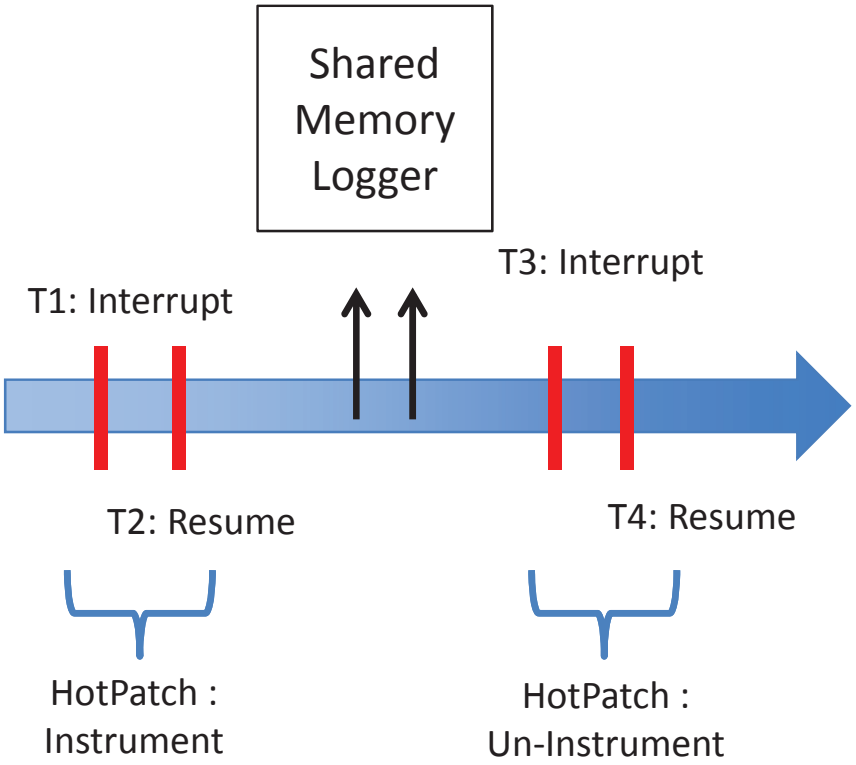


Figure 5.3: HotPatching Workflow.

is our monitoring period, wherein all events are logged to a user-space shared memory logger.

State Transition Flow: Figure 5.2 demonstrates the operational flow of iProbe in the example to instrument the entry and exit of the `func_foo` function. The left most figure represents the instructions of a native binary. As an example, it shows three instructions (i.e., push, pop, inc) in the prolog and one instruction (i.e., pop) in the epilog of the function `func_foo`. The next figure shows the layout of this binary when it is compiled with the instrumentation option. As shown in the figure, two function calls, `foo_begin` and `foo_end` are automatically inserted by the compiler at the start and end of the function respectively. iProbe exploits these two newly introduced instructions as the place-holders for HotPatching. The ColdPatching process overwrites two call instructions with `NOP`s. At runtime, the instrumentation of `func_foo` is initiated by HotPatching those instructions with the call instructions to the instrumentation functions: `begin_instrument` and `end_instrument`. This is illustrated in the right most figure in Figure 5.2.

Logging Functions and Monitoring Dispatchers : The calls from the target function to the instrumentation function are generally defined in the coldpatch stage by the compiler. However, iProbe also provides monitoring dispatchers which are common instrumentation functions that are shared by target functions. Our default instrumentation passes the call site information, and the function address of the target function as parameters to the dispatchers. Each monitoring event can be differentiated by these dispatchers using a quick hashing mechanism representing the source of each dispatch. This allows iProbe to uniquely define instrumentation for each function at run-time, and identify its call sites.

5.2.3 Safety Checks for iProbe

Safety and reliability of the instrumentation technique is a big concern for most run-time instrumentation techniques. One of the key advantages of iProbe is that because of its hybrid design reliability and correctness issues are handled in a better way inherently. In this section we discuss how our HotPatch can achieve such properties in details.

HotPatch check against Illegal instructions: Unlike previous techniques iProbe relies on compiler correctness to ensure safety and reliability in its binary mode. To ensure correctness in our ColdPatching phase, we convert call instructions to instrumentation functions with `NOP` instruction. This does not in any way effect the correctness of the binary, except that instrumentation calls are not made. To ensure run-time correctness, iProbe uses a safety check when it interrupts the application while HotPatching. Our safety check pass ensures that the program counters of all threads belonging to the target applications do not point to the region of code that is being overwritten (i.e. `NOP` instructions are not overwritten while they are being executed. This check is similar to those from traditional Ptrace[Linux Manual Ptrace,] driven debuggers etc [Yamato *et al.*, 2009; liv, ; pan,]. Here we use the Ptrace `GETREGS()` call to inspect the program counter, and if it is currently pointing to the target `NOP` instructions, we allow the execution to move forward before applying the HotPatch. Unlike existing trampoline oriented mechanisms iProbe has a small `NOP` code segments equal to the length of a single call instruction that it overwrites with instrumentation calls, this means that the check can be performed in a fast and efficient manner. It is also important to have this check for all threads which share the code-segment, hence the checking must be able to access the process memory map information, and interrupt all the relevant threads.

Safe parameter passing to maintain stack consistency: An important aspect for instrumentation is the information passed along to the instrumentation function via the parameter values. Since the instrumentation calls are defined by the compiler driven instrumentation, the mechanism in which the parameters passed are decided based on the calling convention used by the compiler.
Calling conventions can be broadly classified in two types: caller clean-up based, and callee clean-up based. In the former the caller is responsible to pop the parameters passed to function, and hence all parameter related stack operations are performed before and after the call instruction inside the code segment of the caller. In the later however, the callee is responsible to pop the parameters passed to it. Since parameters are generally passed using the stack it is important to remove them properly to mantain stack consistency.

To ensure this iProbe enforces that all calls that are made by the static compiler instrumentation must be *cdecl* [cde,] calls where the caller performs the cleanup as compared to *std* calls, where the callee performs it. As the stack cleanup is automatically performed, it maintains stack consistency, and there is a negligible impact in performance due to the redundant stack operations. Alternatively for *std* call convention, push instructions could also be converted to `NOP`s and HotPatched at run-time, we do not do so as a design choice.

Address Space Layout Randomization: Another issue that iProbe addresses is ASLR (address space layout randomization), a security measure used in most environments which randomizes the loading address of executables and shared libraries. However, since iProbe assumes the full access to the target system, the load addresses are easily available. HotPatcher uses the process id of the target to find all load addresses of each binary/shared library and uses them as base offsets to generate correct instruction pointer addresses.

5.2.4 Extended iProbe Mode

As iProbe ColdPatching requires compiler assistance, it is unable to operate on pre-packaged binary applications. Additionally, compiler flags generally have limited instrumentation flexibility as they generally operate on a programming language abstraction(eg. function calls, loops etc.). To provide further flexibility, iProbe provides a couple of extended options for ColdPatching of the application

5.2.4.1 Static Binary Rewriting Mode

In this mode we use a static binary rewriter to insert instrumentation in a pre-packaged binary. Once all functions are instrumented, we use a ColdPatching script to capture all call sites to the instrumentation functions and convert them to `NOP` instruction. While this mode allows us to directly operate on binaries, a downside is that our current static binary instrumentation technique also uses mini-trampoline mechanisms. As explained in Section 5.3 static binary rewriters use trampoline based mechanisms which induces minimum two jumps. In the ColdPatch phase, we convert calls to the instrumentation function to `NOP`s, however the jmp operations to the trampoline function, and simulation of the overwritten instructions still remain. The downside of this approach has a small overhead even when instrumentation is turned off. However, in comparison to pure dynamic instrumentation approach it reduces the time spent in HotPatching. This is especially important if the number of instrumentation targets is high, and the target binary is large, as it will increase the time taken in analyzing the binaries. Additionally, if compiler options cannot be changed for certain sections of the program (plugins/3rd party binaries), iProbe can still be applied using this extended feature.

Our current implementation uses the dyninst [Buck and Hollingsworth, 2000] and cobi [Mussler *et al.*, 2011] to do static instrumentation. This allows us to provide the user a configuration file and template which can be used to specify the level of instrumentation (e.g., all entry and exit points for instrumentation), or names of specific target functions, and the instrumentation to be applied to them. Subsequently in ColdPatch we generate our meta-data list, and use it to HotPatch and apply instrumentation at run-time.

5.2.4.2 Developer Driven Macros

Compiler assisted instrumentation may not provide complete flexibility (usually works on abstractions, such as enter/exit of functions), hence for further flexibility, iProbe provides the user with a header file with calls to macros which can be used to add probe points in the binary. A call to this macro can be placed as required by the developer. The symbol name of the macro is then used in the ColdPatch stage to capture these macros as probe points, and convert them to `NOP`s. Since the macros are predefined, they can be safely inserted and interpreted by ColdPatcher. The HotPatching mechanism is very much the same, using the probe list generated by ColdPatch.

5.2.5 Extended iProbe Mode

As iProbe ColdPatching requires compiler assistance, it is unable to operate on pre-packaged binary applications. Additionally, compiler flags generally have limited instrumentation flexibility as they generally operate on a programming language abstraction(eg. function calls, loops etc.). To provide further flexibility, iProbe provides a couple of extended options for ColdPatching of the application

5.2.5.1 Static Binary Rewriting Mode

In this mode we use a static binary rewriter to insert instrumentation in a pre-packaged binary. Once all functions are instrumented, we use a ColdPatching script to capture all call sites to the instrumentation functions and convert them to `NOP` instruction. While this mode allows us to directly operate on binaries, a downside is that our current static binary instrumentation technique also uses mini-trampoline mechanisms. As explained in Section 5.3 static binary rewriters use trampoline based mechanisms which induces minimum two jumps. In the ColdPatch phase, we convert calls to the instrumentation function to `NOP`s, however the jmp operations to the trampoline function, and simulation of the overwritten instructions still remain. The downside of this approach has a small overhead even when instrumentation is turned off. However, in comparison to pure dynamic instrumentation approach it reduces the time spent in HotPatching. This is especially important if the number of instrumentation targets is high, and the target binary is large, as it will increase the time taken in analyzing the binaries. Additionally, if compiler options cannot be changed for certain sections of the program (plugins/3rd party binaries), iProbe can still be applied using this extended feature.

Our current implementation uses the dyninst [Buck and Hollingsworth, 2000] and cobi [Mussler *et al.*, 2011] to do static instrumentation. This allows us to provide the user a configuration file and template which can be used to specify the level of instrumentation (e.g., all entry and exit points for instrumentation), or names of specific target functions, and the instrumentation to be applied to them. Subsequently in ColdPatch we generate our meta-data list, and use it to HotPatch and apply instrumentation at run-time.

5.2.5.2 Developer Driven Macros

Compiler assisted instrumentation may not provide complete flexibility (usually works on abstractions, such as enter/exit of functions), hence for further flexibility, iProbe provides the user with a header file with calls to macros which can be used to add probe points in the binary. A call to this macro can be placed as required by the developer. The symbol name of the macro is then used in the ColdPatch stage to capture these macros as probe points, and convert them to `NOP`s. Since the macros are predefined, they can be safely inserted and interpreted by ColdPatcher. The HotPatching mechanism is very much the same, using the probe list generated by ColdPatch.

5.3 Trampoline vs. Hybrid Approach

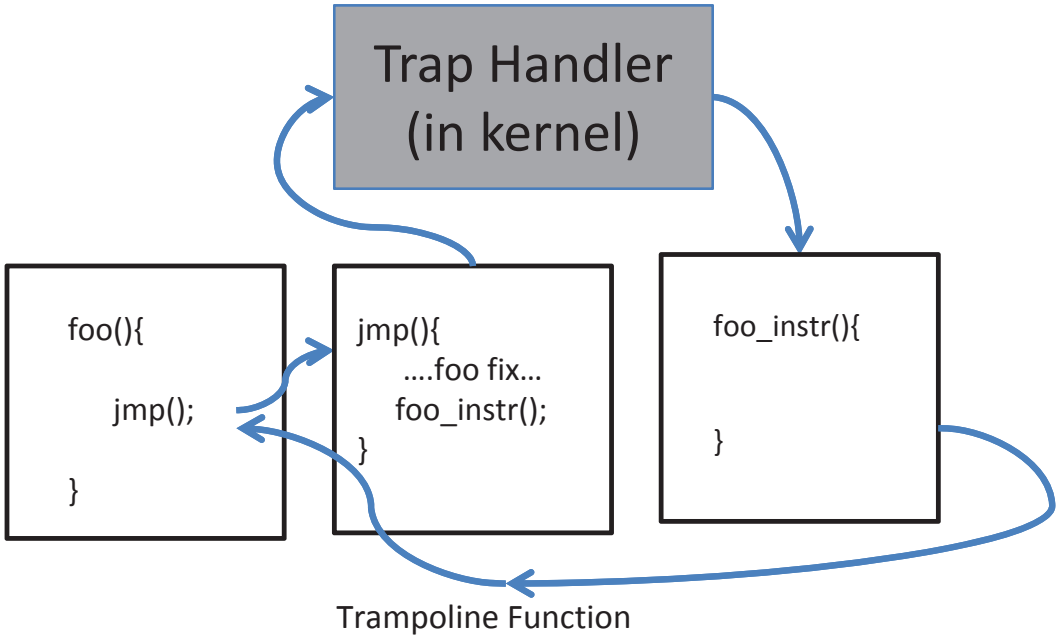


Figure 5.4: Traditional Trampoline based Dynamic Instrumentation Mechanisms.

In this section we compare the advantages of our approach compared to traditional trampoline based dynamic instrumentation mechanisms. We show the steps followed in trampoline mechanisms, and why our approach has a significant improvement in terms of overhead. The basic process of dynamic instrumentation based on trampoline can be divided into 4 steps

- **Inspection for Probe Points:** This step inspects and generates a binary patch for the custom instrumentation to be inserted to the target binaries, and find the target probe points which are the code addresses to be modified.
- **Memory Allocation for Patching:** Appropriate memory space is allocated for adding the patch and the trampoline code to the target binary.
- **Loading and Activation of a Patch:** At run-time the patch is loaded into the target binary, and overwrites the probe point with a jump instruction to a trampoline function and subsequently to the instrumentation function.
- **Safety and Reliability Check:** To avoid illegal instructions, it is necessary to check for safety and reliability at the HotPatch stage, and that the logic and correctness of the previous binary remains.

One of the key reasons for better performance of iProbe as compared to traditional trampoline based designs is the avoidance of multiple jumps enforced in the trampoline mechanism. For instance, Figure 5.4 shows the traditional trampoline mechanism used in existing dynamic instrumentation techniques. To insert a hook for the function `foo()`, dynamic instrumentation tools overwrite target probe point instructions with a jump to a small trampoline function (`jmp()`). Note that the overwritten code by `jmp` should be executed somewhere to ensure the correctness of the original program. The trampoline function executes the overwritten instructions (`foo fix`) before executing the actual code to be inserted. Then this trampoline function in turn makes the call to the instrumentation function (`foo_instr`). Each call instruction can potentially lead to branch mispredictions in the code cache and cause high overhead. Additionally tools like DTrace, and SystemTap [McDougall *et al.*, 2006; Prasad *et al.*, 2005] have the logger in the kernel space, and cause a context switch in the trampoline using interrupt mechanisms.

In comparison iProbe has a `NOP` instruction which can be easily overwritten without resulting in any illegal instructions, and since overwriting is not a problem trampoline function is not required. This makes the instrumentation process simple resulting in only a single call instruction at all times.

In addition pure binary instrumentation mechanisms need to provide complex guarantees of safety and reliability and hence may lead to further overhead. Since the patch and trampoline functions overwrite instructions at run-time correctness check must be made at HotPatch time so

that an instruction overwrite does not result in an illegal instruction, and that the instructions being patched are not currently being executed. While this does not enforce a run-time overhead it does enforce a considerable overhead at the HotPatch stage.

Again iProbe avoids this overhead by offloading this process to the compiler stage, and allocating memory ahead of time.

Another important advantage of our hybrid approach as compared to the trampoline approach is that pure dynamic instrumentation techniques are sometimes unable to capture functions from the raw binary. This can often be because some compiler optimizations may inherently hide function calls boundaries in the binary. A common example of this is *inline functions* where functions are inlined to avoid the creation of a stack frame and concrete calls to these functions. This may be done explicitly by the user by defining the function as *inline* or implicitly by the compiler. Since our instrumentation uses compiler assisted binary tracing, we are able to use the users definition of functions in the source code to capture entry and exit of functions despite such optimizations.

5.4 Implementation

The design of iProbe is generic and platform agnostic, and works on native binary executables. We built a prototype on Linux which is a commonly used platform for service environments. In particular, we used a compiler technique based gcc/g++ compiler to implement the hook place holders on standard Linux 32 bit and 64 bit architectures. In this section we first show the implementation of the iProbe framework, and then discuss the implementation of FPerf a tool built using iProbe.

5.4.1 iProbe Framework

As we presented in the previous section, the instrumentation procedure consists of two stages.

ColdPatch: In the first phase the place holders for hooks are created in the target binary. We implemented this by compiling binaries using the `-finstrument-functions` flag. Note that this can be done simply by appending this flag to the list of compiler flags (e.g., `CFLAG, CCFLAG, CXXFLAGS`) and most of cases it works without interfering with user code.

In details this compiler option places function calls to instrumentation functions (`_cyg_profile_func_enter` and `_cyg_profile_func_exit`) after the entry and before the exit of every function. This in-

cludes inline functions (see second state in Figure 5.2). Subsequently, our ColdPatcher uses a binary parser to read through all the target binaries, and search and replace the instruction offsets containing the instrumentation calls with NOP instructions (instruction 90). Symbolic and debug information is read from the target binary using commonly available `objdmp` tools; This information combined with target instruction offsets are used to generate the probe list with the following information:

```
<Instr Offset, Entry\Exit Point, Meta-Data>
```

The first field is the instruction offset from the base address, and the second classifies if the target is an entry or an exit point of the function. The meta-data here specifies the file, function name, line number etc.

HotPatching: In the run-time phase, we first use the library interposition technique, `LD_PRELOAD`, to preload the instrumentation functions in the form of a shared library to the execution environment. The HotPatcher then uses a command line interface which interacts with the user and provides the user an option to input the target process and the probe list. Next, iProbe collects the base addresses of each shared library and the binary connected to the target process from `/proc/pid/maps`. The load address and offsets from the probe-list are then used to generate a hash of all possible probing points. iProbe then use the meta-data information to provide users a list of target functions and their respective file information. It takes as input the list of targets and interrupts the target process. We then use `ptrace` functionality to patch the target instructions with calls to our instrumentation functions, and release the process to execute as normal. The instrumentation from each function is registered and logged by a shared memory logger. To avoid any locking overhead, we have a race free mechanism which utilizes thread local storage to keep all logs, and a buffered logging mechanism.

5.4.2 FPerf: An iProbe Application for Hardware Event Profiling

We used iProbe to build *FPerf*, an automatic function level hardware event profiler. FPerf uses iProbe to provide an automated way to gather hardware performance information at application function granularity.

Hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units, caches and main memory etc. Using iProbe's all function profiling, we capture the hardware performance counters at the entry and exit of each function. To control the

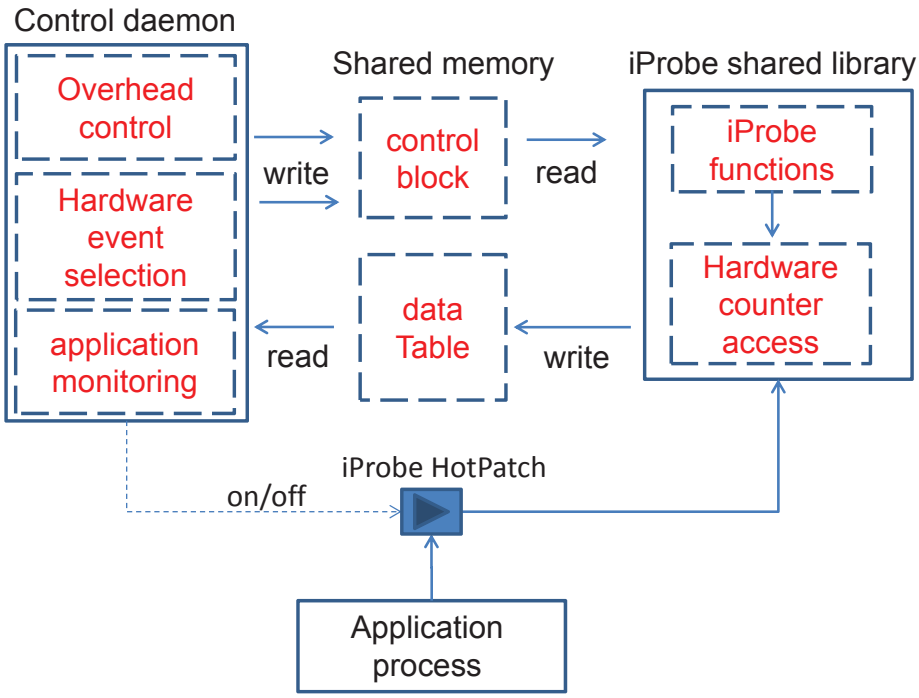


Figure 5.5: Overview of *FPerf*: Hardware Event Profiler based on iProbe.

perturbation on applications and the run-time system, *FPerf* also implements a control mechanism to constraint the function profiling overhead within a budget configured by users.

Figure 5.5 summarizes *FPerf* implementation. It includes a control daemon and an iProbe shared library with customized instrumentation functions. The iProbe instrumentation functions access hardware performance counters (using PAPI[Mucci *et al.*, 1999] in the implementation) at the entry and exit of a selected target function to get the number of hardware events occurring during the function call. We define this process as taking one sample. Each selected function has a budget quota. After taking one sample, the instrumentation functions decrease the quota for that application function by one. When its quota reaches zero, iProbe does not take sample anymore for that function.

The daemon process controls run-time iProbe profiling through shared memory communication. There are two shared data structures for this purpose: a shared control block where the daemon process passes to the iProbe instrumentation functions the profiling quota information, and a shared data table where the iProbe instrumentation functions record the hardware event information for individual function calls. When iProbe is enabled, i.e., the binary is HotPatched, daemon periodically

collects execution data. We limit the total number of samples we want to collect in each time interval to restrict the overhead. This limitation is important because in software execution, the function call happens very frequently. For example, even with test data size input, the SPEC benchmarks generate 50MB-2GB trace files if we log the records for each function call. Functions that are frequently called will get more samples. Each selected function cannot take more samples than its assigned quota. The only exception happens when one function has never been called before; we assign a minimum one sample quota for each selection function. And we pick a function with quota that has not been used up, and decrease the quota of it by one. The above overhead control algorithm is a simplified Leaky Bucket algorithm [Tanenbaum, 2003] originally for traffic shaping in networks. Other overhead control algorithms are also under consideration.

The control daemon also enables/disables the iProbe HotPatching based on user-defined application monitoring rules. Essentially, this is an external control role on when and what to trace a target application with iProbe. A full discussion of the hardware event selection scheme and monitoring rule design is beyond the scope of this paper.

5.4.3 Safety Checks for iProbe

Safety and reliability of the instrumentation technique is a big concern for most run-time instrumentation techniques. One of the key advantages of iProbe is that because of its hybrid design reliability and correctness issues are handled in a better way inherently. In this section we discuss how our HotPatch can achieve such properties in details.

HotPatch check against Illegal instructions: Unlike previous techniques iProbe relies on compiler correctness to ensure safety and reliability in its binary mode. To ensure correctness in our ColdPatching phase, we convert call instructions to instrumentation functions with `NOP` instruction. This does not in any way effect the correctness of the binary, except that instrumentation calls are not made. To ensure run-time correctness, iProbe uses a safety check when it interrupts the application while HotPatching. Our safety check pass ensures that the program counters of all threads belonging to the target applications do not point to the region of code that is being overwritten (i.e. `NOP` instructions are not overwritten while they are being executed. This check is similar to those from traditional Ptrace[Linux Manual Ptrace,] driven debuggers etc [Yamato *et al.*, 2009; liv, ; pan,]. Here we use the Ptrace `GETREGS()` call to inspect the program counter, and if it is currently

pointing to the target `NOP` instructions, we allow the execution to move forward before applying the HotPatch. Unlike existing trampoline oriented mechanisms iProbe has a small `NOP` code segments equal to the length of a single call instruction that it overwrites with instrumentation calls, this means that the check can be performed in a fast and efficient manner. It is also important to have this check for all threads which share the code-segment, hence the checking must be able to access the process memory map information, and interrupt all the relevant threads.

Safe parameter passing to maintain stack consistency: An important aspect for instrumentation is the information passed along to the instrumentation function via the parameter values. Since the instrumentation calls are defined by the compiler driven instrumentation, the mechanism in which the parameters passed are decided based on the calling convention used by the compiler.
Calling conventions can be broadly classified in two types: caller clean-up based, and callee clean-up based. In the former the caller is responsible to pop the parameters passed to function, and hence all parameter related stack operations are performed before and after the call instruction inside the code segment of the caller. In the later however, the callee is responsible to pop the parameters passed to it. Since parameters are generally passed using the stack it is important to remove them properly to mantain stack consistency.

To ensure this iProbe enforces that all calls that are made by the static compiler instrumentation must be *cdecl* [cde,] calls where the caller performs the cleanup as compared to *std* calls, where the callee performs it. As the stack cleanup is automatically performed, it maintains stack consistency, and there is a negligible impact in performance due to the redundant stack operations. Alternatively for *std* call convention, push instructions could also be converted to `NOP`s and HotPatched at run-time, we do not do so as a design choice.

Address Space Layout Randomization: Another issue that iProbe addresses is ASLR (address space layout randomization), a security measure used in most environments which randomizes the loading address of executables and shared libraries. However, since iProbe assumes the full access to the target system, the load addresses are easily available. HotPatcher uses the process id of the target to find all load addresses of each binary/shared library and uses them as base offsets to generate correct instruction pointer addresses.

5.5 Evaluation

5.5.1 Overhead of ColdPatch

The SPEC INT CPU benchmarks 2006 [Henning, 2006] is a widely used benchmark in academia, industry and research as relevant representation of real world applications. We tested iProbe on 8 benchmark applications shown in Figure 5.6. The first column shows the execution of a normal binary compiled without any instrumentation or debug flags. The next column shows the execution time of the corresponding binary compiled using the instrumentation flags (Note here the instrumentation functions are dummy functions). Lastly, we show the overhead of a ColdPatched iProbe binary with `NOP` instead of the call instruction. Each benchmark application was executed ten times using SPEC benchmark tools. The overhead for a ColdPatched binary was found to be less than five percent for all applications executed, and 0-2 percent for four of the benchmarks. The overhead here is basically because of the `NOP` instructions that are placed in the binary as place-holders for the HotPatching. In most non-compute intensive applications (e.g., apache, mysql) we have observed the overhead to be negligible (less than one percent), with no observable effect in terms of throughput. Further reduction of the overhead can be achieved by reducing the scope of the functions which are prepared for function tracing by iProbe; for example only using place holders in selected components that need to be further inspected. Negligible overhead of ColdPatching process of iProbe shows that applications can be prepared for instrumentation (HotPatching) without adversly effecting the usage of the application.

5.5.2 Overhead of HotPatching and Scalability Analysis

We compared iProbe with UTrace (User Space Tracing in SystemTap) [McGrath, 2009], and DynInst [Buck and Hollingsworth, 2000] on a x86_64, dual-core machine with Ubuntu 12.10 kernel. To test the scalability of these tools, we designed a micro-benchmark and tested the overhead for an increasing amount of events instrumented. We instrumented a dummy application with multiple calls to an empty function `foo`, the instrumentation function in the cases simply increases a global counter for each event triggered (entry and exit of foo). Tools were written using all three frameworks to instrument the start and end of the target function and call the instrumentation function.

Figure 5.7 shows our results when applying iProbe and SystemTap on this micro-benchmark. To

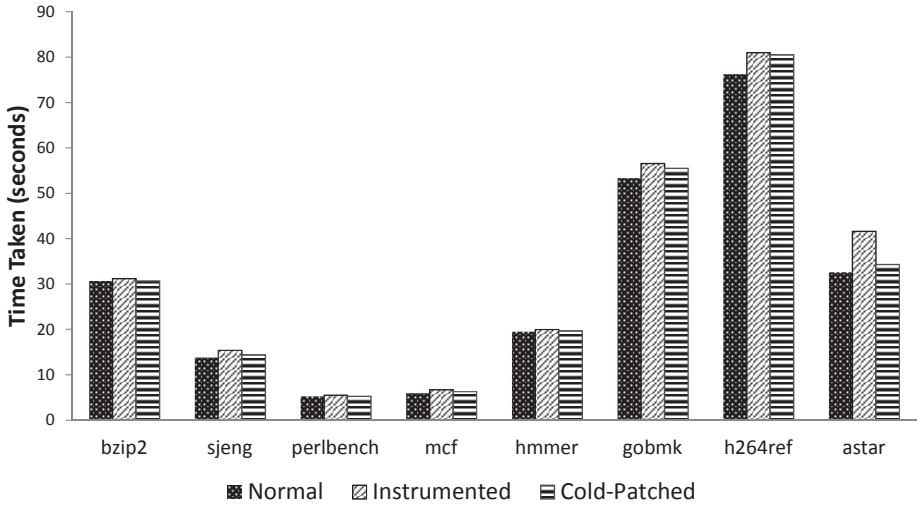


Figure 5.6: Overhead of iProbe "ColdPatch Stage" on SPEC CPU 2006 Benchmarks.

test the scalability of our the tools, we have increased the number of calls made to `foo` exponentially (increase by multiples of 10). We found that iProbe scales very well and is able to keep the overhead to less than five times for millions of events (10^8) generated in less than a second (normal execution) for entry as well as exit of the function. While iProbe executed in 1.5 seconds, the overhead observed in SystemTap is around 20 minutes for completion of a subsecond execution, while DynInst takes about 25 seconds.

As explained in Section 5.3, tools such as DynInst use a trampoline mechanism, hence have a minimum of 2 call instructions for each instrumentation. Additionally SystemTap uses a context switch to switch to the kernel space over and above the traditional trampoline mechanism, resulting in the high overhead, and less scalability observed in our results.

5.5.3 Case Study: Hardware Event Profiling

5.5.3.1 Methodology

In this section, we present preliminary results on FPerf. The purpose of this evaluation is for the illustration of iProbe as a framework for lightweight dynamic application profiling. Towards it, we will discuss the results in the context of two FPerf features in hardware event profiling:

- **Instrumentation Automation**: FPerf automates hardware event profiling on massive func-

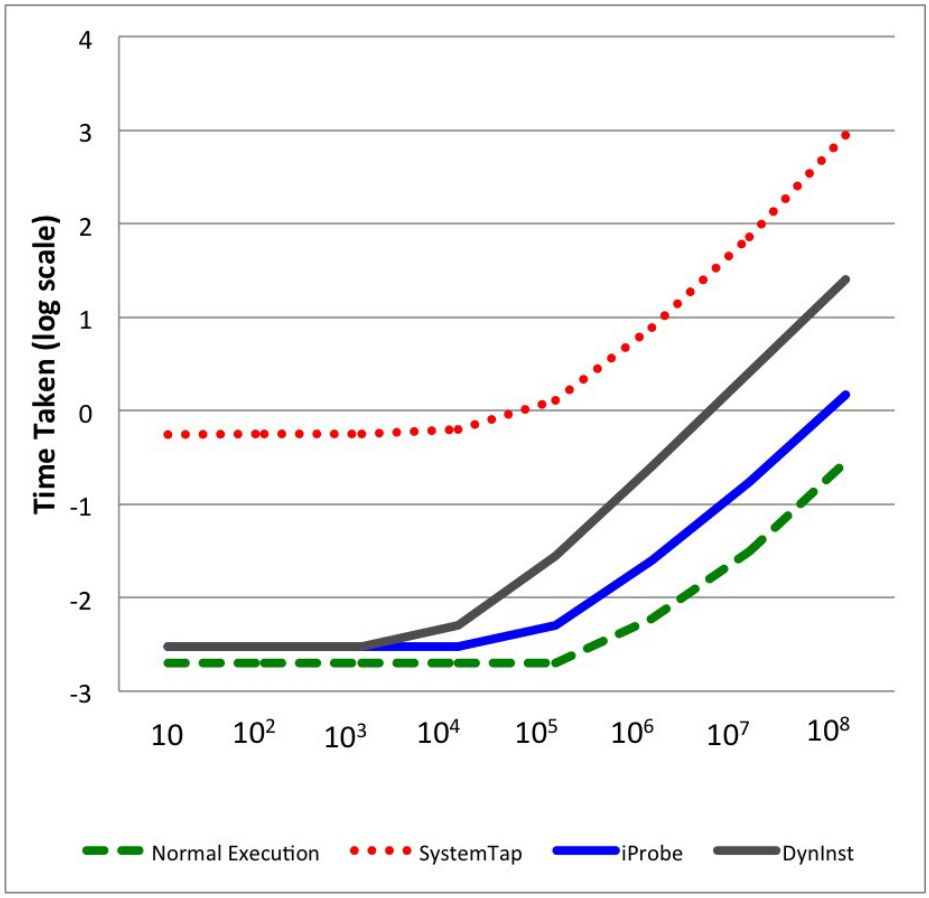


Figure 5.7: Overhead and Scalability Comparison of iProbe HotPatching vs. SystemTap vs. DynInst using a Micro-benchmark.

tions in modern software. This gives a wide and clear view of application performance behaviors.

- **Profiling Automation**: FPerf automates the profiling overhead control. This offers a desired monitoring feature for SLA-sensitive production systems.

While there are many other important aspects on FPerf to be evaluated such as hardware event information accuracy and different overhead control algorithms, we focus on the above two issues related to iProbe in this paper.

Our testbed setup is described in Table 5.1. The server uses an Intel Core^{TM} i5 CPU running at 3.3GHz, and runs Ubuntu 11.10 Linux with 3.0.0-12 kernel. FPerf uses PAPI 5.1.0 for hardware

Table 5.1: Experiment Platform.

CPU	Intel Core^{TM} i5-2500 CPU 3.3GHz
OS	Ubuntu 11.10
Kernel	3.0.0-12
Hardware event access utility	PAPI 5.1.0
Applications	SPEC CPU2006

performance counter reading, and the traced applications are SPEC CPU2006 benchmarks.

5.5.3.2 Instrumentation Automation

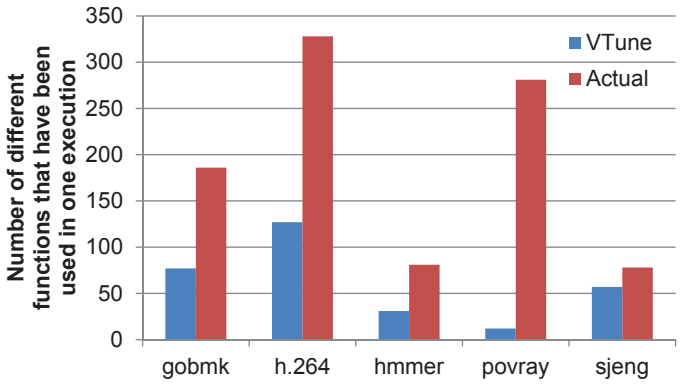


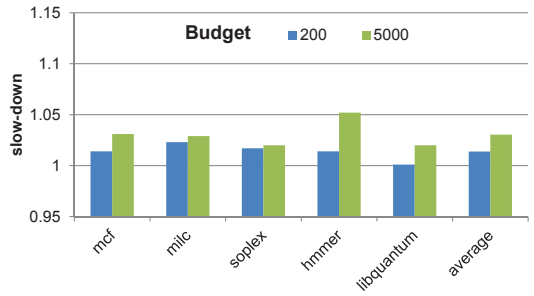
Figure 5.8: The number of different functions that have been profiled in one execution.

Existing profilers featuring hardware events periodically (based on time or events) sample the system-wide hardware statistics and stitch the hardware information to running applications (e.g. Intel VTune [Intel, 2011]). Such sampling based profilers work well to identify and optimize hot code, but with the possibility of missing interesting application functions yet not very hot. In sharp contrast, *FPerf* is based on iProbe framework, it inserts probe functions when entering and exiting each target function. Therefore, *FPerf* can catch all the function calls in application execution. In Figure 5.8, we use VTune and *FPerf* (without budget quota) to trace SPEC workloads with test data set. VTune uses all default settings. We find that VTune misses certain functions. For example, on *453.povray* VTune only captures 12 different functions in one execution. In contrast, *FPerf* does not

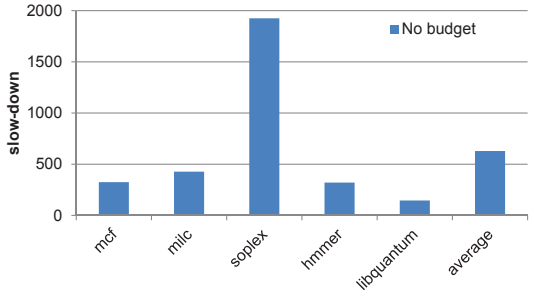
misses any function because it records data at enter/exit of each function. Actually, there are 280 different functions have been used in this execution. having the capability to profile all functions or any subset in the program is desirable. For example, [Jovic *et al.*, 2011] reported that in deployment environment, non-hot functions (i.e., functions with low call frequency) might cause performance bugs as well.

FPerf leverages iProbe's all-function instrumentation and functions-selection utility to achieve instrumentation automation.

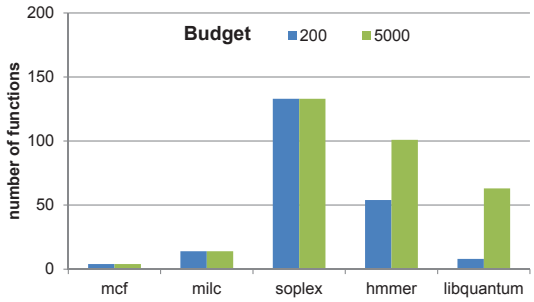
5.5.3.3 Profiling Automation



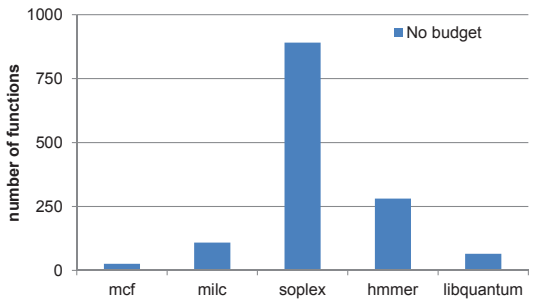
(a) Slow-down with Overhead Control.



(b) Slow-down with No Overhead Control.



(c) Number of Profiled Functions with Limited Budget.



(d) Number of Profiled Functions with No Budgeting.

Figure 5.9: Overhead Control and Number of Captured Functions Comparison.

We tested the measured performance overhead and the number of captured functions of FPerf with different overhead budget. As shown in Figure 5.9, the Y axis of Figure 5.9 (a) and (b) is slow-down, which is defined as the execution time with tracing divided by the execution time without tracing. The Y axis of Figure 5.9 (c) and (d) is the number of profiled functions. The "budget" legend is the total number of samples we assign *FPerf* to take. With no budgeting, *FPerf* records hardware

counter values at every enter/exit points of each function. From Figure 5.9 (b) and (d), no budgeting can capture all the functions but with large 100x-1000x slow-downs. In contrast, FPerf showed its ability to control the performance overhead under 5% in Figure 5.9 (a). Of course, FPerf had the possibility to miss functions, as when the budget is too tight, we only sample a limited number of function enter/exit points.

FPerf leverages iProbe's scalability property (predictable low overhead) to achieve the automation on realizing a low and controllable profiling overhead.

5.6 Summary

Flexibility and performance have been two conflicting goals for the design of dynamic instrumentation tools. iProbe offers a solution to this problem by using a two-stage process that offloads much of the complexity involved in run-time instrumentation to an offline stage. It provides a dynamic application profiling framework to allow for easy and pervasive instrumentation of application functions and selective activation. We presented in the evaluation that iProbe is significantly faster than existing state-of-the-art tools, and scales well in large application software.

Part III

Active Debugging and Applications of Live Debugging

Chapter 6

Active Debugging and Applications of Live Debugging

6.1 Overview

In the sections uptil now, we have introduced a framework for *live debugging*, a tool for pre-packaging binaries to make them *live debugging* friendly. We now discuss some applications of *live debugging* in the real-world, and in particular introduce a new mechanism called *active debugging*.

The debug container allows for debuggers to apply any ad-hoc technique used in offline debugging. However, in order for us to have continuous debugging, it is essential to allow for forward progress in the debug container. Furthermore, the divergence due to instrumentation (either due to slowdown, or functional) should not stop forward-progress in the debug-container. Having a debug-container which is isolated from the production, but still in sync with production services, allows a variety of ad-hoc debugging approaches to be integrated and leverage our framework. Traditional offline debugging approaches, such as dynamic instrumentation can be easily directly applied to the debug-container without any ramifications. Others like memory or performance profiling, and execution traces can also be applied in the debug container. Similarly the *debug container* can be used as a staging mechanism for record-replay on demand to ensure deterministic execution. The key to applying these approaches is that none of them functionally modifies the application.

This chapter is divided in the following key parts: Firstly, in section 6.2, we discuss the advantages and limitations of *Parikshan* in real-world settings. Furthermore, we highlight certain key points that

the debugger most be aware of when debugging applications with *Parikshan*, so that he can do reliable and fast debugging. In section 6.3, we classify debugging production applications in two high level debugging scenario categories: *post-facto*, and *proactive* analysis. We leverage these classifications to explain how different techniques can be applied in *Parikshan*, and in which scenarios debugging on the fly does not make sense. Next we list some existing debugging technologies, like statistical debugging, execution tracing, record and replay etc. to explain how they can be either directly applied or modified slightly and applied with the *Parikshan* framework to significantly improve their analysis.

In the next section, we introduce *active debugging* whereby developers can apply a patch/fix or apply a test in the *debug container*. *active debugging* ensures that any modifications in the *debug container* does not lead to a state change in the *production container*. This will enable the debugger to fix/patch or run test-cases in the debug-container while ensuring forward progress and in sync with production. We are inspired from existing perpetual invivo testing frameworks like INVITE [Murphy *et al.*, 2009](which also provides partial test-isolation), in the production environment. We currently restrict fldebugging to patches/and test-cases only bring about local changes, and not global changes in the application.

In the last section we introduce budget-limited adaptive instrumentation. Which focuses further on how to allow for continuous debugging with the maximum information gain. One of the key criteria for successful statistical debugging is to have higher instrumentation rates, to make the results more statistically significant. There is a clear trade-off between instrumentation vs performance overhead for statistical instrumentation. A key advantage of using this with *Parikshan* is that we can provide live feedback based buffer size and bounded overheads, hence squeezing the maximum advantage out of statistical debugging without impacting the overhead. We evaluate the slack available in each request for instrumentation without risking a buffer overflow and getting out of sync of the production container. Live interactive debugging provided by *Parikshan* further allows for quick updates to the instrumentation and targeting specific areas of the application code. Budget limited instrumentation is inspired from statistical debugging [Liblit *et al.*, 2005], and focuses on a two-pronged goal of bounding the instrumentation overhead to avoid buffer overflows in *Parikshan*, and simulatneously have maximum feedback regarding information gained from real-time instrumentation.

6.2 Live Debugging using Parikshan

The following are some key advantages of *Parikshan* that can be leveraged for debugging user-facing applications on the fly:

- **Sandboxed Environment**: The debug container runs in a sandboxed environment which is running in parallel to the real production system, but any changes in the *debug container* are not reflected to the end-user. This is a key advantage in several debugging techniques which are disruptive, and can change the final output.

  Normal debugging mechanisms such as triggering a breakpoints or patching in a new exception/assertion to figure out if a particular ”condition” in the application system state is breaking the code, cannot be done in a production code as it could lead to a critical system crash. On the other hand, *Parikshan*'s debug containers are ideal for this scenario as they will allow developers to put in a patches without any fear of system failure.

- **Zero Overhead Monitoring**:The most novel aspect of *Parikshan* is that it allows for zero overhead monitoring of the production system. This means that high-overhead debugging techniques can be applied on the debug-container without incurring a slow-down in production.

  Debugging techniques like record-replay tools which have traditionally high recording overheads can generally not be applied in production systems. However, *Parikshan* can be used to decouple the recording overhead from production, and can allow for relatively higher overhead recording with more granularity.

- **Capturing production system state**: One of the key factors behind capturing the root-cause of any bug is to capture the system state in which it was triggered. *Parikshan* has a live-cloning facility that clones the system state and creates a replica of the production. Assuming that the bug was triggered in the production, the replica captures the same state as the production container.

- **Compartmentalizing large-scale systems context**: Most real-world services are deployed using a combination of several SOA applications, each of them interacting together to provide an end-to-end service. This could be a traditional 3 tier commerce system, with an application layer, a database layer and a web front-end, or a more scaled out social media system with

compute services, recommendation engines, short term queuing systems as well as storage and database layers. Bugs in such distributed systems are particularly difficult to re-create as they require the entire large-scale system to be re-created in order to trigger the bug. Traditional record-replay systems if used are insufficient as they are usually focused on a small subset of applications.

Since, our framework leverages network duplication, *Parikshan* can allow looking at applications in isolation and capturing the system state as well as the input of the running application, without having to re-create the entire buggy infrastructure. In a complex multi-tier system this is a very useful advantage to localize the bug.

At the same time there are some things that the administrator must be aware of

- **Continuous Debugging and Forward Progress**: *live debugging* allows users to debug applications on-the-fly. The debug-container where one can do debugging runs in parallel to the production container. This is done by first making a live replica of the system followed by duplicating and sending the network input to the *debug container*. In a way the *debug container* still communicates with the entire system although it's responses are dropped. To ensure forward progress in the *debug container*, it is essential that the *debug container* is in-sync with the production container, so that the responses, and the requests from the network are the expected responses for forward progress in the application running on the *debug container*.

  Take for instance, a MySQL [MySQL, 2001] service running in the *production container* and *debug container*. If during our debugging efforts we modify the state of the debug service such that the MySQL database is no longer in synch with the production service, then any future communication from the network could lead to the state of the debug-container to further diverge from the production. Additionally, depending on the incoming requests or responses the debug application may crash or not have any forward progress.

  No forward-progress does not necessarily mean that debugging cannot take place, however for continuous debugging to take place, once the machine has crashed it needs to be re-cloned from the production container, for further debugging. Further, given programmer knowledge the debugger can try his best to avoid state changes that would lead to an un-intentional crash.

- **Debug Window**: As explained earlier, most debugging mechanisms generally requires instrumentation and tracking execution flow. This means that the application will spend some compute cycles in logging instrumentation points thereby having a slow-down. While *Parikshan* avoids any slow-down in the production environment, there will be some slow-down in the debug-container.

  The amount of time till which the production container remains in synch with the *debug container* is called the debug-window(see section 3.2.3). The window time depends on the overhead, the size of the buffer and the incoming request rate. If a buffer overflow happens because the debug-window has finished, the *debug container* needs to be re-synced with the production container.

  In our experiments, we have observed, that *Parikshan* is able to accomodate significant overhead without incurring a buffer overflow. Administrators or debuggers using *Parikshan* should keep the overhead of their instrumentation in mind when debugging in *Parikshan*. Ofcourse they can always re-clone the *production container* to start a new debugging session.

- **Non-determinism**:

  One of the most difficult bugs to localize are non-deterministic bugs. While *Parikshan* is able to capture system non-determinism by capturing the input, it is unable to capture thread non-determinism. Most service-oriented applications have a large number of threads/processes, which means that different threading schedules can happen in the production container as compared to the debug-container. This means, that a specific ordering of events that caused a bug to be triggered in the production container, may not happen in the debug-container.

  There are multiple ways that this problem can be looked at. Firstly, while it's difficult to quantify, for all the non-deterministic cases in our case-studies, we were able to trigger the bug in both the production and the replica. In the case where the bug is actually triggered in the *debug container*, the debugging can take place as usual for other other bugs. If that is not the case, there are several techniques which provide systematic "search" for different threading schedules based on a high granularity recording of all potential thread synchronization points, and read/write threads. While such high granularity recording is not possible in the production container, it can definitely be done in the *debug container* without any impact on the production

service.

6.3 Debugging Scenarios

Based on our casestudies, and survey of commonly seen SOA bugs we classify the following scenarios for live debugging. In each of the scenarios we explain how different categories of bugs can be caught or analyzed.

6.3.1 Scenario 1: Post-Facto Analysis

In this scenario, the error/fault happens without *live debugging* having been turned on i.e. the service is only running in the production container, and there is no replica. Typically light-weight instrumentation or monitoring is always turned on in all service/transaction systems. Such monitoring systems are very limited in their capabilities to localize the bug, but they can indicate if the system is in a faulty state.

For our post-facto analysis, we use such monitoring systems as a trigger to start *live debugging* once faulty behvaior is detected. The advantage of such an approach is that debugging resources are only used on-demand, and in an otherwise normal system only the *production container* is utilizing the resources.

There are three kind of bugs that can be considered in this kind of situation:

- **Persistent Stateless Bugs**:

  This is the ideal scenario for *Parikshan*. Persistent bugs are those that persist in the application and are long running. They can impact either some or all the requests in a SOA application. Common examples of such bugs are memory leak, performance slow-down, semantic bugs among others. Assuming they are statistically significant, persistent bugs will be triggered again and again by different requests.

  We define *stateless bugs* here as bugs which do not impact the state of the system, hence not impacting future queries. For instance read only operations in the database are stateless, however a write operation which corrupts or modifies the database is stateful, and is likely to impact and cause errors in future transactions.

Hence, such bugs are only dependent on the current system state, and the incoming network input. Once such a bug is detected in the production system, *Parikshan* can initiate a live cloning process and create a replica for debugging purposes. Assuming similar inputs which can trigger the bug are sent by the user, the bug can be observed and debugged the *debug container*.

- **Persistent Stateful Bugs**:

  Stateful bugs are those bugs which can impact the state and change it such that any such bug impacts future transactions in the production container as well. For instance in a database service a table may have been corrupted, or it's state changed so that certain transactions are permanently impacted. While having the execution trace of the initial request which triggered a faulty state is useful, the ability to analyze the current state of the application is also extremely useful in localizing the error.

  Creating a live clone after such an error and checking the responses state of future impacted transaction, as well as the current state of the database can be a good starting point towards resolving the error.

- **Crashing Bugs**:

  Crashing bugs are bugs that lead to a crash in the system thereby stopping the service. Unhandled exceptions, or system traps are generally the cause of such crashes. Unfortunately *Parikshan* has limited utilization for post-facto analysis of a crashing bug. Since *Parikshan* is not turned "on" at the time of the crash, any post-facto analysis for creating a *debug container* is not possible.

6.3.2 Scenario 2: Proactive Analysis

Proactive analysis are scenarios where the user starts debugging when the system is performing normally and their is no bug. This is the same as having instrumentation on or monitoring a production server, except that in this case the instrumentation is actually present in the *debug container*.

One possible scenario is to use the *debug container* to do recording or high granularity instrumentation. This is extremely useful feature to have if you expect to have higher overheads of instrumentation, which are unacceptable in the production environment. On the other hand our *debug*

container can have much higher instrumentation without any performance penalty on the production container. Another useful feature of staged recording is the case where the debugger needs to put in breakpoints or potentially instrumentation which can cause the system to crash can also be put in the *debug container* Proactive recording is basically use to track bugs that could happen in the future as the transaction or request which causes the failure is caught as well, as well as it is caught in the context of the system state. Once a bug is caught, the cause can be independently explored in the *debug container*.

Proactive Analysis, is useful for both stateless and stateful bugs, we do not differentiate between them here as even in the case of a stateful bug, debugging is always turned on. Proactive approaches can be compared to existing approaches like statistical debugging [Liblit *et al.*, 2005] which use active statistical profiling to compare between successful and buggy runs, to isolate the problem. We discuss statistical debugging in section 6.4.2 and present an advanced approach in section 6.5. Other proactive body of work include record-replay infrastructures, which record production systems, and can replay the execution if a bug is discovered. In section 6.4.3, we have discussed staged record-and-replay, which is an advanced record-replay technique that can be applied with the help of *Parikshan*.

6.4 Existing Debugging Mechanisms

6.4.1 Execution Tracing

One of the most common techniques to debug any application is execution tracing. Execution tracing gives a trace log of all the functions/modules executed when an input is received. This helps the debugger in looking at only those execution points and makes it easier to reason out what is going wrong.

Execution tracing can happen at different granularities: for instance an application can be monitored at function level granularity (only entry and exit of function is monitored), or for deeper understanding at read/write, synchronization point or even instruction level granularity. Depending on how much granularity the tracing is done at the overhead may be unacceptable for production systems.

Execution tracing can be de-coupled from original execution and put in *debug container* for

higher granularity instrumentation. Such techniques which decouple execution and analysis also been previously explored in papers like Aftersight [Chow *et al.*, 2008] and DoublePlay [Veeraraghavan *et al.*, 2012]. *Parikshan* provides a systematic framework for decoupling without enforcing strong constraints on the user.

6.4.2 Statistical Debugging

Statistical debugging aims to automate the process of isolating bugs by profiling several runs of the program and using statistical analysis to pinpoint the likely causes of failure. The seminal paper on statistical debugging [Liblit *et al.*, 2005], has lead to several advanced approaches [Chilimbi *et al.*, 2009; Arumuga Nainar and Liblit, 2010; Song and Lu, 2014], and is now a well established debugging methedology.

The core mechanism of statistical debugging is to have probabilistic profiling, by sampling execution points and comparing the execution traces for failed and successful transactions. It then uses statistical models to identify path profiles that are strongly predictive of failure. This can be used to iteratively localize the bug causing execution, and can then be manually analyzed by *Parikshan*.

Statistical debugging is one of the systematic bug localization approaches that can be directly applied in the *debug container*, with the added advantage that the amount of instrumentation that can be applied in the debug-container is much higher than production containers. Apart from regular semantic bugs, previous body of works have shown that statistical debugging is useful in detecting a variety of other bugs like concurrency bugs [Jin *et al.*, 2010], and performance [Song and Lu, 2014].

6.4.3 Staging Record and Replay

One well known sub-category of debugging service-oriented applications are record-replay infrastructures. In the past decade there have been numerous record-and-replay infrastructures which have been introduced in academia. The core focus of these techniques is to faithfully reproduce the execution trace and allow for offline debugging. However, in order faithfully reproduce the exact same instrumentation, the recording phase must record a higher granularity of execution. Unfortunately, this means a higher overhead at the time of recording in the production system. Such recording overhead is usually unacceptable in most production systems.

Record and replay can be coupled with the *debug container* to avoid any overhead on the

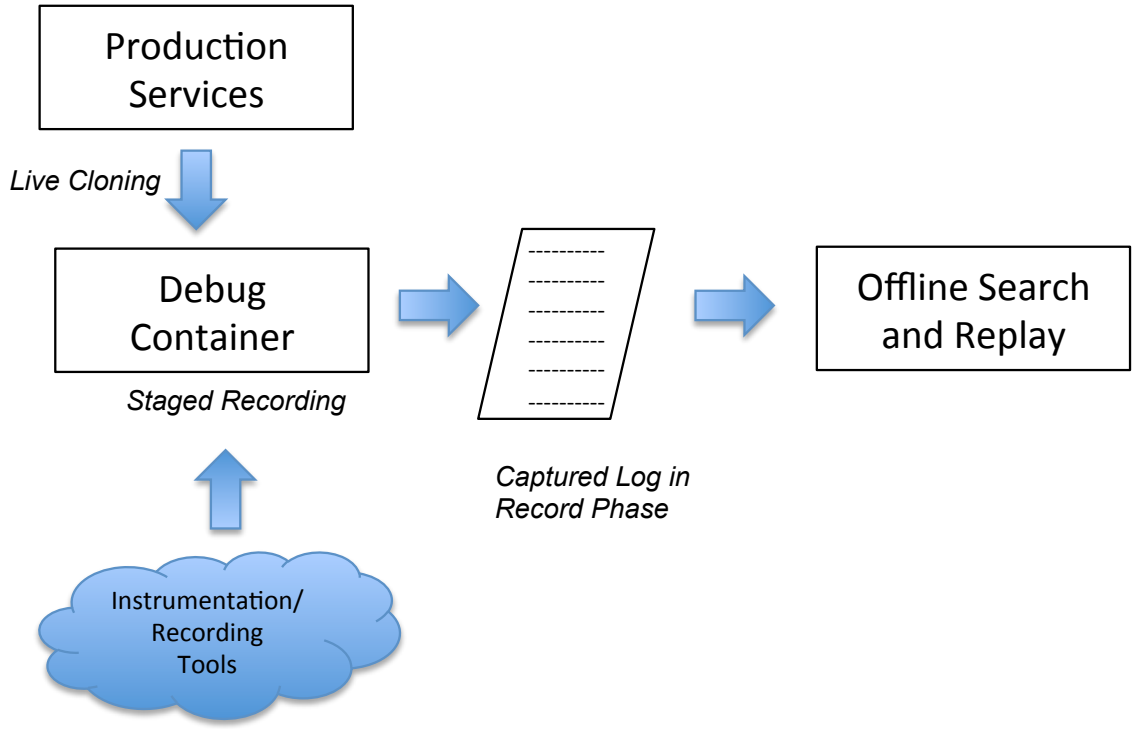


Figure 6.1: Staged Record and Replay using *Parikshan*

production container. This is done by staging the recording for record-and-replay in the *debug container* instead of the production, and then replaying that for offline analysis. In figure 6.1 we show how the production system can first be ”live-cloned”. A copy of the container’s image can be stored/retained for future offline replay - this incurs no extra overhead as taking a live snapshot is a part of the live-cloning process. Recording can then be started on the *debug container*, and logs collected here can be used to do offline replay.

An important aspect to remember here is that **non-determinism** can lead to different execution flows (with low probability as the system is a clone of the original) in the *debug container* v.s. the *production container*. Hence simply replaying an execution trace in the *debug container*, may not lead to the same execution which triggers the bug. However several existing record-and-replay techniques offer ”search” capabilities to replay and search through all possible concurrent schedules which could have triggered a non-deterministic error. Hence we propose that the exact executing schedule which triggered a bug, and using the *debug container* for recording instead of the *production container* is a viable alternative to traditional record-replay techniques.

6.4.4 A-B Testing

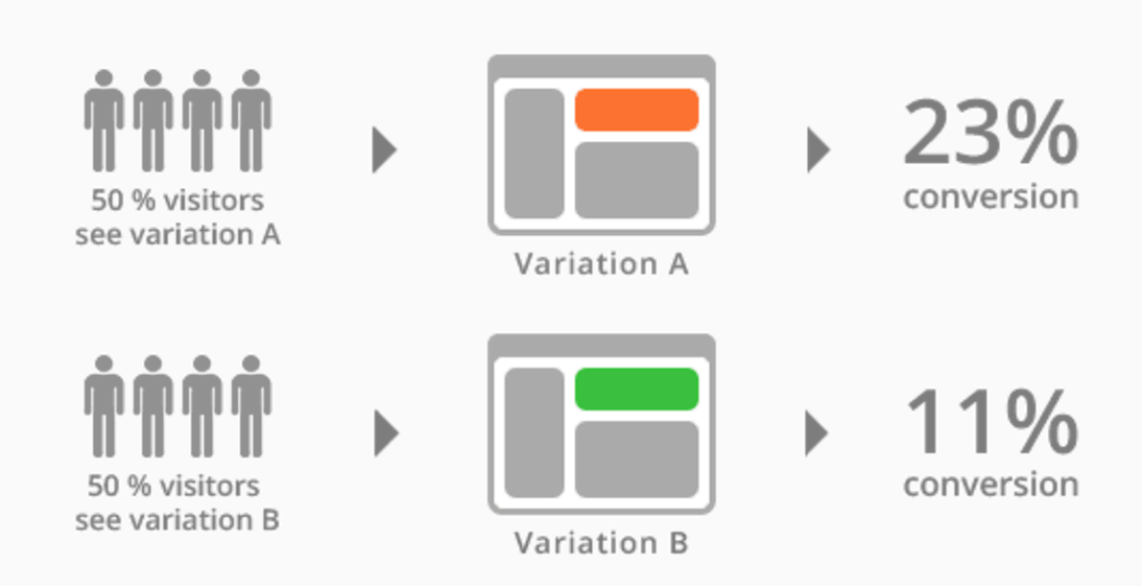


Figure 6.2: Traditional A-B Testing

A/B testing (sometimes called split testing) is comparing two versions of a web page to see which one performs better. You compare two web pages by showing the two variants (let's call them A and B) to similar visitors at the same time. Uer operations in A can then be compared to user scenario's in B to understand which is better, and how well it was received.

Similar A/B Testing, for performance patches or other functional patches, can be done in the *debug container* for evaluation of patches which are functionally similar and have same input/output.

6.4.5 Interactive Debugging

The most common debugging tools used in the development environment are interactive debuggers. Debugging tools like gdb [Stallman *et al.*, 2002], pdb, or eclipse [D'Anjou, 2005], provide intelligent debugging options for doing interactive debugging. This includes adding breakpoints, watchpoints, stack-unrolling etc. The downside to all of these techniques is that they essentially stop the service or execution. Once a process has been attached to a debugger, a shadow process in also attached to it and the rest of the execution follows with just-in-time execution, allowing the debugger to monitor the progress step-by-step therefore making it substantially easier to catch the error. Unfortunately,

this cannot be applied to the *production container*.

However this can be easily applied towards the *debug container*, where the execution trace can be observed once a breakpoint has been reached. While this does mean that the replica will not be able to servce any more requests (except for those that have been buffered), the request which is inside the breakpoint will be processed. Generally breakpoint and step-by-step execution monitoring is used for a limited scope of execution within a single transaction. Once, finished future transactions can also be debugged after doing a re-sync by applying live cloning again.

6.5 Budget Limited, Adaptive Instrumentation

As explained in section 3.2, the asynchronous packet forwarding in our network duplication results in a *debug window*. The *debug window* is the time before the buffer of the debug-container overflows because of the input from the user. The TCP connection from end-users to production-containers are synchronized by default. This means that the rate of incoming packets is limited by the amount of packets that can be processed by the production container. On the other hand, packets are forwarded asynchronously to an internal-buffer in the debug-container. The duration of the *debug window* is dependent on the incoming workload, the size of the buffer, and the overhead/slowdown caused due to instrumentation in the debug-container. Each time the buffer is filled, requests are dropped, and the debug-container can get out of sync with the production container. To get the debug-container back in sync, the container needs to be re-cloned. While duplicating the requests has negligible impact on the production container, cloning the production container can incur a small suspend time(workload dependent).

The duration of the *debug window* can be increased by reducing the instrumentation. At the same time we wish to increase the maximum information that can be gained out of the instrumentation to do an effective bug diagnosis. Essentially for a given buffer size and workload, there is a trade-off between the information gain due to more instrumentation and the duration of the *debug window*. Hence our general objective is to increase the information gain through instrumentation while avoiding a buffer overflow.

We divide this task into pro-active and re-active approaches which can complement each other. Firstly, we pro-actively assign budgets using queuing theory and simulations. We try to find that

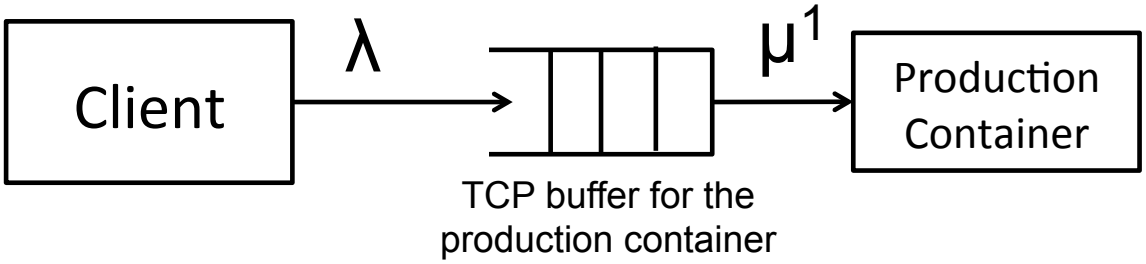


Figure 6.3: *Parikshan* applied to a mid-tier service

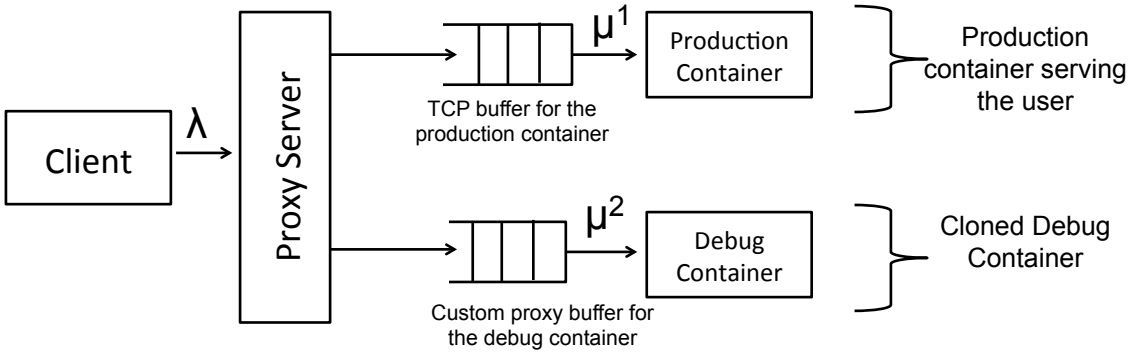


Figure 6.4: External and Internal Mode for live cloning: P1 is the production, and D1 is the debug container.

assuming a poisson distribution for average processing time of each request and the inter-arrival time of requests, we can find expected buffer sizes for a reasonable debug-window length. Secondly, we propose a re-active mechanism triggered by the amount of buffer used. The buffer utilization can be continuously monitored and the instrumentation sampling can be exponentially reduced if the buffer is near capacity. This can be combined with statistical debugging to have the maximum information gain possible.

6.5.1 Proactive: Modeling Budgets

In this section we try to model the testing window by using concepts well used in queuing theory (for the sake of brevity we will avoid going into too much detail, readers can find more about queuing theory models in [Cooper, 1972]). Queueing theory is commonly used to do capacity planning for service-oriented architectures(SOA). Queues in a SOA application can be modeled as a M/M/1/K

queue (Kendall's notation [Kendall, 1953]). Kendall's notation is a well known model which allows a compact representation for queues in SOA architectures. This is a shorthand notation of the type A/B/C/D/E where A, B, C, D, E describe the queue. The standard meanings associated with each of these letters are summarized below.

A represents the *inter-arrival time distribution*

B represents the *service time distribution*

C gives the *number of servers* in the queue

D gives the *maximum number of jobs that can be there in the queue*. **E** represents the Queueing Discipline that is followed. The typical ones are First Come First Served (FCFS), Last Come First Served (LCFS), Service in Random Order (SIRO) etc. If this is not given then the default queueing discipline of FCFS is assumed.

The different possible distributions for **A** and **B** above are:

M exponential distribution

D deterministic distribution

$\mathbf{E_k}$ Erlangian (order k)

G General

Figure 6.3 represents a simple client-server TCP queue in an SOA architecture based on the M/M/1/K queue model. An M/M/1/K queue, denotes a queue where requests arrive according to a poisson process with rate λ, that is the inter-arrival times are independent, exponentially distributed random variables with parameter λ . The service times are also assumed to be independent and exponentially distributed with parameter μ. Furthermore, all the involved random variables are supposed to be independent of each other. In the case of a blocking TCP queue common in most client-server models, the incoming request rate from the client is throttled based on the request processing time of the server. This ensures that there is no buffer-overflows in the system.

In *Parikshan*, this model can be extended to a cloned model as shown in figure 6.4. The packets to both the production and the debug cloned containers are routed through a proxy which has internal

buffer to account for slowdowns in request processing in the debug container. Here instead of the TCP buffer, we focus on the request arrival and departure rate to and from the proxy duplicators buffer. The incoming rate remains the same as λ, as the requests are asynchronously forwarded to both containers without any slowdown. To simplify the problem, we assume the outgoing requests from the proxy buffer to be μ_3 where:

$$\mu_3 = \mu_2 - \mu \tag{6.1}$$

The remaining processing time for both the production container and the debug container is going to be the same. Since the TCP buffer in the production container is a blocking queue, we can assume that any buffer overflows in the proxy buffer are only caused because of the instrumentation overhead in the debug-container, which is accounted for by μ_3.

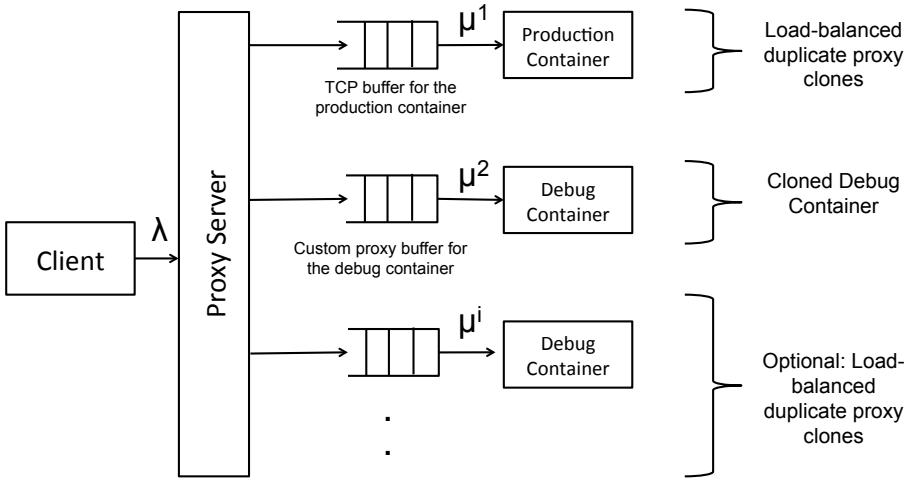


Figure 6.5: This figure shows how queueing theory can be extended to a load balanced debugging scenario. Here each of the debug container receive the requests at rate λ, and the total instrumentation is balanced across multiple debug containers.

6.5.2 Extended Load-balanced duplicate clones

Our model can be further extended into a load-balanced instrumentation model as shown in figure 6.5. This is useful when the debugging needs to be higher, but we have a lower overhead bound through only one clone. Here we can balance the instrumentation across more than one clones, each of which receive the same input. They can together contribute towards debugging the error

6.5.3 Simulation

In this section we present our evaluation strategy for simulating various workloads to model the hitting time and find how much resources to be alloted/instrumentation can be done for a given workload. Our goal of these experiments is to provide the user with an upper-bound for instrumentation in terms of percentage of the average transaction time, based on a given buffer size, and expected time-window needed for debugging.

In our experiments, we plot the time-window observed based on the hitting time explained in the equation **??**. For a given buffer size, we run multiple iterations of instrumentation until we hit the pre-defined time limit. In this case we assume that the time-taken to capture a trace good enough for debugging the problem, is 10 mins. Please note, this is not the total debugging time, but rather just a trace instance large enough to capture the systems and root-cause. The time window will depend on the use-case, and is a user-specified argument.

6.5.4 Reactive: Adaptive Instrumentation

Adaptive instrumentation reduces or increases sampling rate of the dynamic instrumentation in order to decrease the overhead. This allows the debug-container time to catch up to the production container without causing a buffer overflow.

A mechanism similar to TCP's network congestion avoidance mechanisms can be applied on the monitoring buffer. We also derive inspiration from statistical debugging [Song and Lu, 2014; Chilimbi *et al.*, 2009; Liblit *et al.*, 2005], which shows how probabilistically instrumenting *predicates*, can assist in localizing and isolating the bug. Predicates can be branch conditionals, loops, function calls, return instructions and if conditionals. Predicates provide significant advantages in terms of memory/performance overheads. Instead of printing predicates, they are usually counted, and a

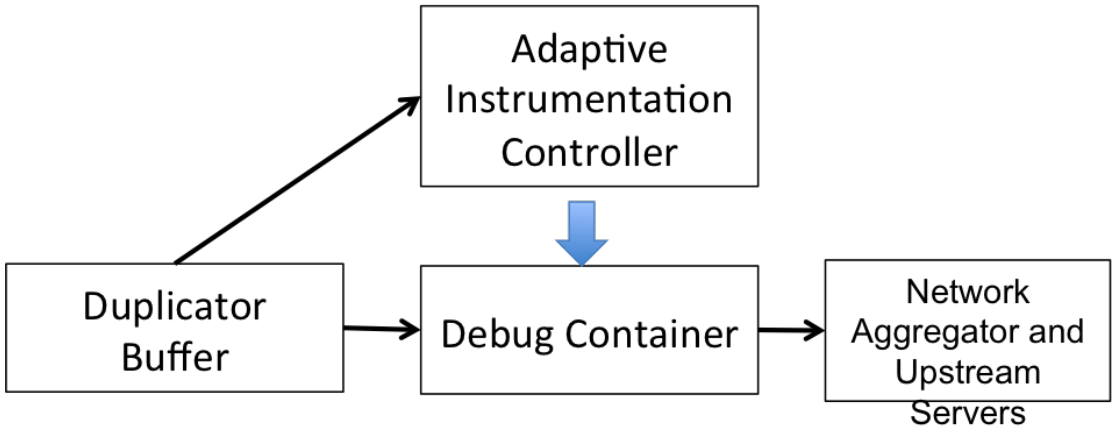


Figure 6.6: Reactive Instrumentation

profile is generated. This reduces the amount of instrumentation overhead, and several predicates can easily be encoded in a small memory space. Similar techniques have also been applied for probabilistic call context encoding in-order to capture execution profiles with low overhead.

The sampling rate of instrumentation in the debug-container can be modified based on the amount of buffer usage. There are three key components of our adaptive instrumentation mechanism.

- **Monitoring Buffer:** The first step involves monitoring the buffer usage of the network duplicator. If the buffer usage is more than X percentage of the buffer, the sampling rate of instrumentation can be exponentially decreased. This would increase the idle time in the debug container allowing it to catch up to the production and reducing the buffer usage.

- **Controller:** The controller allows debuggers to control the sampling rate of instrumentation. The sampling rate can be controlled for each predicate. Similar to statistical debugging the predicates with lower frequency can have higher sampling rates, and predicates with higher frequency can have lower sampling rates. This ensures overall better information gain in any profile collected.

We have looked into assigning a score to each predicate. This is based on the statistical importance of the predicate (similar to statistical debugging), and instrumentation overhead score (dependent on total budget etc.). We are still finalizing the scoring methodology.

6.5.5 Automated Reactive Scores

A statistical record is maintained for each predicate, and the overall success of execution is captured by the error log. We assume worker-thread model, where we are able to associate the success/failure of the transaction by associating process-ids and error log transaction ids. The instrumentation cost for each instrumentation profile can be as follows.

$$\sum_{i=1}^{i=n} x_i = InstrumentationScore(x) * StatisticalScore(x) \tag{6.2}$$

Each predicate is given a total score based on the following parameters:

- **Statistical Importance Score:** The statistical importance score defines the importance of each predicate as an indicator for isolating the bug.The main idea is derived from statistical debugging work done by Liblit et Al
- **Instrumentation Overhead Score:** Adaptive score keeping track of counters of each predicate. Can be used as a weighing mechanism for figuring out the total cost.

6.5.6 Feasibility

We also created a simulator in C/C++ which can buffer size, service times, overhead and incoming request rate to emulate the queuing model. The tool shows buffer overflow, and time taken to reach the overflows.

Our initial investigation has shown that the debug container can sustain significant spikes in overhead without overflowing the production container, as long as the production container is under-capacity. In particular we saw a linear increase in the buffer usage for the debug-container once the workload to the production container reaches it's maximum threshold capacity. This concurred with our earlier assumptions, as a sustained spike in the production container (such that the spike is more than the maximum threshold), would not allow the debug container to catch up. We verified these observations both in our *Parikshan* prototype, as well as the simulation results shown in Appendix **??**. One of the tests in our optimized linux pipe based buffer network proxy, made us realize that for smaller budgets, it is difficult to observe the increase in the buffer usage before a spike leads to an overflow. While the simulation gives us fine grained control in our observations, our micro-evaluation on some real-world software made us realize that control in reality can be much more coarse-grained.

We are currently in the process of making an adaptive sampling instrumentation mechanism. For this we plan to use tools like `iProbe`, systemtap [Prasad *et al.*, 2005] and dyninst [Buck and Hollingsworth, 2000]. These have been previously used in several large-scale instrumentation projects and demonstrated to be effective with low overheads. We will also derive inspiration for our model from previous statistical debugging approaches [Song and Lu, 2014]. These have been demonstrated to be effective in resolving several real-world bugs. This will be explored as part of the thesis.

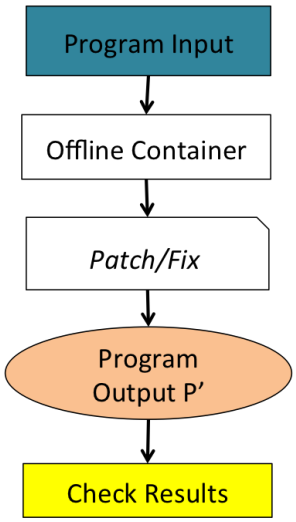


Figure 6.7: Offline Debugging

Figure 6.8: Debugging strategies for offline debugging

6.6 Active Debugging

In Figure 6.7, we show the traditional mechanism of testing or validating patch/fixes in an application. In offline environments, developers apply patches and run the relevant inputs to verify that the patch works correctly. This is an interactive process, which allows one to verify the result and corrections before applying it to the production system. Several cycles of this process is required, which may be followed by staged testing to ensure correctness before applying the update to the production.

active debugging (see figure 6.9) allows debuggers to apply fixes, modify binaries and apply hotpatches to applications. The main idea is to do a fork/exec, or parallel execution of an unmodified

application. The unmodified binary continues execution without any change in the input. The debug-container should ideally mimic the behavior of the production, so as to allow for forward progress in the application as the debug-container will receive the same input as production. The target process will be forked at the call of the testing function, the forked process can then be tested, the input can be transformed, or alternatively the same input can be used to validate any test-condition. At the end of the execution the test-process output can be checked and killed. The advantage of this technique is that any tests/fixes can be validated in the run-time environment itself. This reduces the time to fix and resolve the error. The tests and fixes should have a local impact and should not be allowed to continue

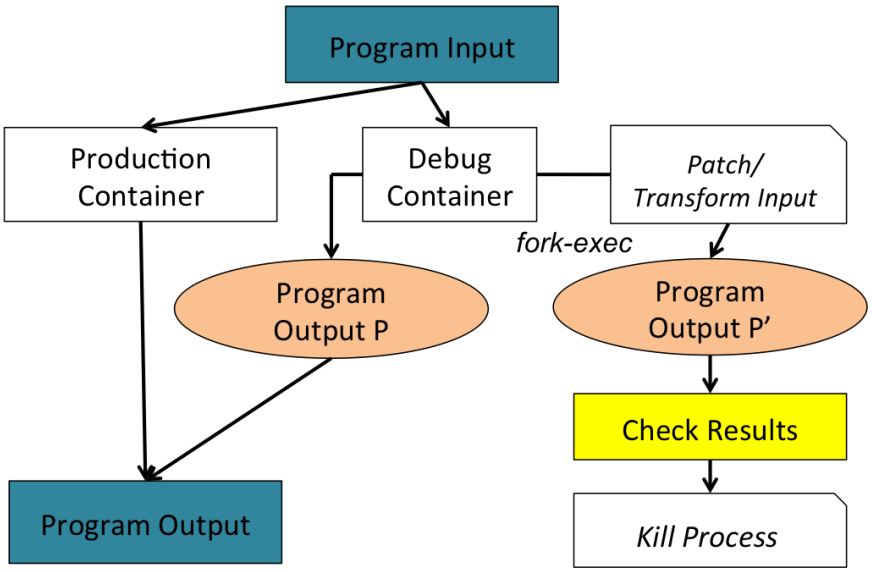


Figure 6.9: Active Debugging

Figure 6.10: Debugging Strategies for Active Debugging

For Java programs, since there is no fork, we can utilize a JNI call to a simple native C program which executes the fork. Performing a fork creates a copy-on-write version of the original process, so that the process running the unit test has its own writable memory area and cannot affect the in-process memory of the original. Once the test is invoked, the application can continue its normal execution, while the unit test runs in the other process. Note that the application and the unit test run in parallel in two processes; the test does not block normal operation of the application after the fork is performed.

The fork-exec design of test-isolation ensures that the "in-process" memory of the process execution is effectively isolated. The production/debug containers are completely isolated hence the test does not impact the production in any way. To ensure further isolation, we propose to allow the test fork to only call wrapper libraries which allow write operations in a cloned cow filesystem. This can be done using a COW supported filesystem with cloning functionality which are supported in ZFS and BTRFS. For instance BTRFS provides a clone operation that atomically creates a copy-on-write snapshot of a file. By cloning the file system does not create a new link pointing to an existing inode; instead it creates a new inode that initially shares the same disk blocks with the original file. As a result cloning works only within the boundaries of the same BTRFS file system, and modifications to any of the cloned files are not visible to the original file and vice versa. This will ofcourse mean that we will constrain the debug/production environment to the File System of our choice. We further propose that all test-cases in the debug-container share the test file system.

6.7 Summary

In this chapter we presented se

Part IV

Related Work

Chapter 7

Related Work

7.1 Related Work for Parikshan

7.1.1 Record and Replay Systems:

Record and Replay [Altekar and Stoica, 2009; Dunlap *et al.*, 2002; Guo *et al.*, 2008; Geels *et al.*, 2007; Veeraraghavan *et al.*, 2012] has been an active area of research in the academic community for several years. These systems offer highly faithful re-execution in lieu of performance overhead. For instance, ODR [Altekar and Stoica, 2009] reports 1.6x, and Aftersight [Chow *et al.*, 2008] reports 5% overhead, although with much higher worst-case overheads. *Parikshan* avoids run-time overhead, but its cloning suspend time may be viewed as an amortized cost in comparison to the overhead in record-replay systems.

Recent approaches in record and replay have been extended to mobile softwares [Hu *et al.*, 2015; Qin *et al.*, 2016], and browsers [Chasins *et al.*, 2015].

7.1.2 Decoupled or Online Analysis

Broadly we categorize decoupled analysis as work where parallel execution similar to *Parikshan* has been employed to gather execution insights. For instance, among record and replay systems, the work most closely related to ours is Aftersight [Chow *et al.*, 2008]. Aftersight records a production system and replays it concurrently in a parallel VM. While both Aftersight and *Parikshan* allow debuggers an almost real-time diagnosis facility, Aftersight suffers from recording overhead in the

production VM. Additionally, it needs the diagnosis VM to either catch up with the production VM, which further slows down the application, or to allow it to proceed with divergence. The average slow-down in Aftersight is 5% and can balloon upto 31% to 2.6x for worst-case scenario.

Another recent paper called, VARAN [Hosek and Cadar, 2015] is an N-version execution monitor that maintains replicas of an existing app, while checking for divergence. *Parikshan*'s debug containers are effectively replicas: however, while VARAN replicates applications at the system call level, *Parikshan*'s lower overhead mechanism does not impact the performance of the master (production) app. Unlike lower-level replay based systems, *Parikshan* tolerates a greater amount of divergence from the original application: i.e., the replica may continue to run even if the analysis slightly modifies it.

7.1.3 Real-Time techniques:

This is a category of approaches which attempt to do real-time diagnosis. Chaos Monkey [Bennett and Tseitlin, 2012] uses fault injection in real production systems to do fault tolerance testing. It randomly injects time-outs, resource hogs etc. in production systems. This allows Netflix to test the robustness of their system at scale, and avoid large-scale system crashes. Another approach called AB Testing [Eisenberg and Quarto-vonTivadar, 2009] probabilistically tests updates or beta releases on some percentage of users, while letting the majority of the application users work on the original system. AB Testing allows the developer to understand user-response to any new additions to the software, while most users get the same software. Unlike *Parikshan*, these approaches are restricted to software testing and directly impact the user.

7.1.4 Live Migration & Cloning

Live migration of virtual machines facilitates fault management, load balancing, and low-level system maintenance for the administrator. Most existing approaches use a *pre-copy* approach that copies the memory state over several iterations, and then copies the process state. This includes hypervisors such as VMWare [Nelson *et al.*, 2005], Xen [Clark *et al.*, 2005], and KVM [Kivity *et al.*, 2007]. VM Cloning, on the other hand, is usually done offline by taking a snapshot of a suspended/ shutdown VM and restarting it on another machine. Cloning is helpful for scaling out applications, which use multiple instances of the same server. There has also been limited work towards live cloning. For

example Sun et al. [Sun *et al.*, 2009] use copy-on-write mechanisms, to create a duplicate of the target VM without shutting it down. Similarly, another approach [Gebhart and Bozak, 2009] uses live-cloning to do cluster-expansion of systems. However, unlike *Parikshan*, both these approaches starts a VM with a new network identity and may require re-configuration of the duplicate node.

7.1.5 Large Scale Software Debugging

Multi-tier production systems are often deployed in a number of machines/containers in scalable cloud infrastructure, and have active monitoring and analysis. In the past few years several products are used for live analytics [Enterprises, 2012; Barham *et al.*, 2004; Tak *et al.*, 2009], which are able to give

7.1.6 Software Programming Paradigms

7.1.7 Virtualization

7.2 Related Work for iProbe

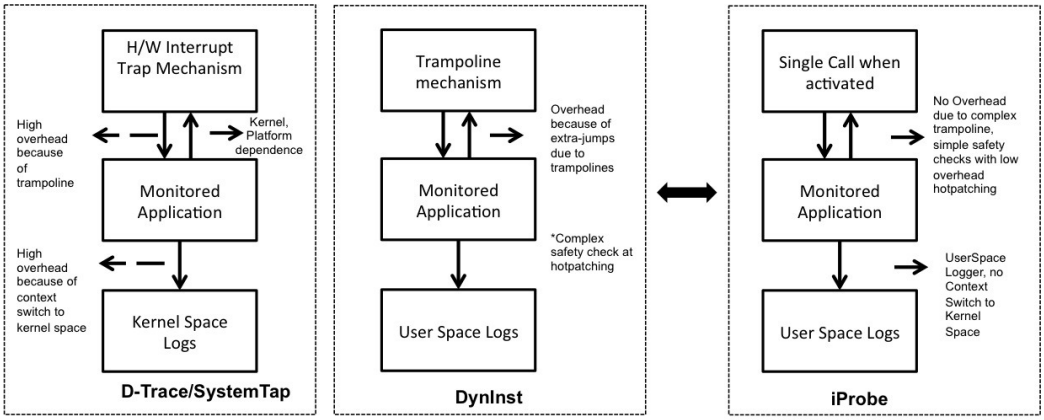


Figure 7.1: Advantages of iProbe over existing monitoring frameworks DTrace/SystemTap and DynInst

Source Code or Compiler Instrumentation Mechanisms: Source code instrumentation is one of the most widely available mechanisms for monitoring. In essence, users can insert debug statements with runtime flags to dump and inspect program status with varying verbosity levels. The log4j [Gupta,

2003] and log4c [log,] frameworks are commonly used libraries to perform program tracing in many open source projects in the source code level. Additionally compilers have several inbuilt profilers which can be used along with tools such as gprof and jprof to gather statistics about program execution. While source code techniques allow very light weight instrumentation, by design they are static and can only be changed at the start of application execution. iProbe on the other hand offers run-time instrumentation that allows dynamic decisions on tracing with comparable overhead.

Run-time Instrumentation Mechanisms: There are several kernel level tracing tools such as DTrace, LTTng, SystemTap [McDougall *et al.*, 2006; Desnoyers and Dagenais, 2006; Prasad *et al.*, 2005] developed by researchers over the years. iProbe differs from these approaches mainly in two ways: Firstly, all of these approaches use a technique similar to software interrupt to switch to kernel space and generate a log event by overwriting the target instructions. They then execute the instrumentation code, and either generate a trampoline mechanism or re-execute the overwritten target instructions and then jump back to the subsequent instructions. As shown in Figure.7.1 this introduces context-switches between user-space and the kernel, causing needless overhead. iProbe avoids this overhead by having a completely user-space based design. Secondly, all these approaches require to perform complex checks for correctness which can cause unnecessary overhead at both hotpatching, and when running an instrumented binary.

Fay [Erlingsson *et al.*, 2012] is a platform-dependent approach which uses the empty spaces at the start of the functions available in Windows binaries for instrumentation. To ensure the capture of the entry and exit of functions, Fay calls the target function within its instrumentation thereby introducing an extra stack frame for each target instrumentation. This operation is similar to a mini-trampoline and hence incurs an overhead. Fay logs function execution in the kernel space and hence also has a context-switch overhead. iProbe avoids such overhead by introducing markers at the beginning and end of each function using a

Another well known tool is DynInst[Buck and Hollingsworth, 2000]. This tool provides a rich dynamic instrumentation capability and has pure back box solution towards instrumentation of any application. However, as shown in Figure.7.1 it is also based on traditional trampoline mechanisms, and induces a high overhead because of unnecessary jump instructions. Additionally it can have higher overhead because of complex security checks. Other similar trampoline based tools like kaho and pannus[pan, ; Yamato *et al.*, 2009] have also been proposed, but they focus more towards

patching binaries to add *fixes* to correct a bug.

Debuggers: Instrumentation is a commonly used technique in debugging. Many debuggers such as gdb [Stallman *et al.*, 2002] and Eclipse have breakpoints and watchpoints which can stop the execution of programs and inspect program conditions. These features are based on various techniques including `ptrace` and hardware debugging support (single step mode and debug registers). While they provide such powerful instrumentation capabilities, there are in general not adequate for beyond the debugging purposes due to overwhelming overhead.

Dynamic Translation Tools: Software engineering communities have been using dynamic translation tools such as Pin [Luk *et al.*, 2005] and Valgrind [Nethercote and Seward, 2007] to inspect program characteristics. These tools dynamically translate program code before execution and allow users to insert custom instrumentation code flexibly. They are capable to instrument non-debug binaries and provide versatile tools such as memory checkers and program profilers. However, similar to debuggers, they are generally considered as debugging tools and their overhead is significantly higher than runtime tracers.

Part V

Conclusions

Chapter 8

Conclusions

8.1 Contributions

In this thesis, we have presented

The main contributions of this thesis are as follows:

- We are the first to present a new technique

8.2 Future Work

There are a number of interesting future work possibilities, both in the short term and further into the future.

In the future, we will explore:

- **Applications**: we aim to apply our system to real-time intrusion detection and statistical debugging.
- **Analysis**: we wish to define "real-time" data analysis techniques for traces and instrumentation done in *Parikshan*.
- **Optimize Live Cloning**: We plan to reduce the suspend time of live cloning, by utilizing several recent works in live migration.

8.2.1 Immediate Future Work

- **Improve live cloning performance:** The current protoype of livecloning is based on container virtualization and previous efforts in live migration in OpenVZ [Kolyshkin, 2006]. However, our implementation is limited by the performance of the current level of performance of current live migration efforts. Live migration is a nascent research topic in user-space container level virtualization, however there has been significant progress in live-migration in virtual machine virtualization.

  One key limitation in the current approach is that it has been built using *rsync* [Tridgell *et al.*, 1996] functionality. This is much slower than current state-of-the-art techniques in full VM virtualization, which rely on network file systems to synchronization images asynchronously [Palevich and Taillefer, 2008]. Other optimizations include post-copy migration [Hines *et al.*, 2009] which does lazy migration - the idea is to do on-demand transfer of pages by triggering a network page fault. This reduces the time that the target container is suspended, and ensures real-time performance. The current implementation in *Parikshan* uses the traditional *pre-copy migration* [Clark *et al.*, 2005], which iteratively syncs the two images to reduce the suspend time.

  Live cloning can be used in two scenarios, either with a fresh start where the target physical machines do not have a copy of the initial image. However, more commonly once the first live clone has been finished, the target is to reduce the suspend time of subsequent live cloning requests. This is different from live migration scenario's. For instance, future research can focus on specifically on reducing this downtime by keeping track of the "delta" from the point of the detection of divergence. This will reduce the amount of page faults in a post-copy algorithm, and can potentially improve live cloning performance compared to migration.

- **Scaled Mode for live-debugging:** One key limitation of live-debugging is the potential for memory overflow. The period till a buffer overflow happens in the proxy, is called the *debug window*. It is critical for continuous debugging that the *debug window* be as long as possible. The size of this window, depends on the instrumentation overhead, the amount of workload, and the buffer size itself.

  Hence, it may be possible that at times for very heavy instrumentation or workload, the *debug*

window becomes too short to be of practical use. To counter this in our design section we briefly mentioned a scaled mode for live debugging.

- Feedback synchronization

8.2.2 Possibilities for Long Term

- Explore implementation in Virtual Machines

8.3 Conclusion

In this thesis we have explored approaches and frameworks for

Part VI

Bibliography

Bibliography

[Allspaw J., 2009] Hammond P. Allspaw J. 10+ deploys per day: Dev and ops cooperation at flickr. OReilly Velocity Web Performance and Operations Conference, 2009.

[Altekar and Stoica, 2009] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206. ACM, 2009.

[Amazon, 2010] EC Amazon. Amazon elastic compute cloud (amazon ec2). *Amazon Elastic Compute Cloud (Amazon EC2)*, 2010.

[Arumuga Nainar and Liblit, 2010] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 255–264. ACM, 2010.

[Axboe, 2008] Jens Axboe. Fio-flexible io tester. *uRL: http://freecode. com/projects/fio*, 2008.

[Barham *et al.*, 2004] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[Beck, 2000] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[Bennett and Tseitlin, 2012] C Bennett and A Tseitlin. Netflix: Chaos Monkey released into the wild. netflix tech blog, 2012.

[Bernstein, 2014] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[Blackburn *et al.*, 2006] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[Boettiger, 2015] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[Borthakur, 2008] Dhruba Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, page 39, 2008.

[Bratus *et al.*, 2010] S. Bratus, J. Oakley, A. Ramaswamy, S.W. Smith, and M.E. Locasto. Katana: Towards patching as a runtime part of the compiler-linker-loader toolchain. *International Journal of Secure Software Engineering (IJSSE)*, 1(3):1–17, 2010.

[Buck and Hollingsworth, 2000] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000.

[Carlson, 2013] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013.

[cde,] Calling conventions for different operating systems.

[Chasins *et al.*, 2015] Sarah Chasins, Shaon Barman, Rastislav Bodik, and Sumit Gulwani. Browser record and replay as a building block for end-user web automation tools. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, pages 179–182, New York, NY, USA, 2015. ACM.

[Chilimbi *et al.*, 2009] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44. IEEE Computer Society, 2009.

[Chow *et al.*, 2008] Jim Chow, Tal Garfinkel, and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, 2008.

[Clark *et al.*, 2005] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[Cooper, 1972] Robert B Cooper. Introduction to queueing theory. 1972.

[D'Anjou, 2005] Jim D'Anjou. *The Java developer's guide to Eclipse*. Addison-Wesley Professional, 2005.

[Deshpande and Keahey, 2016] Umesh Deshpande and Kate Keahey. Traffic-sensitive live migration of virtual machines. *Future Generation Computer Systems*, 2016.

[Desnoyers and Dagenais, 2006] M. Desnoyers and M.R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, pages 209–224. Citeseer, 2006.

[DockerHub,] Build ship and run anywhere. https://www.hub.docker.com/.

[Dunlap *et al.*, 2002] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[Eigler and Hat, 2006] Frank Ch Eigler and Red Hat. Problem solving with systemtap. In *Proc. of the Ottawa Linux Symposium*, pages 261–268. Citeseer, 2006.

[Eisenberg and Quarto-vonTivadar, 2009] Bryan Eisenberg and John Quarto-vonTivadar. *Always be testing: The complete guide to Google website optimizer*. John Wiley & Sons, 2009.

[ElasticSearch,] ElasticSearch. Elasticsearch.

[Enterprises, 2012] Nagios Enterprises. Nagios, 2012.

[Erlingsson *et al.*, 2012] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst.*, 30(4):13:1–13:35, November 2012.

[Felter *et al.*, 2015] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[Flanagan and Godefroid, 2005] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[Furman, 2014] Mark Furman. *OpenVZ Essentials*. Packt Publishing Ltd, 2014.

[Ganai *et al.*, 2011] Malay K. Ganai, Nipun Arora, Chao Wang, Aarti Gupta, and Gogul Balakrishnan. Best: A symbolic testing tool for predicting multi-threaded program failures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 596–599, Washington, DC, USA, 2011. IEEE Computer Society.

[gco,] Google Compute. https://cloud.google.com/compute/.

[Gebhart and Bozak, 2009] A. Gebhart and E. Bozak. Dynamic cluster expansion through virtualization-based live cloning, September 10 2009. US Patent App. 12/044,888.

[Geels *et al.*, 2007] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, volume 7, pages 285–298, 2007.

[Guo *et al.*, 2008] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 193–208. USENIX Association, 2008.

[Gupta, 2003] S. Gupta. *Logging in Java with the JDK 1.4 Logging API and Apache log4j*. Apress, 2003.

[Henning, 2006] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[Hines *et al.*, 2009] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43(3):14–26, 2009.

[Hosek and Cadar, 2015] Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 339–353, New York, NY, USA, 2015. ACM.

[Hu *et al.*, 2015] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 349–366, New York, NY, USA, 2015. ACM.

[Intel, 2011] Intel. Vtune amplifier. http://www.intel.com/software/products/vtune, 2011.

[Jin *et al.*, 2010] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *ACM Sigplan Notices*, volume 45, pages 241–255. ACM, 2010.

[Jin *et al.*, 2012] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[Jovic *et al.*, 2011] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: performance bug detection in the wild. In *OOPSLA*, 2011.

[Kasikci *et al.*, 2015] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 344–360, New York, NY, USA, 2015. ACM.

[Kendall, 1953] David G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *Ann. Math. Statist.*, 24(3):338–354, 09 1953.

[Kivity *et al.*, 2007] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[Kolyshkin, 2006] Kirill Kolyshkin. Virtualization in linux. *White paper, OpenVZ*, 3:39, 2006.

[Laadan *et al.*, 2010] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, number 1, pages 155–166. ACM, 2010.

[Lakshman and Malik, 2010] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[Liblit *et al.*, 2005] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005.

[Liblit, 2004] Benjamin Robert Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, December 2004.

[lin,] Linux ipc pipes. http://man7.org/linux/man-pages/man7/pipe.7.html.

[Linux Manual Ptrace,] Ptrace:linux process trace: http://linux.die.net/man/2/ptrace.

[liv,] Livepatch: http://ukai.jp/software/livepatch/.

[log,] log4c: Logging for c library.

[loggly,] loggly. Loggly.

[Lou *et al.*, 2013] Jian-Guang Lou, Qingwei Lin, Rui Ding, Qiang Fu, Dongmei Zhang, and Tao Xie. Software analytics for incident management of online services: An experience report. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 475–485. IEEE, 2013.

[Lu *et al.*, 2005] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[Luk *et al.*, 2005] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[Martin, 2003] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[Massie *et al.*, 2004] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[Matulis, 2009] Peter Matulis. Centralised logging with rsyslog. *Canonical Technical White Paper*, 2009.

[McDougall *et al.*, 2006] Richard McDougall, Jim Mauro, and Brendan Gregg. *Solaris (TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall PTR, 2006.

[McGrath, 2009] R. McGrath. Utrace. *Linux Foundation Collaboration Summit*, 2009.

[Merkel, 2014] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[Mirkin *et al.*, 2008] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, pages 85–92, 2008.

[Mosberger and Jin, 1998] David Mosberger and Tai Jin. httperfa tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[Mucci *et al.*, 1999] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[Murphy *et al.*, 2009] Christian Murphy, Gail Kaiser, Ian Vo, and Matt Chu. Quality assurance of software applications using the in vivo testing approach. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 111–120, Washington, DC, USA, 2009. IEEE Computer Society.

[Mussler *et al.*, 2011] Jan Mussler, Daniel Lorenz, and Felix Wolf. Reducing the overhead of direct application instrumentation using prior static analysis. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 65–76, Berlin, Heidelberg, 2011. Springer-Verlag.

[MySQL, 2001] AB MySQL. Mysql, 2001.

[nat,] NAT: Network address translation. http://en.wikipedia.org/wiki/Network_address_translation.

[Nelson *et al.*, 2005] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 391–394, 2005.

[net,] Network namespaces. https://lwn.net/Articles/580893/.

[Nethercote and Seward, 2007] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07*, 2007.

[Newman, 2015] Sam Newman. *Building Microservices*. " O'Reilly Media, Inc.", 2015.

[OpenVZ Ploop,] Ploop: Containers in a file. http://openvz.org/Ploop.

[OReilly, DevOps,] What is devops? http://www.radar.oreilly.com.

[Palevich and Taillefer, 2008] John H Palevich and Martin Taillefer. Network file system, October 21 2008. US Patent 7,441,012.

[pan,] Pannus: A hot patching tool, http://pannus.sourceforge.net/.

[Park and Buch, 2004] Insung Park and R Buch. Event tracing for windows: Best practices. In *Int. CMG Conference*, pages 565–574, 2004.

[Patil *et al.*, 2010] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[pet,] Petstore a sample java platform, enterprise edition reference application. http://www.oracle.com/technetwork/java/petstore1-1-2-136742.html.

[Petersen *et al.*, 2009] Kai Petersen, Claes Wohlin, and Dejan Baca. The waterfall model in large-scale development. In *International Conference on Product-Focused Software Process Improvement*, pages 386–400. Springer, 2009.

[Prasad *et al.*, 2005] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Ottawa Linux Symposium (OLS)*, 2005.

[Qin *et al.*, 2016] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 571–582, New York, NY, USA, 2016. ACM.

[Ravindranath *et al.*, 2012] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 107–120, 2012.

[Sambasivan *et al.*, 2011] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, 2011.

[Song and Lu, 2014] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *ACM SIGPLAN Notices*, volume 49, pages 561–578. ACM, 2014.

[splunk,] splunk. Splunk.

[Stallman *et al.*, 2002] R.M. Stallman, R. Pesch, and S. Shebs. Debugging with gdb: The gnu source-level debugger for gdb version 5.1. 1. 2002.

[Sun *et al.*, 2009] Yifeng Sun, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, Binbin Zhang, Haogang Chen, and Xiaoming Li. Fast live cloning of virtual machine based on xen. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, HPCC '09, pages 392–399, Washington, DC, USA, 2009. IEEE Computer Society.

[Svärd *et al.*, 2015] Petter Svärd, Benoit Hudzia, Steve Walsh, Johan Tordsson, and Erik Elmroth. Principles and performance characteristics of algorithms for live vm migration. *ACM SIGOPS Operating Systems Review*, 49(1):142–155, 2015.

[Tak *et al.*, 2009] Byung-Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N Chang. vPath: Precise discovery of request processing paths from black-box observations of thread and network activities. In *USENIX Annual technical conference*, 2009.

[Tanenbaum, 2003] Andrew S. Tanenbaum. *Computer Networks, Fourth Edition*. Prentice Hall PTR, 2003.

[Thomson and Donaldson, 2015] Paul Thomson and Alastair F. Donaldson. The lazy happens-before relation: Better partial-order reduction for systematic concurrency testing. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 259–260, New York, NY, USA, 2015. ACM.

[Tridgell *et al.*, 1996] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. 1996.

[Veeraraghavan *et al.*, 2012] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst.*, 30(1):3:1–3:24, February 2012.

[Wang *et al.*, 2014] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of annual IEEE/ACM international symposium on code generation and optimization*, page 98. ACM, 2014.

[Xavier *et al.*, 2013] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

[Yamato *et al.*, 2009] K. Yamato, T. Abe, and M.L. Corpration. A runtime code modification method for application programs. In *Proceedings of the Ottawa Linux Symposium*, 2009.

[Yuan *et al.*, 2014] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, 2014.

[Zhang *et al.*, 2014] Hui Zhang, Junghwan Rhee, Nipun Arora, Sahan Gamage, Guofei Jiang, Kenji Yoshihira, and Dongyan Xu. CLUE: System trace analytics for cloud service performance diagnosis. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–9. IEEE, 2014.