Graduate School Form 30
(Updated 11/20/2014)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Amiya Kumar Maji

Entitled
Dependability where the Mobile World Meets the Enterprise World

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

SAURABH BAGCHI

ANAND RAGHUNATHAN

ELISA BERTINO

JAN S. RELLERMEYER

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

SAURABH BAGCHI

Approved by Major Professor(s): _____

_____

Approved by: Michael R. Melloch 03/02/2015

Head of the Department Graduate Program Date

DEPENDABILITY WHERE THE MOBILE WORLD MEETS THE ENTERPRISE WORLD

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Amiya K. Maji

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2015

Purdue University

West Lafayette, Indiana

Dedicated to my parents and sister whose unconditional love
is the source of my inspiration.

ACKNOWLEDGMENTS

This dissertation has been shaped by the direct and indirect involvement of many persons over the years. I shall take this opportunity to express my gratitude to them. Firstly, my sincere thanks to my advisor Prof. Saurabh Bagchi whose support and guidance is at the core of this thesis. His incredible patience and time-management skills, not to mention his wide array of knowledge, are qualities I aspire to emulate. A big Thank You for your encouragements, specially when the chips were down. Thanks also go to my doctoral committee members—Prof. Anand Raghunathan, Prof. Elisa Bertino, and Dr. Jan S. Rellermeyer—whose insightful questions and suggestions helped uncover several subtle nuances in this dissertation.

I have been privileged to collaborate with few renowned researchers during the course of the dissertation. My mentor during internship at IBM Research, Akshat Verma, deserves special mention for teaching me the importance of preliminary experiments in systems research and how they simplify the process of finding solutions. I would also like to express my gratitude to Dr. Jan S. Rellermeyer from IBM Research, Austin, for his active collaboration during preparation of our DSN 2012 paper. To my other collaborators and fellow graduate students—Subrata Mitra, Dr. Fahad Arshad, Dr. Bowen Zhou, Kangli Hao, and Dr. Salmin Sultana—whose contributions often made the difference between meeting a deadline or not—thank you for your help.

My friends, both within and outside Purdue, played an important role in helping reduce the stress of graduate school. Thank you for your encouragements, good wishes, and the moments of fun which I will always cherish. Last but not the least, my heartfelt gratitude to my family members for their incredible support during these years. The love and kindness I have received from them is beyond words. To my parents, sister, grandparents, and other family members—I love you and dedicate this dissertation to you.

TABLE OF CONTENTS

Page

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Maji, Amiya K. Ph.D., Purdue University, May 2015. Dependability where the Mobile World Meets the Enterprise World. Major Professor: Saurabh Bagchi.

As we move toward increasingly larger scales of computing, complexity of systems and networks has increased manifold leading to massive failures of cloud providers (Amazon Cloudfront, November 2014) and geographically localized outages of cellular services (T-Mobile, June 2014). In this dissertation, we investigate the dependability aspects of two of the most prevalent computing platforms today, namely, smartphones and cloud computing. These two seemingly disparate platforms are part of a cohesive story—they interact to provide end-to-end services which are increasingly being delivered over mobile platforms, examples being iCloud, Google Drive and their smartphone counterparts iPhone and Android.

In one of the early work on characterizing failures in dominant mobile OSes, we analyzed bug repositories of Android and Symbian and found similarities in their failure modes [ISSRE2010]. We also presented a classification of root causes and quantified the impact of ease of customizing the smartphones on system reliability. Our evaluation of Inter-Component Communication in Android [DSN2012] show an alarming number of exception handling errors where a phone may be crashed by passing it malformed component invocation messages, even from unprivileged applications. In this work, we also suggest language extensions that can mitigate these problems.

Mobile applications today are increasingly being used to interact with enterprise-class web services commonly hosted in virtualized environments. Virutalization suffers from the problem of imperfect performance isolation where contention for low-level hardware resources can impact application performance. Through a set of rigorous experiments in a private cloud testbed and in EC2, we show that interference

induced performance degradation is a reality. Our experiments have also shown that optimal configuration settings for web servers change during such phases of interference. Based on this observation, we design and implement the IC^2 engine which can mitigate effects of interference by reconfiguring web server parameters [MW2014]. We further improve IC^2 by incorporating it into a two-level configuration engine, named ICE, for managing web server clusters [ICAC2015]. Our evaluations show that, compared to an interference agnostic configuration, IC^2 can improve response time of web servers by upto 40%, while ICE can improve response time by upto 94% during phases of interference.

1. INTRODUCTION

The last decade has seen two significant evolutions in the field of computing. First, there is an increased mobility of computing devices. Starting from the days of static desktop-based computers we have now entered the era of mobile computing where smartphones and tablets are more popular than any other computing devices. This can be easily verified by the phenomenal growth of the smartphone market. According to surveys by Gartner [1], number of mobile phone sales from 2013 onwards far exceed computer sales. The overall market share of PCs is expected to decline further resulting in 87% of connected devices being tablets or smartphones by 2017 [2]. The second evolution is the tremendous increase in volume of data generated and transferred over the Internet. Social networks and media sharing websites are the key enablers of this evolution. To support this unforeseen increase in scale of computing in a cost-efficient manner, organizations have adopted cloud-computing where physical resources are shared between multiple users in a virtualized environment leading to higher utilization.

While smartphones allow users to connect to the Internet in a nearly uninterrupted fashion, they also open up the possibility of a new range of applications with innovative user interfaces and input modules. On the other hand, cloud computing allows service providers to scale up to larger numbers of users many of whom connect via mobile devices. Today, mobile computing and cloud computing form a symbiotic ecosystem where each augments the functionality and utilization of the other. Consider the example of a person posting her location to a social networking website through her smartphone. Here, the smartphone with help of its gps sensor acts as the key enabler for this information sharing. On the other end of the spectrum, cloud enables a multitude of functionality on the smartphone—a simple example being online search.

We argue that smartphones and the cloud are not disjoint computing platforms but form the basis for a unified computing platform. Both of them are responsible for executing parts of the use-cases described above. From an end-user's point of view, both of these must operate reliably to provide uninterrupted services. Hence, improving dependability of cloud-based applictions and their mobile counterparts are equally important. Let's take the example of a user purchasing books (or games) from her smartphone (a vendor-specific example can be that of Android's Google Play app and the cloud-based Google-play store). A complex transaction of this nature involves successful completion of the purchase both at the mobile and the online store, followed by the book being delivered to the user's phone. Failure in any of the steps may result in user inconvenience or monetary loss or both. Unfortunately, reports show that both mobile and cloud platforms are very failure prone. An example of the Google Play app crash can be found in [3], while its cloud counterpart, the Google Play services crashed in January, 2014 [4].

The landscape in mobile and cloud computing is changing rapidly over time thanks to contributions from both researchers and developers. This can be verified by the fact that Android has released 11 revisions (3 major) to its SDK Tools in 2014 alone (similar inferences may be drawn from the improvements in both open-source and proprietary virtualization solutions). A short time-to-market for softwares is often characterized by poorer testing and larger number of bugs. We believe that the recent advances in mobile and cloud computing, and current practices for mobile app development leave significant scope for dependability research.

In this dissertation, we look at the dependability aspects of both these platforms. First, we look at the dependability of smartphone applications and their inter-component communication (ICC). Next we look at performance anomalies (of which service outage is a small subset) in cloud-based services. Below, we present key motivations for our work.

1.1 Motivations

Characterizing Mobile Failures. The release of iPhone in 2007 started a new revolution in mobile computing, however, it was not until 2008 (when Android was open-sourced) that researchers had freely available source code and failure reports to analyze their reliability. Several researchers have studied failure characteristics of popular operating systems like Windows and Linux [5,6] in the past. But evaluations of mobile operating systems was rarely seen before release of Android. A notable exception is the study of failures in Symbian OS-based smart phones [7]. However, since Symbian was not open source at the time of the analysis, the authors of this earlier study were limited in what they could do. They used failure logs from 25 smart phones being used by volunteers participating in the study. Our work [ISSRE2010] extends this prior work by significant proportions. Our study of failure characteristics in smartphones is the first of its kind in that it looks into *open-source* mobile OSes and classifies their failures based on a much larger number and greater diversity of failure reports.

IPC Robustness. Several example failures that showed propagation of errors across different components in Android [ISSRE2010] motivated us further to experimentally evaluate dependability of smartphones. In case of smartphones, sources of inputs to applications can be significantly diverse—these include touchscreen, keyboard, radio, microphone, sensors, untrusted third-party applications, or data from one of many network drivers—and therefore *it has great potential for receiving unexpected data*. Given the unorthodox techniques people employ to bypass password locks on their smartphones [8], receipt of *unexpected* data is not a rarity. Our second experiment [DSN2012] was designed to see how well Android reacts to unexpected data, and more specifically to test its Inter Process Communication primitives. We define robustness as the ability to handle unexpected data gracefully, therefore, lack of robustness would imply an application crashing in response to an IPC message. In

the context of Android applications, these crashes manifest as uncaught exceptions in the stack trace.

Dependability of Cloud Services. While continuing our research on robustness of smartphones, we realized that many of the popular mobile apps are supported by cloud-based services, examples being, Facebook, Pandora Radio, Angry Birds, Skype etc. (some of which showed exception handling errors in our earlier experiments). At the same time, several incidents of cloud service outage [9, 10] motivated us to look into the dependability aspects of cloud services. Today's infrastructure clouds (IaaS) are supported by some form of virtualization, which allows multiple users to run their applications in containers called virtual machines (VM). Although one of the goals of virtualization is to support isolation among multiple VMs, existing hardware and software virtualization mechanisms do not provide perfect performance isolation. For example, x86-based platforms do not allow for reservation of cache usage, memory bandwidth, or I/O bandwidth. Lots of past work has demonstrated the extent of this interference for production virtualization environments, such as, those using Xen or KVM [11–13]. Existing work on mitigating interference require intrusive modifications to hypervisors [14–17], which is infeasible in public clouds (such as EC2) due to limited permissions of guest VMs. We were, therefore, encouraged to find innovative solutions that would not require access to hypervisor.

Interference Mitigation using Middleware Reconfiguration. During our experiments on evaluating impact of interference, we found that web service applications have a large number of parameters that have direct impact on performance. The optimal values for these parameters change significantly during interference, therefore, we can mitigate interference by changing these parameter values when such an event is detected. We use this observation to design and implement an autonomous reconfiguration engine for web services middleware (called IC^2, or Interference-aware Cloud application Configuration). We further observed that in web server clusters, interference may be mitigated by moving web requests away from the impacted web server VM (WS-VM) to less loaded servers. This not only improves response times

of the requests that are directed to other servers, but also the impacted WS processes fewer requests efficiently. This observation led to the design of a two-level configuration engine for web server clusters (called ICE or Integrated Configuration Engine)—the first level reconfiguration happens at the load-balancer in front of the cluster, while the second level reconfiguration happens in individual WS VMs suffering from interference (similar to IC^2). Both IC^2 and ICE helped us in mitigating interference effectively without modification to the hypervisor and, therefore, present practical solutions for public clouds.

1.2 Objectives

In this dissertation, our objective is to improve the dependability of smartphones and the cloud-based services that interact with them. To this end, we characterize the failures in these platforms and develop tools and techniques to detect and mitigate them. More specifically, we seek to answer the following questions:

1.2.1 Smartphones

1. What are the applications and libraries that fail frequently in Android and Symbian? Are there any similarity in failure manifestations of these platforms?
2. What are the types of code fixes that are applied frequently in Android?
3. Does customizability of smartphones impact their reliability?
4. How robust are Android's ICC primitives? Can the Android runtime contain exceptions within an application? How well does an Android component behave in the presence of a semi-valid or random Intent?
5. How can we refine the implementation of Intents so that input validation can be improved?

1.2.2 Cloud Services

1. Does interference cause performance degradation in public clouds such as Amazon EC2? If so, how often and how much degradation is observed?

2. Does interference change the optimal configuration of an application? What are the challenges associated with finding the optimal parameter values?

3. How can we detect when an application is suffering from interference? How can we automatically reconfigure an application to mitigate performance interference?

4. Does interference show performance degradation in load-balanced web server clusters? How can we reconfigure such clusters quickly and efficiently to deal with interference?

5. How much performance improvement can be seen by reconfiguring middleware parameters?

1.3 Approach

Smartphones. To answer the first three questions in 1.2.1, we collected a large volume of failure data from Android and Symbian bug repositories. Based on the collected samples (628 bugs in Android and 153 bugs in Symbian) we looked into the manifestation of failures in different modules in Android and Symbian, and identified which modules are reported to cause the greatest unreliability. We further collected a sample of 233 bug fixes in Android spread over 29 projects and classified the fixes based on the type of corrections needed in code. We also measured the complexities of these projects (in terms of lines of code and cyclomatic complexity) and compared their bug densities found in our sample set.

To answer the last two questions on smartphones, we developed our Android robustness testing tool, JarJarBinks (in remembrance of the Gungan warrior of Star

Wars fame, whose unusual accent created significant problems for the Droid). JarJarBinks includes four Intent generation modules—semi-valid, blank, random, and random with extras, and the ability to automatically send a large number of Intents to all the components. JarJarBinks runs as a user level process, it does not require knowledge of source codes of the tested components, and can be easily configured for the robustness testing on any Android device. During our experiments we sent more than 6 million Intents to 800+ application components across 3 versions of Android (2.2, 2.3.4, and 4.0) and discovered a significant number of input validation errors. Our most striking finding was the ability to run privileged processes from user level applications (such as JarJarBinks) in Android without requiring the user-level application to be granted any special permission at install time. We also suggested some design-level guidelines for improving robustness of ICC in Android.

Cloud Services. We answer the first two questions in 1.2.2 by an extensive empirical evaluation of the Olio [18] web-application benchmark in EC2 as well as in a private cloud testbed. We run Olio with a variety of configuration settings and under various cache-intensive interferences. The configuration settings we look at are the performance critical parameters in the web application runtime, i.e. Apache and Php runtime. These parameters are related to the parallelism (maximum number of threads) in Apache and Php runtime (`MaxClients`,`PhpMaxChildren`), and their idle timeouts (`KeepaliveTimeout`). We detect interference by using a collection of hardware performance counters (Private Testbed) or a set of application performance metrics (in EC2, where hardware performance counters are not available). A huristic-driven configuration controller (IC^2) is used to predict a good configuration value during interference.

We repeated similar experiments in a load-balanced web-server setup and found interference can degrade response time of websites significantly (question 4 in Section refsec:conf-motiv). To mitigate interference our autonomous configuration engine ICE performs two-level reconfigurations of the web-server cluster. The first level, geared towards agility, is to reconfigure the load balancer in such a manner so that

fewer requests are forwarded to the affected WS VM. We found that this provides the maximum benefit in terms of reduced response time. The second level of reconfiguration, which configures Apache and Php-fpm as described in IC^2, is activated only if the interference lasts for a long time. This prevents the server from incurring unnecessary reconfiguration overheads if the interference is short. We evaluated both IC^2 and ICE by measuring response times of Cloudsuite during periods of interference. We compare these with the response times when IC^2 or ICE are not enabled (baseline). Overall, we found that IC^2 could improve response time by upto 40% and ICE could improve response time by upto 94% compared to baseline.

1.4 Contributions

We summarize our findings and contributions to improve the dependability of smartphones and cloud-based applications as follows:

1.4.1 Dependability of Smartphones

1. We were the first to analyze the failure characteristics of *open-source* mobile OSes based on a large number and diversity of failure samples. We found that the kernels of both these platforms are significantly hardened (only few bugs per million lines of code), however, we need more efforts in improving the middleware. Our analysis of code modifications in Android showed that a large fraction (77%) of the bug fixes needed only small changes in code, as opposed to significant code modification. We also presented a classification of the code fixes.

2. By measuring the number of environment variables in the Android platform across its different versions and comparing it with Linux Kernel, we analyze the customizability of Android. This indicates Android's dependence on a few critical variables, which implies that these environment variables need to be set

with care; else, significant error propagation can occur. We found that both Android and Symbian provide a radically high level of customizability both in building and executing the OS. While this customizability does lead to some loss of reliability, especially in Development tools and Third-party applications, such loss can be mitigated by more rigorous testing.

3. We developed a tool named JarJarBinks for evaluating the robustness of ICC in Android. JarJarBinks runs as a user-level process, does not require knowledge of source-code of applications, and can send semi-valid and random Intents to any application on the same phone (either targeted, i.e. explicit Intent or via mediation of Android runtime, i.e. implicit Intent).

4. Our experiments showed, in general, less than 10% of the components tested crashed; all crashes are caused by unhandled exceptions. Our results suggest that Android has a sizable number of components with unhandled `NullPointer Exceptions` across all versions. The most striking finding that we have is the ability to run privileged processes from user level applications without requiring the user-level application to be granted any special permission at install time. We found three instances, where we could crash the Android runtime from JarJarBinks. Such a crash makes the Android device unusable till it is rebooted. This has huge potential for privilege escalation, denial-of-service, and may even lead to more security vulnerabilities, if an adversary could figure out how to have these malformed (or "fuzzed") Intents be sent out in response to some external message.

5. To improve software design from the point of view of reliability, we found that subtyping combined with Java annotations can be used very effectively to restrict the format and content of an Intent. Through this mechanism, the attack surface of Android can be reduced significantly.

1.4.2 Dependability of Cloud-based Applications

1. We rigorously studied the performance variability of web-based applications in a public cloud environment. In this study, we run the CloudSuite [19] benchmark in Amazon's EC2 for 100 hours over a 5-day period. We then compare the statistics obtained from these runs with sample runs of CloudSuite in a private cloud testbed. We observe that CloudSuite has much longer response time distribution in EC2 (ranging upto 5.5s) than in the local testbed (upto 0.42s only) with identical resource configurations. This validates our hypothesis, that public clouds have high degree of performance uncertainty.

2. We conducted a study to understand if applications can be configured to deal with interference. We observed that an ideal operating configuration for Apache web server depends on the type and degree of interference. Further, parameters in different elements of the software stack depend on each other and the inter-dependency changes with the degree of interference; and finally, the application performance curves with the configuration values are discontinuous in places, making traditional control-theoretic approaches for parameter tuning [20] ineffective. Specifically we found three parameters corresponding to the degree of concurrency and the time to live of existing connections to be particularly significant.

3. We present a *simple, heuristic-driven* configuration manager, IC^2 to reconfigure the application upon interference. IC^2 solves three key challenges for dynamic reconfiguration—first, it presents a machine learning based technique for detecting interference; second, it uses a heuristic-based controller for determining suitable parameter values during periods of interference; and finally, it reduces the cost of reconfiguration of standard Apache distributions by implementing an online reconfiguration option in the Httpd server. A prototype implementation of IC^2 was deployed both in EC2 and our private testbed. The experiments

show that IC^2 can recapture lost response time by upto 29% in EC2 and 40% in our private testbed.

4. We present the design and implementation of a two-level reconfiguration engine for load-balanced web server clusters to deal with interference in cloud. Our solution, called ICE, includes algorithms for: a) detecting interference quickly (primarily cache and memory bandwidth contention), and b) predicting new weights for impacted server to reduce load on them. Our evaluation experiments show that on a combination of Apache+Php+HAProxy middleware, ICE can reduce median response time of web servers by upto 94%. We evaluate ICE for two different load balancer scheduling policies (weighted round-robin, and weighted least-connection) in HAProxy and find that it improves response time across both scheduling policies (upto 94% and 39% respectively). Median interference detection latency was 3s.

5. To evaluate the generalizability of our framework, we also ran some experiments with the Darwin media streaming server running with LVS load balancer. Our results show that reconfiguring server weight in LVS can be used to reduce the inter-frame delay for an affected Darwin VM. We also found that the optimal `num_threads` parameter in Darwin is vastly different with and without interference indicating our WS reconfiguration is also applicable to Darwin.

1.5 Outline

The dissertation is organized as follows. In Chapter 2, we present relevant background material and terminology used in mobile application frameworks. Similarly, Chapter 5 defines various terminologies used in Cloud Services. We also present an overview of web server configurations in this chapter. Chapters 3–4 and Chapters 6–7 are organized in a manner such that each chapter, in combination with relevant background material (Chapter 2 or Chapter 5), can be followed independently. For each of these chapters, we first present the problem statement we address, followed by

our design, implementation, experimental results, and suggestions for improvements. Chapter 3 contains our case study on failures in Android and Symbian, while, Chapter 4 contains our empirical evaluation of robustness of Android ICC. Chapters 6 and 7 present our solutions for mitigating performance interference in cloud under two different setups, one where each web server VM is managed in isolation (Chapter 6) and another where the web server cluster is managed as a whole (Chapter 7). Chapter 8 presents an overview of earlier research related to our work and compare them with the proposed solutions. Finally, we summarize our findings from Chapters 3–7 in Chapter 9 and conclude this dissertation.

2. OVERVIEW OF SMARTPHONES: SYSTEMS AND APPLICATIONS

In this chapter, we present some of the terminology and definitions used throughout the rest of the dissertation. Here we highlight some of the key software layers of mobile systems and identify principal components of mobile applications (also called apps). We begin with an overview of Android which complements the discussions in Chapters 3–4. The discussion on Symbian is relevant to Chapter 3.

2.1 Android

2.1.1 Android Architecture

Android is an open source platform for mobile system development with a standard Linux operating system, a customized runtime, a comprehensive application framework and a set of user applications. It offers many features covering the areas of application development, internet, media, and connectivity. These features include Application framework, Dalvik virtual machine, Integrated browser, Optimized graphics, SQLite for structured data storage, Media support for common audio, video, and still image formats, GSM Telephony, Bluetooth, EDGE, 3G, and WiFi, Camera, GPS, Compass, and a rich Development environment. Based on Linux kernel, it provides a robust driver model, security features, process management, memory management, networking assistance and drivers for a large set of devices. The Android platform primarily consists of five layers as shown in Fig. 2.1.

Applications: This includes a set of core applications that come with the Android distribution like Email Client, Messaging application, Contacts application, Calendar, Map browser, Web browser etc.

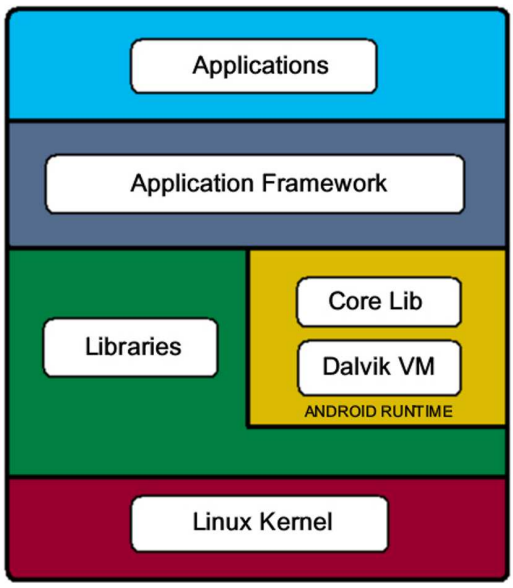


Fig. 2.1.: Android System Architecture

Application Framework: This layer has been designed to facilitate the reuse of components in Android. With the help of Application Framework elements (such as, Intents, Content Providers, Views, and Managers) in Android, developers can build their applications to execute on Android Kernel and inter-operate among themselves and with existing applications.

Libraries: Libraries include System C library, Surface Manager, 2D and 3D graphics engine, Media Codecs, the SQL database SQLite and the web browser engine LibWebCore.

Android Runtime: The Android runtime consists of two components. First, a set of Core libraries which provides most of the functionality available in Java. Second, the Dalvik virtual machine which operates like a translator between the application side and the operating system. Dalvik [21] is a register based [22] virtual machine optimized to run under constrained memory and CPU requirements. Every Android application runs in its own process, with its own instance of the Dalvik virtual machine.

Linux Kernel: Android uses a modified version of Linux for core system services such as Memory Management, Process Management, Network Stack, Driver Model and Security. For more information on the Android platform and a schematic of the Android architecture the readers are referred to [23].

2.1.2 Android Application Components

Here we first explain the different kinds of application components in Android and then explain how the different components coordinate among themselves to achieve a task. This background would be essential to understand the experimental methodology that we have developed because we choose the inter-component messages (called Intents in Android) as the target of our fuzz testing. To understand how Android application components co-ordinate to achieve a task, consider two sample applications (Email and Contacts) shown in Figure 2.2, that co-operate in replying to an email. Consider, a user launching an email application from home screen. This starts an *Activity* (user interface (UI)) showing the user's *Inbox*. She then clicks on an email she wants to read which starts another UI showing a particular *Email* message. To reply, she clicks *Reply* button to invoke a third activity where she can type her response. Consider, she wants to copy her reply to more recipients, so she hits the *"cc"* button to find the address of the recipient. This invokes a fourth activity, i.e., *Select Contact* in Contacts application showing the available email addresses. This fourth activity to user appears as a part of email application but in reality it is from a separate application (Contacts) which runs in a separate process. Further, the main activity in Contacts application, i.e., *Select Contact* calls a *Content Provider*, another application component for data storage, to retrieve the recipient's email address. The sequence of called activities, *Inbox, Email, Reply, Select Contact* to achieve a given *task* involves inter-component communication which can be either inter-application or intra-application.

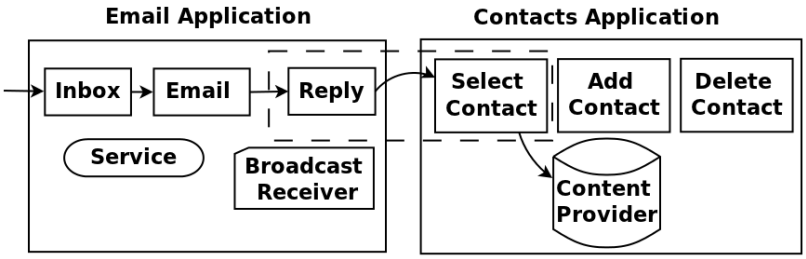


Fig. 2.2.: Android Application Components

Each user application in Android (a *.apk file) typically runs in a separate process and can be composed of *Activities*, *Services*, *Content Providers* and *Broadcast Receivers*. These four components communicate through messages called *Intents* that are routed through Android runtime and the Kernel. The underlying runtime manages the Inter-component Communication. At application installation time, the contract with the runtime is specified in *AndroidManifest.xml*. This contract details on type of components, application permissions, etc. Here we briefly define each of the component types.

Activities: An Activity is a graphical component, which is used to provide the client with a user interface. It is invoked when a user launches an application. An activity can send and receive Intents to and from runtime. It is implemented by extending the *Activity* class while its life cycle is managed by a module in application framework layer called *Activity Manager*.

Services: A Service is used when an application task needs to run in background for a longer time period. For example, a user can run music player in background. Also, a component can bind to a Service to send a request, e.g., a music player Activity can bind to a music player Service to stop the current song that is being played.

Content Providers: A Content Provider is used to manage access to persistent data. The data can be shared between multiple Activities in different applications. Contacts application, as an example, can use the content provider to get a person's phone number.

Broadcast Receivers: A component that is solely responsible to receive and react to event notifications is called a Broadcast Receiver. For example, in SMS application, the Broadcast Receiver component receives an SMS message and displays an alert.

2.1.3 Android IPC

The inter-process communication (IPC) in Android occurs through a kernel space component called Binder (`/dev/binder`), a device driver using Linux shared memory to achieve IPC. The higher level user space components know how to use the binder, i.e., how to pass data represented by *Intents* to Binder. Specifically, when a given component, e.g. Activity Manager, wants to do IPC (either an IPC send or an IPC receive) at OS boundary, it opens the driver supplied by the Binder kernel module. This associates a file descriptor with the thread that called binder, and this association is used by the kernel module to identify the caller and callee of Binder IPCs. All IPC at OS boundary takes place through this descriptor. At the higher level, application-runtime boundary, the application components send Intent messages, e.g., an Activity sends Intents to Activity Manager.

2.1.4 Intents

Intent, a data container, is an abstraction for an action to be performed and forms the core of Android's IPC mechanism. An Intent encapsulates *action, data, component, category* and *extra* fields in its object. As an example, an action can be *dial*, with data as *phone number* and *component* as phone application's main activity. *Category* and *extra* fields give extra information on *action* and *data* respectively. An Intent message can be specifically (**Explicit Intent**) sent to a target component by naming it or it could be resolved by runtime to find a target component. When the target is not explicitly specified in Intent message (**Implicit Intent**), the Android runtime resolves the target component to be invoked by looking up the Intent mes-

sage and matching it against components that can handle the Intent. A given target component can handle an Intent, if it is advertised in a tag called *Intent-filter* in *AndroidManifest.xml*. Different ways in which Intents are sent by application components are: (1). By launching an Activity using `startActivity(Intent)` type of methods; (2). By sending to Broadcast Receivers using `sendBroadcast(Intent)` type of methods; (3). By communicating with a service using `bindService(Intent, ServiceConnection, int)` type of methods; (4). By accessing data through Content Providers.

2.1.5 Android Security

Android provides two important security mechanisms that are different from traditional Unix systems, i.e., application sandboxing and permissions. Sandboxing means each Android application (*.apk) is given its own unique UID at install time that remains fixed throughout its lifetime. This is different from traditional desktop systems where a single user ID is shared among different processes. In Android, since two applications run as two different users, their code may not be run in the same process, thus requiring the need of IPC. Moreover, applications are also assigned separate directories where they can save persistent data. Applications can specify explicitly whether it will share its data with other applications in *AndroidManifest.xml*.

Application permissions is a Mandatory Access Control (MAC) mechanism for protecting application components and data. To use resources, an application requests permissions through *AndroidManifest.xml* file using the *uses-permission* tag at installation time. For example an application that needs to monitor incoming SMS messages would explicitly specify permission of "`android.permission.RECEIVE_SMS`". To protect or share an application's own components, an application can define and specify a certain permission for a caller. This mechanism gives fine-grained control of different protected features of the device but fixes these permissions to install time as opposed to runtime.

2.2 Symbian

Prior to 2011, Symbian held the largest market share for smartphone OSes. It is a lightweight operating system designed for mobile devices and smart phones, with associated libraries, user interface, frameworks and reference implementations of common tools, originally developed by Symbian Ltd [24].

Since mobile phones' resources and processing environments are highly constrained, Symbian was created with 3 design principles: (i) Real time processing, (ii) Resource limitation, and (iii) Integrity and security of user data. To best follow these principles, Symbian uses a hard real-time, multithreaded microkernel, and has a request-and-callback approach to services. Symbian's system model is segmented into 3 main layers [25]:

OS Layer: Includes the hardware adaptation layer (HAL) that abstracts all higher layers from actual hardware and the Kernel including physical and logical device drivers. It also provides programmable interface for hardware and OS through frameworks, libraries and utilities etc. and higher-level OS services for communications, networking, graphics, multimedia and so on.

Middleware Layer: Provides services (independent of hardware, applications or user interface) to applications and other higher-level programs. Services can be specific application technology such as messaging and multimedia, or generic to the device such as web services, security, device management, IP services and so on.

Application Layer: Contains all the Symbian provided applications, such as multimedia applications, telephony and IP applications etc.

Symbian is optimized for low-power battery-based devices and ROM-based systems. Here, all programming is event-based, and the CPU is switched into a low power mode when applications are not directly dealing with an event. Similarly, the Symbian approach to threads and processes is driven by reducing memory and power overheads. Readers are referred to [25] for further details on the Symbian architecture.

In the next chapter, we present our study on failure rates of various applications and middleware components in Android and Symbian.

3. CHARACTERIZING FAILURES IN ANDROID AND SYMBIAN

As smart phones grow in popularity, manufacturers are in a race to pack an increasingly rich set of features into these tiny devices. This brings additional complexity in the system software that has to fit within the constraints of the devices (chiefly memory, stable storage, and power consumption) and hence, new bugs are revealed. How this evolution of smartphones impacts their reliability is the question we answer in this chapter. Here, we analyze the reported cases of failures of Android and Symbian based on bug reports posted by third-party developers and end users and documentation of bug fixes from Android developers. First, based on 628 developer reports, our study looks into the manifestation of failures in different modules of Android and their characteristics, such as, their persistence and dependence on environment. Next, we analyze similar properties of Symbian bugs based on 153 failure reports. Our study indicates that Development Tools, Web Browsers, and Multimedia applications are most error-prone in both these systems. We further analyze 233 bug fixes for Android and categorized the different types of code modifications required for the fixes. The analysis shows that 77% of errors required minor code changes, with the largest share of these coming from modifications to attribute values and conditions. Our final analysis focuses on the relation between customizability, code complexity, and reliability in Android and Symbian. We find that despite high cyclomatic complexity, the bug densities in Android and Symbian are surprisingly low. However, the support for customizability does impact the reliability of mobile OSes and there are cautionary tales for their further development.

3.1 Objectives

In this chapter, our objective is to analyze failure manifestations in Android and Symbian. In that regard, we ask ourselves the following questions:

- What are the applications and libraries that fail frequently in Android and Symbian? Are there any similarity in failure manifestations of these platforms?
- What are the types of code fixes that are applied frequently in Android?
- Does customizability of smartphones impact their reliability?

We answer each of these questions by studying a large number of bugs from Android and Symbian bug repositories. Below, we present our data collection methodology.

3.2 Data Collection

At the time of this study (May 2010), the Android issue reporting site [26] contained more than 8300 bug reports that are categorized into two types, namely *defects* and *enhancements*. We consider these as the manifestations of faults from an application developer's perspective. Our study considers only bugs marked as *defects*. Since the "issue" descriptions are stored as unstructured texts and contained varying amount of details, we had to go through them manually. To prune the database further, we used a set of keywords to list bugs containing those tags. These **keywords** are—*crash, shutdown, freeze, broken, failure, error, exception*, and *security*. The motivation behind choosing these keywords is that these events typically represent significant user inconvenience [7]. The dataset was also filtered to remove duplicate entries. This initially gave rise to a list with 758 bugs reported between Nov 2007 and Oct 2009. Since the bugs between Nov 2007 and Oct 2008 are reported before the official release of Android we further removed these pre-release bugs. The final dataset for Android thus consisted of 628 distinct bugs.

To get an understanding of the terminology used in the issue database, let us consider the bug displayed in Fig. 3.1. This is the entry of a bug related to memory

exhaustion (Issue ID 2203) in the Android issue reporting site [26]. The developer claims that when she tries to rotate the UI of Android more than 20 times, it terminates with an "OutOfMemoryAlert". The bug type is specified as "Defect" and it has medium priority. The "Closed" date in the report indicates that the bug has been fixed and it will be released in future (hence, labeled as "FutureRelease").

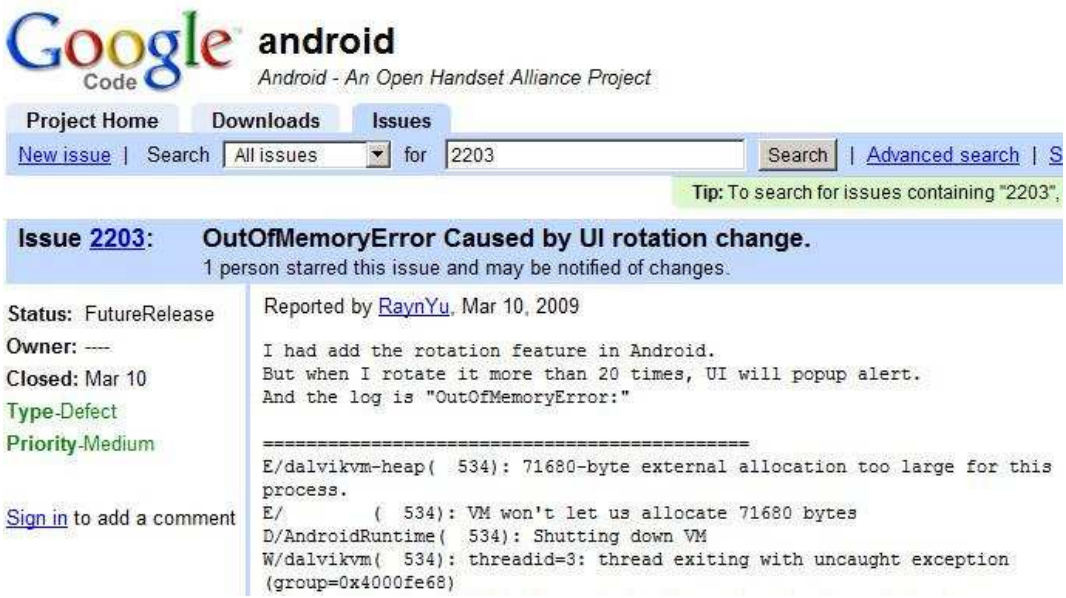


Fig. 3.1.: A Sample Bug Report on the Android Issue Reporting Site

The different categories of the pruned set of bugs (after further removing unhelpful categories "Questions" and "Declined") with the associated counts are shown in Table 3.1.

We applied the same methodolgy for collecting bugs from the Symbian bug tracker [27] which had more than 2700 bugs (as on May 2010). Initially the database was queried with the keywords mentioned earlier. The collected failure reports were pruned by removing duplicates and entries of type "enhancement" or "feature". This gave us a dataset with size 275 spanning the period May 2009—April 2010. After removing the pre-release bugs (before 4 Feb, 2010) our dataset contained 153 distinct failures. The bugs in our Symbian dataset have the status of *new* (the bug has just been reported), *assigned* (the bug has been assigned to an engineer for fixing), *pro-*

posed (a solution has been proposed that is awaiting verification), *closed/worksForMe* (the bug is closed since it could not be reproduced), *resolved/fixed* (the bug has been fixed and the suggested resolution is awaiting verification from the package owner), or *closed/fixed* (the bug had been fixed and the fix is in a release or pre-release version). Table 3.1 shows a breakup of the bugs considered in our analysis for each of the two platforms.

Table 3.1.: Breakup of Bugs Considered in our Analysis

Android		Symbian	
New	330	New	106
Assigned	13	Assigned	15
Reviewed	67	Proposed	21
NeedsInfo	9	Resolved/Fixed	5
FutureRelease	119	Closed/WorksForMe	2
Released	69	Closed/Fixed	4
Unassigned	2		
Unreproducible	19		
Total	628	Total	153

Next, we analyze the failure reports from two viewpoints—the first one is to identify the frequency of failures in different segments of Android and Symbian, whereas, the second one is to classify whether the bugs are permanent, intermittent, or transient.

3.3 Manifestation of Errors

3.3.1 Location of Manifestation of Errors

From the details of the bugs, we initially identified the location where a bug is manifested. Notice that 'location' has different interpretations from different perspectives. From a user's point of view, the location of a bug is the application which fails to run correctly, whereas, from a system developer's point of view, the location is the exact component of Android that fails (often found from stack trace). Our analysis

presents categorization of bugs primarily from application developers' perspective, however, a small fraction of the bugs are also reported by end-users (containing fair amount of details).

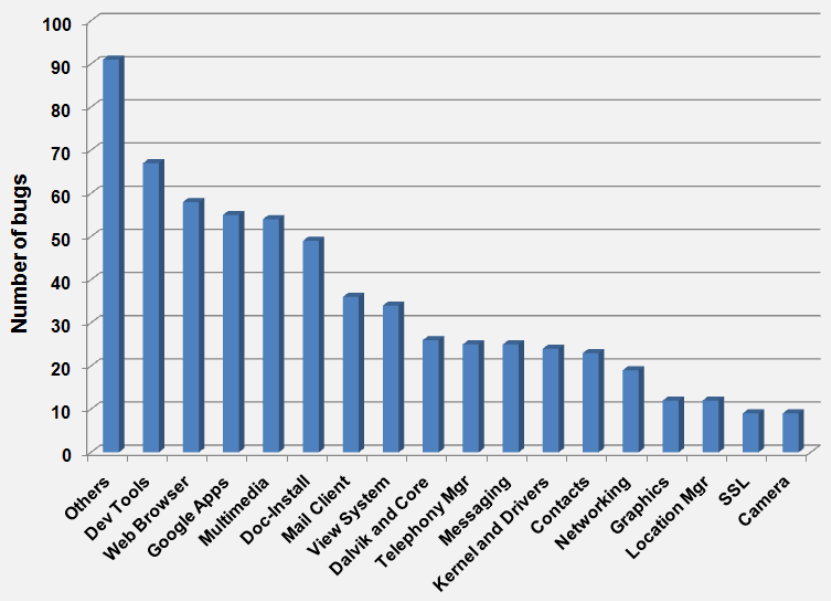


Fig. 3.2.: Manifestation of Bugs in Different Segments of Android. The total number of bugs considered here is 628.

From the bug reports, we first identified 55 different *segments* in Android where bugs were reported. By segment, we mean an individual application or an individual library at which the bug manifested itself, from an end-user's perspective. However, this proved to be too many segments for getting an understanding of the underlying bugs and some segments had very few bug reports. Therefore, we performed an aggregation of some of the related applications and libraries into aggregated segments. Through this we arrived at 18 Android segments. *'Segment'* now represents a built-in application (e.g. Camera, Web Browser etc.), a library in Android (e.g. Graphics, SSL etc.), or an aggregate. The aggregates are: Eclipse, Android Development Tool (ADT), Android Debug Bridge (ADB) as Development Tools; GPS and Location Manager as Location Manager; all the applications that come with Android and are related to Google's services, such as Gmail, Map, and Marketplace as Google Apps;

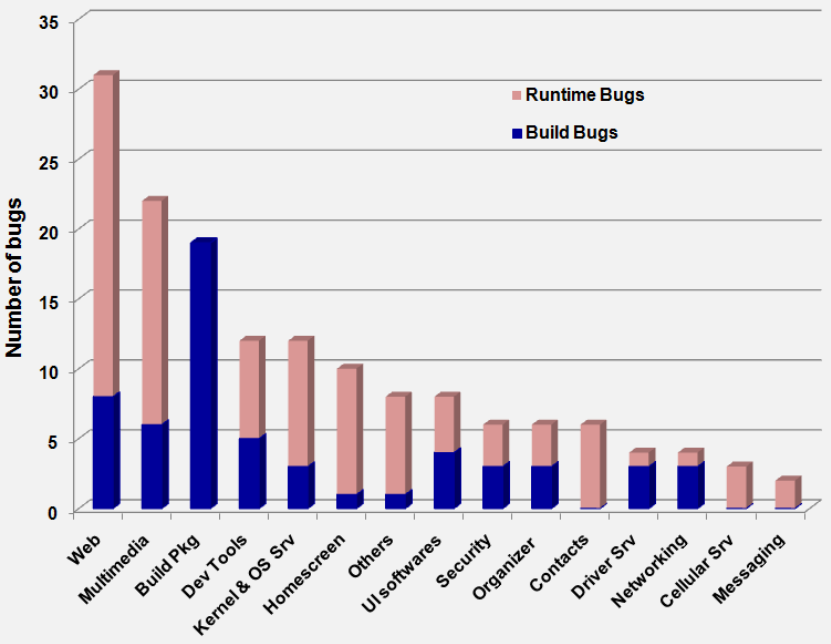


Fig. 3.3.: Manifestation of Bugs in Different Segments of Symbian. The total number of bugs considered here is 153.

Networking library and Wi-Fi library as Networking; Image viewer, Media library, and Media player as Multimedia. Though this grouping introduces disparity among segments in terms of code sizes, they faithfully emulate a user's view (e.g., if a user finds a problem with wi-fi reception, she is likely to report this as a networking issue). We present another analysis on code complexity and bug density later in this chapter (Section 3.4.5). The results on failure manifestation for Android is displayed in Fig. 3.2. The Y-axis in the graph indicates the count of distinct bugs that were reported against a segment.

A large collection of applications that did not have enough severity individually (examples contain failures of Activity Manager, Content Provider, Memory Manager, SQLite etc.) were merged to form the largest segment "Others" (91) in Fig. 3.2. The next most failure-prone segment in Android was Development tools (67) followed by other significant segments such as Web Browser (58), Google Apps (55), Multimedia (54), Documentation and Installation related bugs (49), Mail Client (36), and the

View System (34). It is encouraging to find that a relatively few numbers of bugs are related to Kernel and Device drivers (24 of 628), and the Dalvik and Core Library (26 of 628).

A similar analysis of the Symbian bugs initially resulted in a distribution of 153 bugs in 41 segments ("packages" in Symbian terminology). To get a better understanding we again combined the related packages into a single segment, e.g., the packages *wrttools, web, websrv,* and *webuis* as Web; the packages *podcatcher, mm, graphics, imagingext, mmappfw,* and *musicplayer* as Multimedia; the packages *homescreen* and *homescreensrv* as HomeScreen. This resulted in 15 segments of which only 3 are individual packages (these are messaging, contacts, and organizer) and the rest represent groups of related packages.

It can be seen from Fig. 3.3 that the segment web (31) is most bug prone in Symbian followed by Multimedia (22). Bugs related to building of Symbian packages (19), bugs in the Development tools (12), and bugs in Kernel and OS Services (12) are also significant in number. During our analysis, we observed another interesting pattern in Symbian—as many as 59 bugs in our data set (38.6% of all the bugs) were due to build/compilation errors, missing files, missing references, etc. We denote these types of bugs as "build" bugs. Some of the packages contained significant number of "build" bugs. Examples include web (8 of 31), Multimedia (6 of 22), Development tools (5 of 12), and UI softwares (4 of 8). We display the relative counts of "build" bugs and "runtime" bugs by splitting the bars in Fig. 3.3. Note that the "Build Pkg" in Fig. 3.3 represents bugs specific to the named package and it forms only a fraction of the "build" bugs. The number of "build" bugs in Android did not have such prominence, hence, we do not display the breakup in our analysis. We attribute the large number of build bugs in Symbian to its recent release and believe that these will have less prominence in future releases.

By comparing Fig. 3.2 and Fig. 3.3 it can be seen that of the top 6 bug prone segments in both the platforms, 4 are identical. These are Web browser (*Web* in Symbian), Multimedia, Development tools and Doc-Install (*Build* in Symbian). In-

terestingly, web browsing and multimedia are perhaps the most significant features of a smartphone as perceived by the users. Unfortunately, these are also most failure prone leading to dissatisfaction of users as seen in numerous posts in the user forums (refer Section 3.3.3). One may argue that our findings are biased by the fact that Web browser and Multimedia are more extensively tested by the user community than other applications. However, this once again reiterates the need for making these applications more robust and secure. We further note that the Web browsers in both Android and Symbian are built from the WebKit engine. This raises concern about the reliability of *third-party applications* that are used in the mobile OSes. Apart from WebKit, SQLite, SSL, and Graphics (based on OpenGL) were also found to be error prone in Android.

The presence of large number of bugs in Development tools in both these platforms (10.67% and 7.84% of all bugs in Android and Symbian respectively) draws special attention to this segment. We believe, the efficiency and reliability of these development environments will be a key factor in determining which platform has a larger developer community. Moreover, faults in the development environment can significantly affect the performance of applications and may even be responsible for creating security holes (e.g., a development tool that does not check for known vulnerabilities like SQL Injection and Cross-site scripting). An encouraging finding from our analysis is that both the platforms have lesser number of errors in the lower layers—*Dalvik and Core*, *Kernel and Drivers* in Fig. 3.2 and *Kernel and OS service*, *Driver services* in Fig. 3.3—compared to application level failures. Android, which comes with more applications than Symbian, also has more application level failures (like Mail client and Google apps in Fig. 3.2).

3.3.2 Persistence of Bugs

From the failure reports, we found that only a few bugs in Android were transient (10) or intermittent (49). Most of the failures are permanent (566) in nature and need

to be fixed by modifying the Android code. A few bugs (3) could not be categorized due to lack of sufficient information. It may be noted here that the actual number of transient and intermittent bugs may be much higher as users often refrain from reporting them. Furthermore, many non-permanent bugs were also declined by Android engineers as they could not be reproduced. Similar analysis with the Symbian bugs indicated that only 4 were intermittent bugs and the rest (149) were permanent. We observe that the large number of permanent bugs in both systems (90.12% in Android and 97.38% in Symbian) may be due to the fact these are new operating systems and their codebases are not yet stable.

3.3.3 Analysis of User Forums

Besides the developers' reports, we also studied publicly available data on the T-Mobile G1 user forums for incidence of Android failures [28]. Our analysis considered threads related to *Messaging, Google Applications, Phone & Data Connection,* and *Operating System & Software Development*. It was observed that most of the reports in the user forums are trivial questions or suggestions for enhancements. Therefore we discarded these messages from our dataset. The final list consisted of 105 distinct failures. The failures are frequently reported for Mail Client (15), SD Card (11), Media Player (9), Messaging (9), GPS and Location Manager (8), Web browser(8), Android Marketplace (6), and Calendar (5). This result is not identical to that when we considered the failure reports from a wider audience (Section 3.3.1). This may indicate that the scope of problems in the different modules varies with the hardware device being used. We further noted that the common user-initiated recovery actions are—*Restart application, Wait for some time, Restart phone, Modify settings, Factory reset, Take out battery, Update firmware,* and *Use third-party software*. For example, many users reported that the location displayed in the GPS of Android is 1-2 miles away from where they actually were. Waiting for sometime, rebooting the phone, or doing factory reset may solve the problem, but they do not work on every G1 phone.

These findings about user-initiated recovery actions are consistent with those in the prior work on the Symbian OS [7] in terms of the categories.

3.4 Analysis of Code Modifications and Fixes

The statistics presented in the previous section primarily considered manifestation of failures. Though valuable for identifying the impact of bugs in various segments of mobile OSes, it does not give any indication about how these bugs originated. It is therefore necessary to study the root causes of the bugs and to correlate the user-visible failures with these root causes. This may also help us in identifying error propagation. With these objectives, we studied the code modifications in some of the bug fixes and gained useful insights about the failures. Of the two mobile OSes analyzed in this chapter, we have details of bug fixes only for Android [29].

3.4.1 Data Collection

For this work, we looked into the Android code repositories to study the bug fixes. The code reviews stored in [29] presented us with the details of the fixed bugs. It is to be noted that the failures analyzed in this section have some overlap with the failures studied in Section 3.3 but do not form a strict subset. Several bug-fixes were found which did not appear in the Issue listing site and vice-versa [26]. Our dataset for this analysis contains 233 bug-fixes from 29 projects in the Android repository. These bugs were fixed during the period October 2008 to October 2009. The significant projects within this collection, in terms of the number of bugs, are `kernel/common`, `kernel/msm`, `kernel/omap`, `platform/framework`, `platform/dalvik`, `platform/build`, `platform/system`, and some applications in `platform/packages`. An exhaustive listing of all the Android projects may be found in [30]. In the following sub-section, we present our analysis of root causes of these bugs.

3.4.2 Categorization of Code Modifications

For our analysis, we considered a code-fix to be a *Major* change if it involved modifications of more than 10 lines of code, or modifications at more than 5 places in the source file(s). It was observed from the bug-fixes that most of the bugs (179 of 233) required only few lines of code changes. Among the minor modifications, we further identified what types of code changes were most frequent. We categorize different types of code modifications as follows.

1. *Add/modify attr val:* Update the value assigned to a variable (e.g. the code `A.x=B.y` is corrected as `A.x=C.z`) or declare a new variable.

2. *Add/modify cond:* Add some new checks (if-stmt), or add a missing else clause, or modify the condition expression.

3. *Modify settings:* Update system constants or include modification in the makefiles, application configuration files etc.

4. *Add/modify func call:* Introduce a new function call, or modify the arguments of an existing invocation.

5. *Lock problems:* Bug was caused because a critical segment was not locked or a lock was not removed and deleted upon exit.

6. *Add/modify lib ref:* Bug was caused because code was accessing some non-existent or incorrect libraries or classes.

7. *Modify data type:* Update the datatype of a variable.

8. *Preprocess change:* Introduce a preprocessor directive (e.g. adding an `ifdef`).

9. *Reorganize code:* Change the order of execution of certain code blocks.

10. *Others:* The bug fixes that could not be placed under any of the above-mentioned categories are considered here.

We designed these categories to incorporate maximum details of root causes in our classification while maintaining a programmer-centric view. We do not adopt existing classification approaches like ODC [31] since ODC primarily measures the

effectiveness of various software development stages. Our work, on the other hand, looks at manifestations of failures and related programming errors.

In our analysis, some minor fixes contained multiple changes (e.g. some bugs needed both modification of settings and addition of new attribute values), hence, they were considered under both the categories. This resulted in a list of 193 *fixes* for 179 distinct failures. We show the breakup of the different bug fix categories in Fig. 3.4. We observe that only 23% of the bugs required major changes. These modifications primarily include addition of new functions, data structures, and constants. Among the minor changes (77% of all the bugs), most of the modifications were of the type *Add/modify attr value* (21%) and *Add/modify cond* (19%). The large percentage of *Add/modify attr val* (21%) arises due to the fact that some applications/drivers in Android are still undergoing major code revisions. The list of changes may be seen from the release notes of different versions of Android. Within the sizable category *Add/modify cond* (19%), we observed several instances where an *if-stmt* did not have a corresponding *else* clause. This resulted in exceptional cases not being handled correctly. Detailed specification of the program behavior could have avoided such errors. It is known that introducing new conditional statements adds to the cyclomatic complexity of a program. Hence, while implementing these fixes, the designers must be careful so that understandability and testability of the resulting code is not altered significantly, e.g., if a fix introduces a high degree of nesting for conditional statements, the designers may try to simplify by reorganizing the code.

3.4.3 Tension between Customizability and Reliability

The presence of 14% *Modify settings* bugs motivated us to delve deeper into the Android code base and analyze the flexibilty provided by Android runtime environment to the upper layer applications. The customizability claim is buttressed by the fact that Android may be adapted for a wide range of mobile hardware (people

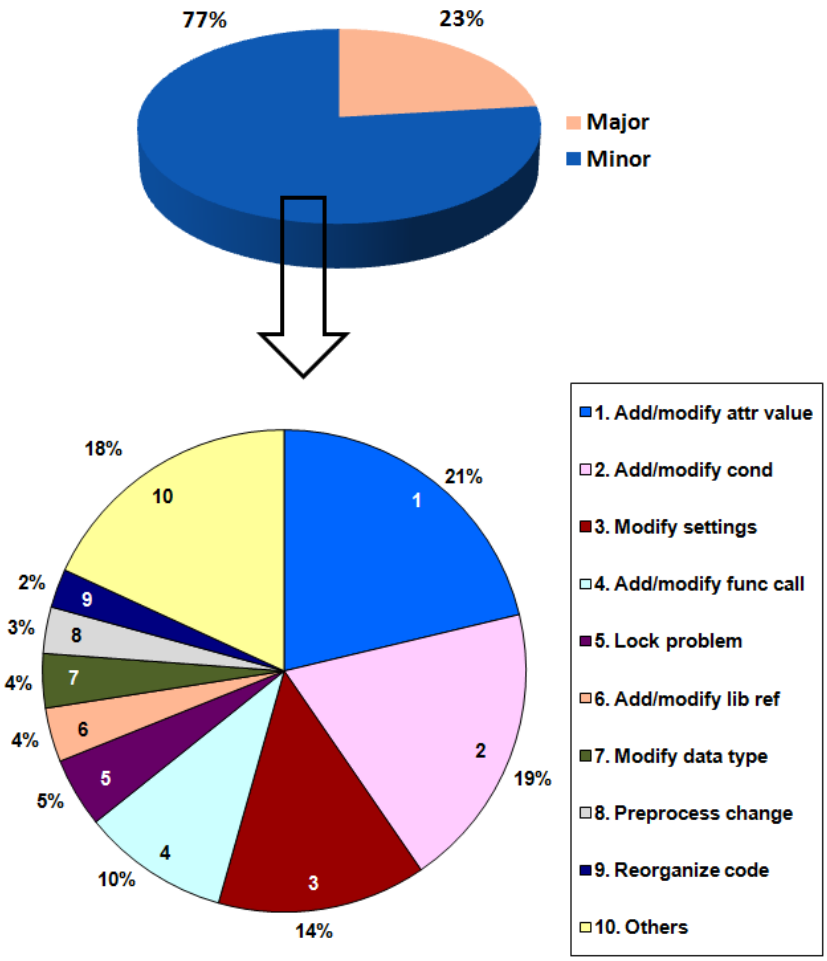


Fig. 3.4.: Different Types of Code Modifications. Total number of bugs considered for this analysis was 233.

have even used it to run notebooks), it incorporates virtual machine, SQLite for data storage, and the fact that people may use their phone to write programs.

We observed that a large chunk of the failures requiring *Modify Settings* surfaced during building (compiling) the Android source code. This resulted in modifications of the Makefiles to support new architectures and APIs, modification of environment variables, changing application permissions etc. Note that many of the application configuration files are also generated during the build process itself. Hence, we consider that *Modify settings* errors relate to customizability of the system, where customizability is defined to include both the process of building the system and of

executing the system. However, this category of bugs only covers a subset of the errors caused by the need to support customizability. According to our categorization, if a new condition needs to be introduced in the code file to handle a configuration parameter, that would be classified under *Add/Modify Condition*, while some support for customizability may necessitate large changes in the software (considered under the category *Major*). Thus, the percentage of bugs to support customizability is non-trivial. This suggests that customizability does have some negative impact on the reliability. But this is not egregiously high for the level of customizability supported by Android. Further, improvements in software practices, especially targeted at the problematic segments that we have identified here, plus a natural maturing of the code base (recollect that we are talking of something that has been open sourced for only about a year and a half) will likely bring these bug incidences down.

3.4.4 Analysis of Environment Variables

We observe that applications often read system configurations and locations of various executables from the environment variables defined in the runtime kernel. These parameters, though not comprehensive, are a significant indicator of a framework's customizability. Extending from the conclusions in the previous section, we counted the number of environment variables defined in the Android platform and compared it with a standard Linux Kernel (version 2.6.32). We wrote a script to scan the source codes of different versions of Android to find the occurrence of `export` or `setenv` keywords. We then built lists of environment variables for each of the Android versions and the Linux Kernel. Next, we counted the references to each environment variable and summed them up. Though a variable may be referred multiple times within a single line, we consider these as a single reference. The results obtained from our analysis are presented in Table 3.2. "Max ref" is the maximum number of references to a single environment variable in the entire code base.

Table 3.2.: Count of environment variables and their references in different versions of Android and the Linux Kernel

	# env vars	Total refs	Max ref
Android 1.1	62	819	577
Android 1.5	63	854	584
Android 1.6	76	1545	584
Android 2.0	82	2083	592
Linux Kernel 2.6.32	127	953	158

The number of environment variables in different versions of Android is steadily increasing. Though this number (82 in Android 2.0) is lower than in the Linux Kernel (127), it is still significant considering that Android is built as a mobile OS and runs on devices with more constrained resources than the Linux kernel. Also, the growth in the number of references to environment variables between February 09 (Android v. 1.1) and October 09 (Android v. 2.0), 154%, is striking evidence of the rapid march toward a customizable mobile OS. More than 85% of the references to all the environment variables were made from codes in `external/` folder in Android which include third-party libraries and built-in applications.

In Fig. 3.5, we illustrate the distribution of the number of references to environment variables. The number of variables with reference count of zero indicates that a large number of environment variables were not referenced outside the line where they were defined or "exported". We also noted that majority of the references were made to only a few of the variables (less than 5). For example, in Android 1.6, the environment variables `DESTDIR`, `MK`, `CFLAGS`, and `LDFLAGS` are referenced 584, 308, 194, 117 times respectively. The maximum reference count for a single environment variable was much higher in Android than in Linux Kernel. The references to environment variables in Linux Kernel are almost evenly distributed, whereas, Android is more dependent on a few critical variables. This indicates in Android a possibility of significant error propagation if these key environment variables happen to be

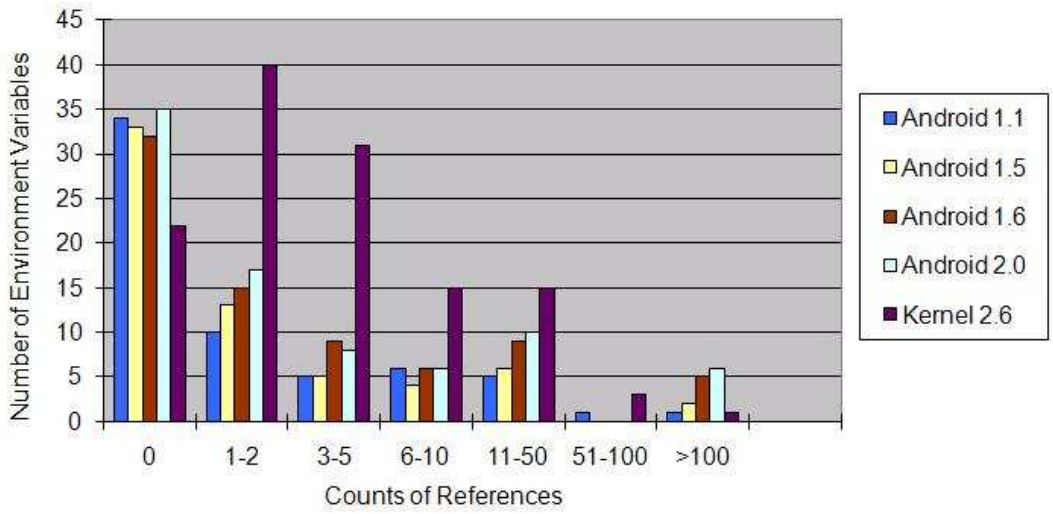


Fig. 3.5.: Distribution of references to environment variables

incorrectly set. As a corollary, Android will benefit from building in reasonableness checks for these key environment variables.

3.4.5 Cyclomatic Complexity and Number of Bugs

Cyclomatic complexity, which measures the number of linearly independent paths through a program's source code, is frequently used as a metric of code complexity. To understand the relation between cyclomatic complexity and bug density in Android and Symbian, we selected a set of projects (packages) in these two platforms that had the maximum number of bugs. Since, the Android issue reporting site [26] does not contain the root cause of a bug, we considered the bug counts in section 3.4 for computation of bug density. For Symbian, we found that each bug was assigned to its corresponding project. Hence, the bug density in Symbian is computed with the counts presented in Fig. 3.3. We computed cyclomatic complexities (CC) of these projects (packages) with Understand 2.0—a source code analysis and metrics generation tool [32]. The unit for computation of cyclomatic complexity is a function, as is typically done. Source codes for these projects (packages) were downloaded from Android [30], and Symbian [33] repositories. Note, that the Android repository gives

Table 3.3.: Cyclomatic Complexity and Bug Density of different projects in Android

Projects	**Bug Density** $\times 10^4$	**No. of bugs**	**Source LOC**	**Avg. Cyclomatic**	**Max Cyclomatic**
kernel/omap	0.04	21	5,311,427	1.12	4,973
kernel/msm	0.06	29	4,724,260	5.60	4,973
kernel/common	0.07	31	4,688,175	5.82	4,973
dalvik	0.18	14	771,865	2.23	766
development	0.46	10	216,344	2.18	169
framework/base	0.79	51	645,978	2.40	221
packages/apps/camera	1.33	2	14,962	2.15	20
packages/apps/mms	1.74	4	23,013	2.02	46
system/core	1.90	13	68,798	4.31	167
hardware/msm7k	2.42	3	12,382	4.00	23

Table 3.4.: Cyclomatic Complexity and Bug Density of different segments in Symbian

Segments (Fig. 3.3)	**Bug Density** $\times 10^4$	**No. of bugs**	**Source LOC**	**Avg. Cyclomatic**	**Max Cyclomatic**
Kernel and OS Services	0.03	12	3,684,192	3.02	1,470
Security	0.08	6	752,148	2.29	134
Multimedia	0.12	22	1,866,577	2.44	558
Web	0.17	31	1,807,828	3.01	2,442
HomeScreen	0.38	10	263,305	2.25	149
Build Pkg	0.63	19	299,868	2.24	268

us the latest source codes, whereas, our bug pool is older than the source code. Hence, the calculated bug density is not completely consistent. Nevertheless, this gives an approximate measure of code complexity in Android and may be used to suggest improvements in code quality. Table 3.3 and Table 3.4 show the results obtained from our analysis.

It was observed that in Android many projects in `kernel/*` have large overlap in their code files. As a result, the Max cyclomatic complexity and Source Lines of Code (SLOC) of these projects are similar or identical. Qualitatively, the bug density is quite low for both these systems indicating a high standard of code development, even though these are relatively new software projects. For reference, the pre-release bug density in Windows XP was 2.66×10^{-3} [34]. However, the bug density in the Kernel and OS Services of Symbian was lower compared to Android Kernel.

In standard literature, it is suggested that cyclomatic complexity of functions should be limited to 20 for manageability of code. Hence, the Max. cyclomatic complexity figures presented in the table may initially appear erroneous. A careful examination of the source code, however, revealed that some functions extensively use macros inline, each of which contains multiple if-else statements. When the macros are replaced by the preprocessor with their corresponding codes this gives rise to high cyclomatic complexity. Such inlining is the main reason for our high cyclomatic complexity in both these tables. We further noted that large case-switch statements were, in many cases, responsible for cyclomatic complexity above 100. The presence of long case-switch statements necessitates the creation of extensive code documentation explaining each of the cases. Verifying that all the cases have been handled properly is a challenging task. Going forward, as mobile OSes increases in complexity, developers will need to pay careful attention to managing the long switch statements.

Average cyclomatic complexity (CC) per function, on the other hand, was significantly lower (between 1.12 and 5.82 in Android and between 2.24 and 3.02 in Symbian). This is primarily due to the presence of a large number of default and inherited functions which have complexity 1. We also measured the CC of the Linux kernel (version 2.6.32). This system consisting of 6,082,112 lines of source code had a maximum CC of 4,973 which is identical to the Android kernel. This can be explained by the fact that Android kernel is built using a modified version of Linux kernel (v2.6). It was observed that the max CC in Symbian Kernel and OS Services is much lower than that in Android and the Linux Kernel, while their average CCs are comparable.

3.5 Directions for Future Research

In this chapter, we presented a failure characterization of mobile OSes based on large number of bugs in Android and Symbian. This lays the foundation for further research on smartphone reliability by highlighting the components that fail often.

However, the results presented in this chapter were based on data collected during early phases of Android and Symbian release (2008-2010). We surmise that the failure frequency of various components may have changed over the years. We find some evidence of this in our next work on Android IPC Robustness, where the number of crashes in Android 4.0 were far fewer than Android 2.2. Moreover, Android has released many new and interesting components (or significantly redesigned older ones) over the years for which we did not have failure data. An example of this may be Android's "In-App Billing" APIs. We find the following to be an exciting research direction for analyzing smartphone reliability.

Reliability with Software Evolution in Android: A study on Android reliability across its versions would help us understand how its reliability has changed over the years. We found that during major releases of Android components, several new issues have surfaced and often older phones were no longer compatible with new services. As an example, [35] shows that a large number of users had problems accessing Google Play Services after they upgraded to newer versions. A study across versions would help us identify the components that has a (possibly) faulty regression testing setup.

4. ROBUSTNESS TESTING OF ANDROID IPC

Android has a modular framework with multiple components in each application, and a security-conscious design where each application is isolated in its own virtual machine. However, its isolation guarantees would be rendered ineffectual if an application were to deliver erroneous messages to targeted applications and thus cause the target to behave incorrectly. In this section, we present an empirical evaluation of the robustness of Inter-component Communication (ICC) in Android through fuzz testing methodology, whereby, parameters of the inter-component communication are changed to various incorrect values. We show that not only exception handling is a rarity in Android applications, but also it is possible to crash the Android runtime from unprivileged user processes. Based on our observations, we highlight some of the critical design issues in Android ICC and suggest solutions to alleviate these problems.

4.1 Objectives

Of the two Inter Component Communication (ICC) primitives in Android—Intent and Binder—we use Intent as the subject of our robustness study due to its flexibility. Intents are used for a variety of purposes in Android applications which include but are not limited to—starting a new activity, sending and receiving broadcast messages, receiving results from another activity, starting and stopping a service etc. To support these operations across a myriad applications from multiple vendors over many versions, Intent messages have a flexible structure and therein lies the potential for vulnerability. In a vulnerability analysis of Android IPC, Chin *et al.* [36] argued that it is easy to spoof, snoop, and target Intents to specific application components unless these are protected by explicit permissions, which is a rare occurrence. Our

experimental results concur with this analysis and show that the attack surface can go even deeper (i.e. up to the framework layer or lower as shown in [23]). Due to these reasons we chose Intents as the primary focus of our study. In essence, we try to answer the following questions:

(A) How well does an Android component behave in the presence of a semi-valid or random Intent?

(B) How robust are Android's ICC primitives? Can the Android runtime contain exceptions within an application?

(C) How can we refine the implementation of Intents so that input validation can be improved?

To evaluate (A), we sent explicit Intents to each Activity, Service, and Broadcast Receiver registered in the system. We evaluate (B) by sending a set of implicit Intents and answer (C) by presenting a qualitative assessment in Section 4.4.

4.2 Experimental Setup

4.2.1 Design of JarJarBinks

We built our robustness testing tool, JarJarBinks, from Intent Fuzzer at [37]. The initial codebase contained basic functions like displaying set of components registered in the system, and sending blank Intent messages to Broadcast Receivers, and Services. However, it did not support testing Activities. We added this key feature in JarJarBinks along with an Intent generation module described in Section 4.2.2. Fig. 4.1 shows the location and operation of JarJarBinks (JJB) with reference to Android architecture [23]. It queries Android PackageManager to get a list of components (Activities, Services, and Broadcast Receivers) registered in the system and then uses ActivityManager to send Intents to these components. We use the following methods from Android API to send Intents: `startActivityForResult` for Activities, `startService` for Services, and `sendBroadcast` for Broadcast Receivers.

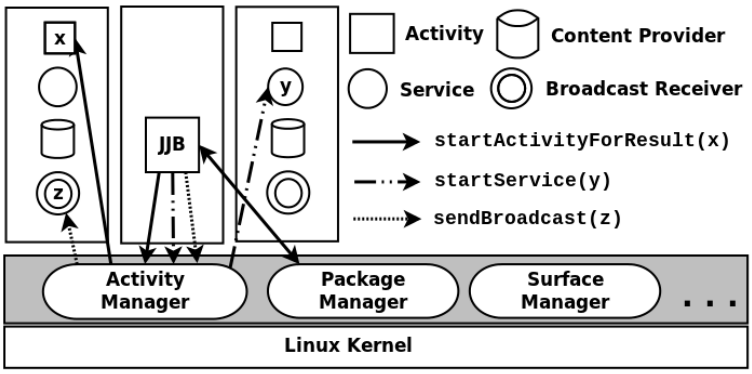


Fig. 4.1.: JarJarBinks: Interaction with Android Layers

One of the major challenges in automated testing of Android Activities is to close a callee Activity after sending an Intent. Typically, once a new Activity is displayed, it expects some interaction from the user and pauses the caller Activity. We resolved this by using `startActivityForResult()` and `finishActivity()` APIs in Android. Unlike `startActivity()`, `startActivityForResult()` can force-finish a child activity by using its `requestCode` as a handle. This way we could avoid manual intervention in most cases. Another design issue with automated testing of ICC in Android is to avoid resource exhaustion in the system (e.g., sending a continuous stream of Intents very fast would create a large number of Activities (windows) causing WindowManager to run out of resources). For this purpose, we used a pause of 100ms between sending of each successive Intent. This was sufficient to launch and finish a new Activity (or Service) in our testing environment. Though we did not explicitly test Content Providers in JarJarBinks, semi-valid content URIs were specified in some of our fault injection campaigns triggering parsing of these content URIs and corresponding permission checks.

It may be highlighted that one of our goals was to keep the implementation of JJB simple and less intrusive, thereby, not introducing new bugs in the firmware. We, instead, focus on a rigorous analysis of the results obtained from our experiments. Despite its simplicity, the volume and severity of failures generated through JJB is truly astonishing. One shortcoming of JJB is its semi-manual approach—our strategy

of killing a child Activity (by calling `finishActivity`) did not work well in two situations: first, when a system alert was generated due to application crash, this could not be closed programmatically (we consider this as a good security design; JJB being a user-level application cannot hide system alerts), second, when an activity was started as a new task the caller could not close it by calling `finishActivity()` (this mostly happened while launching login screens of applications like Skype, Facebook, Settings etc.). Both these cases required manual intervention and will be addressed in our future work. In the following section, we present an overview of our Intent generation module.

4.2.2 Generating Intents

An Intent message is essentially a data container having a set of optional fields—{`Action, Data, Type, Package, Component, Flags, Categories,` and `Extras`}—which can be specified by a caller. Of these, `Action` (an action to perform, e.g. to view or edit a contact) and `Data` (a URI for a data item, e.g. URI for a contact record on phone) are most frequently specified by a caller. `Component` specifies the target component, `Flags` control how an Intent is handled, `Category` specifies additional information about the action to execute, and `Extras` include a collection of name-value pairs to deliver more inputs to the target component. `Type` (content mime-type) is usually determined from `Data` (when it is specified), while, `Package` can be determined from `Component` if one is specified.

In JarJarBinks, we modify the fields `Action, Data, Component,` and `Extras` in a structured manner as part of a fault injection campaign and keep the other fields blank (we select `Extras` since this can potentially include random or malicious data from users). For most experiments `Action` is selected from a set of Android-defined action strings found at [38]. Generation of data URIs is a non-trivial operation due to the presence of a multitude of URI schemes. A URI consists of three parts `URI := scheme/path?query`, where `scheme` denotes URI type, `path`

gives the location to the data, and `query` is an optional query string. At present we support the following URI schemes—`"content://"`, `"file://"`, `"folder://"`, `"directory://"`, `"geo:"`, `"google.streetview:"`, `"http://"`, `"https://"`, `"mailto:"`, `"ssh:"`, `"tel:"`, and `"voicemail:"` in JarJarBinks. For each of these except `"content://"`, we created a predefined set of semi-valid URIs. For `"content://"` URIs, JarJarBinks first queried the PackageManager to get a list of registered Content Providers in the system and then randomly selected one of them to build a `content://provider` URI. Our Intent generation can be broadly classified into two types.

Implicit Intents

Components in the system can advertise their ability to handle Intents by specifying Intent-filters in their manifest file. Implicit Intents do not specify a target, but are delivered to the best matching component in the system. The matching between sender and receiver is the responsibility of the Intent delivery mechanism of the platform. Intent-filters can restrict the `Action` of the Intent, the `Category`, or the `Data` (through both the URI and the data type fields) or any combination of the three. The test set for implicit Intents is therefore any Intent that matches at least one Intent-filter in the system. In order to generate Intents, we collect all Intent-filters of all applications and all restrictions of either the `Action` or the `Category`. On our target platform, we could not find components using the `Data` in Intent-filters. For each application and each of its Intent-filter, the following experiments were performed:

(A) *Valid Intent, unrestricted fields null:* We generate an Intent that matches exactly all the restricted attributes of the Intent-filter but leave all other fields blank. For example, if the Intent-filter specifies `<action android:name="ACTION_EDIT" />`, only this information is used to populate the Intent fields.

(B) *Semi-valid Intent:* We pick all Intent-filters that have at least one degree of freedom and set these fields sequentially to each of the valid literals we discovered in

any other Intent-filter. For the above example, the `Category` field would be subject to fuzzing since only `Action` is restricted through the filter. Thus, the fuzzed fields are individually valid for some component in the system, but not their combination. Since each individual field in the generated Intent is valid, there is still a high chance that it is routed to *a* component.

Explicit Intents

Our goal here is to find how well the receiver of an Intent behaves after getting unexpected data. At a high level, our fuzz campaign on explicit Intents is distributed over three component types—Activities, Services, and Broadcast Receivers. For each component type, JarJarBinks first queries PackageManager to retrieve a list of components of that type in the system (e.g. all the Services, or Activities). After this, for each selected component (e.g. Calender Activity) JarJarBinks runs a set of four fuzz injection campaigns (FIC).

FIC A: *Semi-valid Action and Data:* Here a semi-valid `Action` string, and `Data` URI are generated as described earlier (refer Section 4.2.2). However, the combination of the two may be invalid. For example, an Intent of this category may be `Intent {act=ACTION_EDIT data=http://www.google.com cmp=com.android.someCompo -nent}`. During this FI, the `Action` and `Data` sets are combined to generate all known `{Action, Data}` pairs each generating a new Intent. Total number of Intents generated are $-\texttt{Action}'' \times -\texttt{Data}''$ for each component. Fields other than `Action` and `Data` are kept blank.

FIC B: *Blank Action or Data:* In this experiment, we specified either `Action` OR `Data` in an Intent but not both together. Other fields are left blank. `Intent {data=http://www.google.com cmp=com.android.someComponent}` is an example of this FI. This campaign generates $-\texttt{Action}'' + -\texttt{Data}''$ Intents for each component.

FIC C: *Random Action or Data:* Here either `Action` OR `Data` is specified as described earlier, and the other is set to random bytes. An example of this type of Intent

may be `Intent {act=ACTION_EDIT data=a1b2c3d4 cmp=com.android.someCompo -nent}`.

FIC D: *Random Extras:* For this FI, we first created a set of 100 valid {`Action, Data`} pairs following Android documentation. For each of these pairs, 1-5 `Extra` fields were added randomly. The name of an `Extra` was selected from the set of Android defined `Extra` strings, while its value was set to random bytes. An example Intent can be shown as, `Intent {act=ACTION_DIAL Data=tel:123-456-7890 cmp=com.android.someComponent has Extras}`.

Our choice of experiments is justified by the fact that an application component may get a malformed Intent either due to error propagation from other applications or from an active adversary. While FICs A and B verify the robustness of a callee component against null objects and incompatible actions, FICs C and D emulate the behavior of a potential adversary.

4.2.3 Machines and Firmware

We conducted our robustness test on three versions of Android, distributed on three phones and three computers—two of the phones (Motorola Droid) had Android 2.2 as its firmware (release date: June 2010 and nicknamed "Froyo"), while one (HTC Evo 3D) had Android 2.3.4 (release date: April 2011 and nicknamed "Gingerbread"); the computers all ran Emulators loaded with Android 4.0 in Linux environments (release date: October 2011 and nicknamed "Ice Cream Sandwich", the image of which was useful during long late night experiments with it). The HTC Evo was used for running experiments on implicit Intents. Experiments on explicit Intents, where we sent a large number (9000) of Intents to each Android component, being more time consuming, was run in parallel on two Droid phones (having identical hardware and firmware). The emulators were used for testing Android 4.0, the latest version of Android, for which a physical device has been available only in late November 2011, clearly not enough time for us to carry out experiments. Android 4.0 is a promising

target of the study since it has been widely hailed as "the biggest Android update in ages" (PC Magazine) and is touted to bring real improvements to the Android platform. Initially, it was noted that the devices as well as the emulator had nearly 800 components (Activities, Services, and Broadcast Receivers combined) per version of Android which include a large number of third-party applications. In this work, we focus our attention to Android framework and common applications that are pre-loaded into every Android distribution (e.g. contacts, calendar, messaging etc.). These application are also used by third-party application in implementing common functionalities. Hence, rigorous evaluation of these built-in applications are of prime importance. In Android namespace hierarchy, these applications all share the package name prefix of `com.android`. After filtering the list of components with this prefix we found 398 components (297 Activities, 42 Services, and 59 Broadcast Receivers) in Droid and 455 components (332 Activities, 54 Services, and and 69 Broadcast Receivers) in Emulator.

In addition to built-in applications, we also tested 5 Most popular (as on 3 Dec, 2011) free apps from Android Marketplace (recently renamed Google Play). These apps—Facebook, Pandora Radio, Voxer Walkie Talkie, Angry Birds, and Skype—had a total of 103 Activities and 11 Service components. Even though our set of Marketplace apps is small, the large number of Activities (103 as opposed to 294 in Droid) gives us a realistic comparison of their robustness with that of Android. Our experiments started by subjecting all these (Android and Marketplace) components to a flow of Intents from JarJarBinks over a seven day period. In the following section, we present our findings.

4.3 Results

During the course of our experiments, more than 6 million Intents were sent to 800+ components across 3 versions of Android. We define an experiment as follows: *Choose one particular component and inject all the Intents targeted to that component.*

The injection is done according to the Fault Injection Campaigns (thus, if we are doing FIC A, the `<Action, Data>` *pairs are changed to semi-valid values).*

We collected execution logs from the mobile phones and emulators using `logcat`, a logging application in Android platform tools. This generated more than 3GB of log data which were later analyzed to gather information about the failures and their root causes. We define a **crash** to be a user visible failure, i.e., a system alert displaying the message `"Force Close"` (in Android 2.2) or `"Application x stopped unexpectedly"` (in Android 4.0). These failure messages manifest in the log files as a log entry stating `"FATAL EXCEPTION: main"` and are essentially effects of uncaught exceptions thrown by the Android runtime. It is to be noted that sending(receiving) of certain Intents (e.g. `<action=ACTION_SHUTDOWN>` or Intents with `"content:"` URIs in `Data` field) in Android are protected by permissions and when JJB sends these Intents `SecurityException`s are generated. JJB is able to handle these exceptions gracefully and we discard these from our results. At present we focus on crash failures as opposed to thread hangs due to their visibility and negative user experience.

We discuss our results from three perspectives: (i) prevalence of crashes caused in the application components due to the fuzzed Intents for the various types of components and different fault injection campaigns; (ii) distribution of uncaught exceptions thrown by components in response to the fuzzed Intents; and (iii) error propagation from a user-level application to the Android framework.

In general, Android 2.2 displayed many more crashes than Android 4.0 and components in all the versions were vulnerable to `NullPointerExceptions`. It was possible to crash some components by sending them an implicit Intent that matched exactly with their Intent-filter (i.e. nothing other than the mandatory fields were specified). In Android 2.2, three of the application crashes caused cascading failures which eventually restarted the Android runtime. The Android Emulator also showed signs of stress-related failures, whereby, the `system_server` (the framework component that coordinates interaction between Kernel space and user space) restarted periodically after testing a fixed number of components. The `system_server` is a key part of

the Android environment—it runs a host of essential services (Power Manager, Device Policy, Search Service, Audio Service, Dock Observer, etc.). A crash of the `system_server` kills all user level application and services and restarts the Android runtime.

Below we present our experimental results organized into three discussions.

4.3.1 Results for Explicit Intents

In Section 4.2.2, we described how we generated explicit Intents for four different fault injection campaigns. In FIC A we sent an invalid `<Action, Data>` pair to components, in FIC B we sent an Intent with either `Action` or `Data` blank, in FIC C random bytes were assigned to either `Action` or `Data`, and finally in FIC D random bytes were assigned to `Extras` values. During our experiments we found a large number of crashes—2148 in Android 2.2, 641 in Android 4.0, and 152 for Marketplace apps. One may argue that a comparison between Android 2.2 on a real phone and Android 4.0 on an emulator compromises the validity of our results. To verify this, we conducted a smaller-scale test of Android 2.2 on emulator and Droid and did not find any major difference. Our choice of Android 4.0 on emulator was driven by the lack of a physical device in a timely fashion. Even if results obtained from a physical device change from its emulator (i.e. absolute numbers of crashes change), it does not invalidate the general trends described in our results. Below, we present an analysis of the observed crashes.

Distribution of Failed Components

We define a failed component to be a program that crashes at least once during a fuzz injection campaign. Due to the nature of our Intent generation it is possible that a component fails repeatedly in one experiment where that component is targeted, e.g. an activity that dereferences `Data` field without null check will crash for all Intents that has a blank `Data` field. Counting such repeated crashes masks the actual

Table 4.1.: Summary of component crashes in different versions of Android in response to fuzzed Intents in four different injection campaigns. Here one component crashing one or more times in response to one or more malformed Intents directed at it counts as one crash.

	Droid (Android 2.2)						Emulator (Android 4.0)						Marketplace Apps on Droid (Android 2.2)					
	Activities 297		Services 42		Broadcast Receivers 59		Activities 332		Services 54		Broadcast Receivers 69		Activities 103		Services 11		Broadcast Receivers 10	
	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%	#crash	%
A. Semi-valid	30	10.1	1	2.4	2	3.4	29	8.7	3	5.6	2	2.9	4	3.9	0	0.0	0	0.0
B. Blank	21	7.1	1	2.4	6	10.2	8	2.4	3	5.6	6	8.7	2	1.9	1	9.1	0	0.0
C. Random	18	6.1	1	2.4	4	6.8	9	2.7	3	5.6	2	2.9	2	1.9	0	0.0	0	0.0
D. With Extra	13	4.4	1	2.4	1	1.7	7	2.1	3	5.6	0	0.0	3	2.9	0	0.0	0	0.0

number of faults at source code, therefore, for a fault injection campaign like ours, a better metric of a framework's reliability can be obtained by finding how many failed components it has. Table 4.1 presents the number of failed components for various types (Activity, Broadcast Receiver, and Services) in each of our experiments. The number at the top, under the component type represents the total number of components of that type, e.g., Android 2.2 has 297 Activities. The number in the column "#crash" denotes the number of components that crashed.

It is encouraging to see that in all cases but two, the percentage of failed components is less than 10. The percentage of failed components in Android 4.0 is generally lower than in Android 2.2, with the exception of Services. Across experiments, Activities display higher fraction of failed components in FIC A than the rest. However, this may also be due to the fact that FIC A sends nearly twice as many Intents than FICs B, C, and D combined. The high count of failed components across component types in FIC B is another key finding of our experiments. This indicates that many Android components do not perform null checks before dereferencing a field from an Intent and, therefore, are vulnerable to blank fields. This fact is also verified by our data in the next section.

The failure percentages of Marketplace apps are nearly identical to that of Android 4.0 components with the exception of FIC A for Activities and for Services, where Marketplace apps are significantly more robust. However, it was observed that 3 of the apps had at least one component that failed one or more experiments. Though our sample size for Marketplace apps (5) is too small to make any claims about general robustness of third-party apps, we expected the Top 5 to be more robust as they come from reputed vendors. This intuition is only partially borne out by the analysis results.

Distribution of Exception Types

To understand how well the Android framework handles exceptional conditions, we measured the distribution of exception types from failure logs. Here, we are focused on uncaught exceptions, because they result in the crashes. Since we are interested in measuring what percentage of all the crashes are constituted by a given exception type, here we count each crash individually. Thus, if in one experiment, 100 fuzzed Intents are sent to a component and the component crashes 20 times, we will have 20 data points (unlike in Section 4.3.1 where we would have counted the component as having crashed and it would have resulted in a single data point). It can be seen from Fig. 4.2 that `NullPointerExceptions` (NPE) make up the largest share of all the exceptions. Though the percentage of NPEs in Android 4.0 (36.50%) has improved since Android 2.2 (45.99%), this is still significant and concurs with our findings in Section 4.3.1. The results are given in terms of percentage of all the exceptions, thus for a given Android version, all the exceptions' numbers should sum to 100%. Other exceptions like `ClassNotFoundException` and `IllegalArgumentException` are significantly lower in Android 4.0 than in its previous version. Though exception types are sensitive to input data, we are applying similar inputs to the two different versions of Android. Therefore, our comparisons across the two versions are still valid.

However, the most significant finding from this study is the *introduction of unpredictable environment-dependent errors* in Android 4.0. Fig. 4.2 shows that the second, third and fourth largest exception types in Android 4.0 are `android.view.WindowManager$BadTokenException` (26.83%), `java.lang.IllegalStateException` (23.56%), and `java.lang.RuntimeException` (3.12%). These exceptions are almost non-existent in Android 2.2. A dominant reason for these crashes was garbage collection, where resources allocated to activities were released—a severe side-effect being restart of the Android `system_server`. It was observed that the same fuzzed Intent sent to the same component at a different time point in the experiment did not always

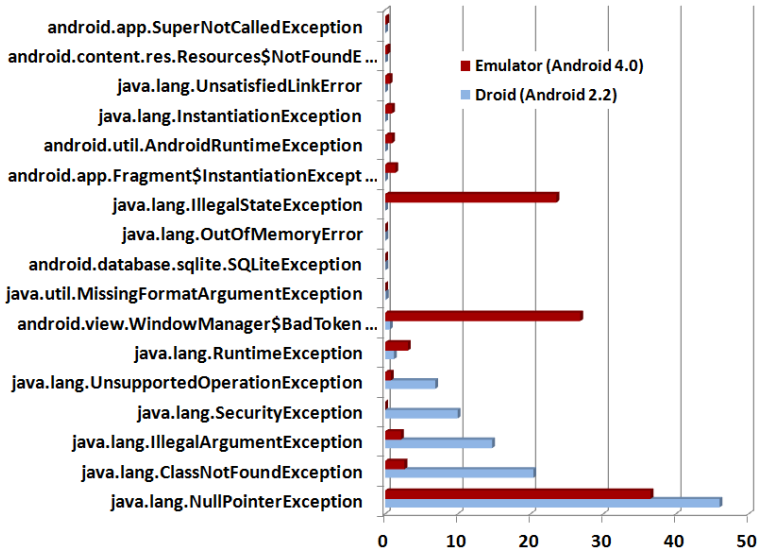


Fig. 4.2.: Distribution of different types of (uncaught) exceptions in Android 2.2 and 4.0. The bars represent percentage of all the exceptions, thus will sum to 100 (for each Android version). Note that we do not include Marketplace apps for this study.

cause the failure, or caused a different failure. The exact manifestation depended on the state of the device (the Emulator of the device to be more precise).

Another important point to note for Android 2.2 is the presence of exceptions that are typically thrown by the framework to notify the calling component of erroneous input or state, e.g., `java.lang.IllegalArgumentException`, `java.lang.SecurityException`, `java.lang.UnsupportedOperationException` etc. It is the responsibility of the calling function to implement proper exception handling, however such behavior is often missing in standard Android components.

System Crash from User Level Applications

Another significant discovery from our experiments was the *cascading failure of the Android runtime system*. We found a total of three Activities in the built-in applications that caused Android's `system_server` to restart. Due to the sensitive nature of these bugs and their potential security impact on millions of Droid users, we shall not disclose the names of the applications or the Activities in this forum.

```
.........................
I/ActivityManager(   62): Starting activity: Intent { act=ACTION_PACKAGE_DATA_CLEARED cmp=a
ndroid/.accounts.[redacted] }
W/dalvikvm(   62): threadid=7: thread exiting with uncaught exception (group=0x4001d800)
E/AndroidRuntime(   62): *** FATAL EXCEPTION IN SYSTEM PROCESS: android.server.ServerThread
.........................
E/AndroidRuntime(   62): Caused by: java.lang.NullPointerException
E/AndroidRuntime(   62):        at android.accounts.[redacted].onCr
eate([redacted].java:58)
E/AndroidRuntime(   62):        ... 6 more
I/Process (   62): Sending signal. PID: 62 SIG: 9
I/Zygote  (   33): Exit zygote because system server (62) has terminated
```

Fig. 4.3.: Partial stack trace of crash of ActivityX, which eventually causes the entire device to crash

Instead, we use the generic name ActivityX for purposes of explanation. All of the failures occurred due to `NullPointerException`s. Upon inspection of the configuration files of these activities, it was revealed that all these activities run under the "system" process of Android (i.e. `system_server`). When these activities tried to access some fields inside an Intent, they did not catch the `NullPointerException`, which crashed the current thread and eventually sent Signal 9 (SIGKILL) to Android `system_server`. A special concern is that to test these components JarJarBinks did not need any extra permission at install time. Thus, potentially, any user level application is capable of sending the malformed Intents to these vulnerable Activities, causing the entire device to crash. Such promiscuous use of privileged operations is a concern for millions of customers using Android 2.2/2.3 handsets.

```
.............................
57   final Bundle extras = getIntent().getExtras();
58   mAccount = extras.getParcelable(EXTRAS_ACCOUNT);
.............................
```

Fig. 4.4.: Code responsible for crash of ActivityX, which eventually causes the entire device to crash

Let us take a look at the stack trace for one of these crashes. This crash occurred when we sent an Intent `{act=ACTION_PACKAGE_DATA_CLEARED cmp= android/.ActivityX}` to the Activity. The stack trace for this crash (refer Fig. 4.3) showed an error at line 58 of the source

file `ActivityX.java`. The relevant code snippet is shown in Fig. 4.4. This code tries to read the extra field `EXTRAS_ACCOUNT`. However, since our Intent did not specify an `Extras` field, it raises a `NullPointerException`. This uncaught NPE kills the thread of this activity and eventually the process, which, in this case, is `system_server`. The problem can be avoided by verifying that the `extras` object in line 57 is not null before accessing it, or by handling the exception gracefully. The severity of this bug lies in its ability to crash Android `system_server`, in other words, to render the device unusable till the Android runtime is restarted.

4.3.2 Results for Implicit Intents

In experiment A, we sent implicit Intents that applications had opted in to receive but we left all unspecified fields blank, e.g., when a filter only restricts the `Action`, there is no `Category`, `Data`, or `Extras` field set. Overall, the HTC phone had 211 applications registered from which we could derive 1910 Intent-filters. For each Intent-filter, we sent out exactly one Intent matching the filter through `startActivity()`. Note that some of these Intent-filters are registered by Services, hence, sending a matching Intent through `startActivity()` simply results in an `ActivityNotFoundException`. Those Intents that were delivered to an application, crashed 5 of the recipients. 12 unexpected exceptions occurred during the experiment, which are exceptions other than `ActivityNotFound` or any flavor of security exception. Most frequent exception was once again the `NullPointerException` followed by `IOException` and `Resources$NotFoundException`. All three are the result of insufficient input validation either causing a missing value to get dereferenced (NPE) or, even worse, propagated as an argument to a IO or resource loading call. At the end of the experiment, the phone crashed with a system reboot in 50% of the cases due to cascading failures. Even though the number of failures is not large relative to the number of applications tested, it has to be pointed out that all Intents we sent are completely *valid* according to what a sender is able to find out through the

Intent-filters. *The problem arises from the fact that there is a significant amount of unspecified assumptions about the Intents that the receivers take for granted and fail to verify (e.g., a specific information in the* ***Extras*** *data being present).*

Experiment B goes a step further by combining all valid combination of `Action` and `Category`, thereby, significantly enlarging the number of Intents sent.

From the Intent-filters, we were able to derive 643 distinct `Actions` and 37 `Categories` that were used in at least one of the filters. For each application, we now generated all possible combinations of `Action` and `Category` that were valid according to the filter. The experiment consistently crashed the phone after 26 out of the 211 applications tested. This happened even though we set the delay between the Intents to 2 seconds to allow for manual interaction (e.g., closing dialog boxes) and thereby avoiding resource exhaustion.

From this small set of 26 tested applications, we observed 83 exceptions. The distribution of the specific exception types is shown in Table 4.2 with `NullPointerExcep-tion` and `IOException` again being the most frequent ones. Overall, 14 applications crashed during the experiment and showed a dialog to the user and only half of them were actually targeted directly, i.e., were the applications from which the filter was derived. The majority of the applications (including basic apps like Clock, Internet, Gallery, etc.) were most likely affected due to collateral failures, e.g., an Intent matching more than one filter and getting routed to more than one component.

4.3.3 Discussions

Our experiments have so far revealed three important aspects of Android—first is the presence of many components with poor exception handling code (most of these relate to `NullPointerExceptions`), second is the prevalence of environment-dependent errors in Android 4.0, and third is the presence of privileged components with unrestricted access. The first problem can be addressed by a methodical training of developers on good exception handling practices. Application developers should

Table 4.2.: Frequency Distribution of Crashes with Implicit Intents by Exception Type

Exception Type	**#Crashes**
NullPointerException	32
IOException	22
RuntimeException	13
ArrayIndexOutOfBoundsException	6
android.content.res.Resources$NotFoundException	4
ClassCastException	3
TimeoutException	1
com.sprint.internal.SystemPropertiesException	1
IllegalArgumentException	1

always check for exceptional conditions when dealing with inputs (Intents) from external sources. Resolution of the second and third problems need more work at the Android framework level. The third issue also exposes some potential problems with Android's default policy for process-assignment of an application component. At present a component X in application A can run in the process of application B if A and B are signed with the same developer key. Despite signature-based permissions, this may pose a problem for vendors that build custom ROMs. If a component (C) of this custom build is permitted to run as privileged process, it may wreak havoc like ActivityX in a similar fashion (note that component C and the kernel of this build are signed with the same key). A potential solution is to restrict accessibility of component C with an explicit permission, in other words, every component running in a privileged process must be protected by explicit permissions.

JJB Limitations: Apart from its handling of new tasks and alert dialogues (where a tester must manually close these), JJB has another limitation—it cannot distinguish between thread hang, resource exhaustion, and UI wait. Detecting thread hangs in response to a malformed Intent would require knowledge of a component's life cycle which is currently not visible in logs generated by `logcat`. Our future work would look into adding this capability in JJB.

4.4 Suggestions for Robust IPC

The key challenge in making Intents more robust is the lack of a formal schema. Intents are effectively untyped; their application-level type is only determined by a String identifier but is not reflected by the Java type system. Therefore, there is no explicit contract between a sender and a receiver of an Intent and mutual agreement is expected among the two about what format of data a specific Intent needs to have and what an invalid message is. Additional data is stored in a map-like data structure that is not fully type safe either. The data structure keeps separate key spaces for values of different types and provides typed methods for adding and retrieving data but it is again not formally specified what the expected additional values are and which type they are supposed to have. It is up to the author of the receiver code to perform the input validation, which is a repetitive and error-prone task. To make matters worse, primitive types are stored and retrieved as actual primitives, which means that in the absence of the value the result is the neutral element of the type, e.g., `false` in the case of a boolean value. The absence of a primitive value in the extra data is therefore not detectable by the receiver. Another problem arises from software evolution. Implicit message formats are hard to keep consistent across different versions of the applications, especially within an ecosystem where components are contributed by different sources. There is no way to version a specific Intent or to indicate compatibility between a sender and a receiver.

4.4.1 Subtyping/POJO Approach

One way to make the message format more explicit and therefore possible to capture for an automated message verification system is to use subclasses for Intents instead of a single flat type. Extra data belonging to a message would be expressed as fields of the subtype. In the spirit of Plain Old Java Objects (POJOs), there would be getters and setters for the field. As a side effect, the Java compiler can now do automatic type checking since the messages use a type schema that the compiler

is able to understand and enforce. What this approach does not achieve is further constraints on the values of data. For instance, there is no way to enforce a certain reference-type value to be not null or a numeric value to be always smaller than 10. Furthermore, there is currently no way in Java to express version information of classes in a standardized and accessible way. The cost for using the subtyping approach is that the total footprint of the platform is slightly increased since every Intent type now becomes a separate class in a separate file.

With a little experiment we found that a single class (subclass of Intent) with 3 fields (String, int, URL) having bean-like setters and getters adds 273 bytes to the footprint of an Android application, while the increase in size for a class with 6 fields is 403 bytes. Considering a handset where we have 200 Intent types, this implies a 80KB additional footprint for turning all these Intents into Subtypes with 6 fields. We argue that this is, in fact, an upper bound on footprint increase since we consider average 4-6 fields per Intent. In reality, most Intents have only between 2-3 fields, with few having a large number of fields (e.g., informative Intents like Battery Status).

4.4.2 Java Annotations

One way to express additional constraints about the message format when choosing the subtyping approach is the use of Java Annotations. Annotations are fully embedded into the language (since Java 1.5) and can be processed by the Java compiler. Therefore, it is possible to use the annotations already at compile time for criteria that are amenable to static checking. For dynamic checks, the corresponding code can either be realized as a common generic checker facility implemented as part of the Intent delivery mechanism of the platform or synthesized and injected into Intent receivers.

4.4.3 IDL and Domain Specific Language

Extended input validation requires additional knowledge about the message format since the semantic gap between the implicit message format and what can explicitly be expressed by classes and the Java type system is still large. For instance, an Intent responsible for a contact lookup might want to be able to do approximate matching and return the contact names together with a matching factor between zero and one. In the Java type system, it would have to use a float type for the latter data but thereby would extend the range of permitted values to the entire IEEE 754 floating point number range. Another example is the problem that every reference type can always be set to null so that there is no way to express mandatory data in messages. One way to more expressiveness is to use a domain specific language to express the schema of the Intents.

Historically, a similar approach has been taken with many RPC systems which used an interface definition language (IDL). This IDL describes exactly the format of a remote invocation in enough detail so that the stub and skeleton code can be synthesized from this description. Systems like CORBA extensively used IDLs but arguably also web services employ the same principle, e.g., through the WSDL files. For instance, a type system like XML Schema allows value restrictions and would be a viable candidate for a domain specific language approach to specifying Intents. A well-designed domain specific language can express any type of constraint and therefore permit full input validation including version checks.

There are two different possibilities to interface general-purpose languages with domain-specific languages. External DSLs are free-standing and independent of the host language. IDLs, for instance, are external DSLs. As a result, however, code written in the host language and the meta-data written in the DSL have to be developed independently and cannot easily be cross-validated by existing tools. Internal or embedded DSLs are themselves implemented in the host language and therefore

agree much better with existing tools. They are, however, restricted to what the host language can express.

4.5 Directions for Future Research

In this chapter, we presented the design and implementation of JarJarBinks (JJB), a tool for testing the robustness of Android's IPC primitives. With the help of JJB, we found a large number of exception handling bugs (some of which even crashed the runtime system) in various versions of Android. Despite its success JJB had few limitations—i) Since JJB is a user-space application, it could not automatically close alert dialogues or new processes. This made our testing process semi-manual. ii) JJB cannot distinguish between thread hang, resource exhaustion, and UI wait. Detecting thread hangs in response to a malformed Intent would require knowledge of a component's life cycle which is currently not visible in logs generated by logcat. Both of these may be solved by implementing JJB in the ActivityManager layer. However, such changes would require the users to have root access on their phones.

Based on these observations, we find several exciting research directions for improving smartphone dependability.

1. *Extending JJB by Instrumenting ActivityManager:* By moving parts of JJB in ActivityManager we can perform several new tests that were not possible with a user-space app. Examples are: testing for thread hangs and testing components that require authentication tokens (e.g. Facebook or Twitter apps in our experiments). By bypassing the authentication phases of these apps and intercepting network traffic, we can test these components more rigorously. However, the downside of moving JJB in ActivityManager is that it would require changes in firmware of the tested phones.

2. *Robustness of Mobile Payment Services:* In recent times, both Android and iPhone have introduced their mobile payment systems (Google Wallet and Apple Pay respectively). These present lucrative targets for hackers and malware

writers (note that owner of the phone himself may be the adversary against these services). Few existing research show security vulnerabilities of such payment APIs, however, the robustness of these applications and (cloud-based) services are not well understood. This present a promising direction for dependability research.

5. OVERVIEW OF CLOUD SERVICES

Cloud Computing, which started as a way to maximize server utilization and reduce IT costs, has evolved into a ground-breaking technology changing our everyday lives. Amazon Web Service got its start in 2006 and from then to now, it has grown to revenues of $2.25B in 2013 and store 2 trillion objects by Q2 of 2013 [39]. This statistic, although representative of the growth in the overall cloud adoption, only shows a small fraction of cloud utilization in the public domain. Benefits provided by cloud services such as massive scalability, on-demand provisioning (elasticity), and reduced operational cost have enticed small and large businesses alike.

Cloud computing provides high efficiency in part by multiplexing multiple customer workloads onto a single physical machine. Access to the CPU is given to multiple virtual machines (VMs) in the form of virtual CPUs (vcpu). To share compute resources fairly, each vcpu is given a pre-specified amount of scheduler credits. Similarly, physical memory (RAM) of the computer is also shared among VMs, typically in an on-demand basis. Often, physical resources of the servers are overcommitted (i.e. the sum of virtual resources assigned to VMs is greater than the physical resources) with the assumption that all VMs do not utilize their requested resources to the full extent simultaneously. Server Virtualization, the key enabling technology underlying infrastructure clouds (IaaS), has seen a staggering amount of research in recent years leading to many excellent algorithms for efficient sharing of cpu and memory [40, 41]. However, with this efficiency comes performance interference. When two VMs execute on the same physical machine, they not only contend for cpu and memory but also contend for low level hardware resources such as cache, memory bandwidth, io bandwidth etc.. Existing hardware and software virtualization mechanisms do not provide perfect isolation for sharing these resources resulting in serious concerns about unpredictable application performance in the cloud. Our work primarily deals with

such performance issues and helps improve application performance in the face of resource contention.

5.1 Overview of Cloud Dynamics

An IaaS cloud is a dynamic environment, where virtual machines are frequently created, cloned and deleted. Further, the resources allocated to virtual machines are transparently modified using VM live resizing and virtual machines migrated from one host to another using dynamic live migration [42]. In theory, virtualization promises perfect isolation between virtual machines. In practice, an application running in a VM can get impacted by cache contention [43] or network contention [44]. Page sharing mechanisms and lack of reservation for memory bandwidth may further lead to lack of isolation across virtual machines [45].

This lack of isolation combined with frequent VM resizings and VM live migration lead to frequent changes in the environment for a VM. Verma *et al.* [46] report that an average of 25% of all virtual machines may get migrated in a 2 hour period due to dynamic consolidation. The impact of co-located VMs and VM resource settings on an application is captured by operating context, first introduced in [47].

Definition 1 *An operating context of an application captures the impact on a virtual machine due to the environment outside the virtual machine. The operating context for a VM VM_i is defined as a 2-tuple consisting of the physical host $H(VM_i)$ and the set of co-located VMs VM_j on the host.*

Static environments have a fixed operating context, whereas VM creation, cloning, deletion, and live migration lead to frequent changes in the operating context for an application running in the cloud.

Similarly, we define the resource context of a VM as the resource assigned to the virtual machine.

Definition 2 *Resource Context for a virtual machine captures the resources assigned to a VM and consists of the CPU and memory entitled to the VM.*

A VM's resource context changes when it uses live resizing. This is usually more prevalent is private cloud platforms but is rarely seen in public clouds such as EC2.

5.2 Performance Interference in Cloud

Performance isolation in multicore systems has traditionally been a hard problem since partitioning low level hardware resources, such as cache and memory bandwidth is not only challenging but also has high overhead. For example, to partition cache or memory bandwidth, the controllers of these subsystems would need to maintain a list of all running VMs, their current allocations, and resource demands. Fundamentally, the cache and memory controllers are at a much lower level in the system stack than VMs and therefore, allowing visibility of VM resource issues to these controllers will need a profound re-design of the system stack. Such a solution will also reduce the overall utilization of the resources, which is particularly important considering that these resources are more scarce than the resources for which isolation works well. For example, the latest generation Intel i7 processor has 12 MB of L3 cache, compared to memories of several GBs which are standard on even low-end servers. Contention for these shared hardware resources can, therefore, lead to variable performance across VMs. We define performance interference in cloud as follows.

Definition 3 *Performance interference in cloud platforms is defined as the situation where performance of a VM (application response time or throughput) suffers due to the activity of other VM(s) co-located on the same physical machine.*

For example, if two VMs A and B are running on the same physical machine and A starts using a larger share of the memory bandwidth, then throughput of B may degrade. Note, that the cause of performance anomaly is external to the affected VM B and beyond its control. Performance interference can be considered as a special case of *change in operating context* (or effect thereof) defined earlier.

5.2.1 Effect of Interference

Several existing papers have highlighted the problem of interference due to contention of shared resources and their extent of performance degradation [13,48,49]. [13] reports that contention between two network intensive VMs can increase benchmark runtime upto 2x, while disk-disk and cache-cache contention can increase runtimes by 4.5x and 5.5x respectively. Another interesting observation here is that a network intensive VM can degrade performance of a cache intensive VM by upto 7x. We found similar results in our experiments, where a cache intensive benchmark can increase average response time of a web server from a fraction of a second (10^{-1}) to several seconds. End customers in today's commodity clouds have no way of efficiently dealing with such interference except hoping that the cloud providers detect such interference and take action. For example, Amazon EC2 does not allow customers to migrate their VMs to less busy servers. An alternative in EC2 is to run a VM on a dedicated host, thereby, nearly eliminating performance interference (note that storage and network contention may still exist) and side-channel attacks—but this comes at a significantly higher price.

5.2.2 Existing Solutions for Interference Mitigation

Existing solutions primarily try to solve the problem from the point of view of a cloud operator. The core techniques used by these solutions include a combination of one or more of the following: a) Scheduling, b) Live migration, c) Resource containment. Research on novel scheduling policies look at the problem at two abstraction levels. Cluster schedulers (consolidation managers) try to optimally place VMs on physical machines such that there is minimal resource contention among VMs on the same physical machine [16]. For example, it will try to schedule two VMs that are running network-hungry applications on different physical machines. Novel hypervisor schedulers [14] try to schedule VM threads so that only non-contending threads run in parallel. An example may be the following execution schedule: a cpu intensive

VM can be running parallely with an IO intensive VM, whereas two IO intensive VMs are run at different time slices. Another approach in scheduling is to allocate extra time slices to suffering VMs to minimize lost work [17]. Live migration involves moving a VM from a busy physical machine to a free machine when interference is detected [15]. Resource containment is generally applicable to containers such as LXC, where the cpu cycles allocated to batch jobs is reduced when interference is detected [50, 51].

However, all these approaches have their shortcomings. Firstly, a VM's resource usage pattern may change over time, often unpredictably. A consolidation manager (cluster scheduler) cannot foresee such usage changes without knowing of the applications running within the VM, and that is usually considered too intrusive and hence, not made available to the consolidation manager. Secondly, all these approaches require access to the hypervisor (or kernel in case of LXC), which is beyond the scope of a cloud consumer. These solutions need to be incorporated by cloud providers in their virtualization infrastructure. We found that, despite having (arguably the best of) schedulers, the public cloud service, Amazon Web Service, shows significant amount of interference as we show in Section 6.2.

Solutions that use live migration for avoiding interference have their drawbacks as well. In [46], the authors show that VM live migration is very resource intensive, especially when the source server is highly loaded. Live migration in such a scenario is often long drawn and fails frequently. Further, it significantly impacts application performance during the migration. So, it is not suitable to deal with short-lived interference, which we observe is prevalent in EC2.

We therefore need to find practical solutions that do not require modification of the hypervisor. We observe that, to improve application performance during interference, we must reduce contention for the shared resources. One approach to achieve this would be to reconfigure the affected application in a manner that reduces its load. We found that standard web servers and application servers have several configuration parameters that allow an administrator to control its load. We use these parameters

to build our interference mitigation framework. In the next two chapters, we show that this is indeed a practical solution with promising results.

5.3 Web Services in Cloud

In this dissertation, we focus on improving the performance of cloud applications that are latency sensitive. More specifically, we consider the response times of web service applications. We selected web services as our preferred application domain for two primary reasons. First, web applications constitute a large portion of cloud workload. [52] reports that nearly 25% of all ip-addresses in Amazon EC2 host a public website. Our interference-aware configuration manager has the potential to reduce a large fraction of performance anomalies of these websites in the face of interference. Second, web applications and middleware components typically have a large number of tunable parameters with known performance benefits. For example, 2 of the parameters (`MaxClients`[1] and `KeepaliveTimeout`) that we use for reconfiguration in IC^2, has been used by researchers [20] for tuning Apache in other contexts.

5.3.1 Web Application Configurations

Enterprise web applications typically follow a three-tier architecture consisting of a Front-end Server, an Application Server, and a Database Server. The front end server receives web requests from clients, while the application server processes the request, fetches results from the database and sends the desired responses back. We use the Apache Httpd server as our front-end server and Php runtime engine (Php-fpm) as our application server. In this dissertation, we argue that optimal configuration parameters of these applications and middleware depend on the operating context in which they are running.

Apache web server (HTTP server version 2.4) has 258 configuration directives (Only counting the directives in the core, the base, and the multi-threaded module.)

[1]These are concurrency and timeout related parameters in Apache httpd server.

Adjusting these configuration parameters in a non-virtualized environment is difficult enough—for example, the configuration may need to change with the type of workload and the intensity of the workload. This task becomes even more challenging when the reconfiguration has to be done at runtime in response to changes in *other customer's applications* over which the service consumer has no control. We next highlight the configuration parameters that are adjusted by our interference mitigation engine. It may be noted that the parameters we select are generic thread-pool management parameters and are applicable to a wide variety of multi-threaded enterprise middleware.

`MaxClients`: `MaxClients` (MXC) captures the maximum number of parallel threads the web server employs to serve requests. This is typically configured based on the workload intensity, number of hardware threads available on the physical server, and its RAM capacity. Virtualization replaces hardware threads by virtualized cpus (vcpu), and provisions memory dynamically based on VM activity. Presence of co-located VMs may change actual number of hardware threads or physical memory available to a web server impacting performance.

`KeepaliveTimeout`: `KeepaliveTimeout` (KAT) indicates how long a web server would keep an idle client connection in its connection pool (typically occupying a thread). `KeepaliveTimeout` is known to be sensitive to the nature of web application and users' browsing patterns. We found it is sensitive to both operating context and resource context changes.

`PhpMaxChildren`: We refer to `pm.max_children` parameter of Php fastcgi process manager (`php-fpm`) as `PhpMaxChildren` (PHP) in this dissertation. This defines the maximum number of threads used by the Php interpreter. `PhpMaxChildren` generally has high sensitivity to resource context and low sensitivity to operating context.

Our interference mitigation tools (IC^2 and ICE) update these three parameters autonomously to maintain stable application performance.

5.4 Overview of Load Balancers

Load balancers are used to scale websites to serve larger number of users by distributing load among multiple servers. Existing research on load balancing can be primarily categorized into two orthogonal problems: i) **Routing:** How to redirect or route traffic for a web server to its internal "real" servers *transparently* and *efficiently*? ii) **Scheduling:** How to decide which server would get the current request so that the servers have equal load?

There are many load balancing solutions that are available in the market. A majority of these operate at the network layer and tries to balance network load between servers. Both IBM and Cisco have commercially available hardware load balancers that provide high availability and good performance. Among the open source software load balancers, HAProxy [53] and Linux Virtual Server (LVS) [54] are popular choices. These two load balancers have complementary features which are good for different situations. While Linux Virtual Server operate at layer 4 of the network stack (transport), HAProxy operate at layer 7 (i.e. application). LVS is therefore good at network load balancing, whereas, HAProxy can understand http requests and balance application sessions across servers. Another key difference between the two is that LVS operate at Kernel level (by using custom modules that have been integrated with Linux mainline), but HAProxy operate at application level. Both these load balancers implement four major scheduling policies. These are:

Round Robin (RR): Each server is picked in a circular order.

Least Connection (LC): Server with the least number of active connections are chosen. This is better than round robin in general since it considers server state ($num\_connections$).

Weighted Round Robin (WRR): This policy helps when servers have unequal capacity. The scheduler picks a server based on its weight. For example if servers $S_1, .., S_n$ have weights $w_1, .., w_n$, then S_k is chosen if $w_k = max(w_1, .., w_n)$. Note that weight w_k is reduced every time a server is selected, hence it keeps track of how many

requests a given server has served so far. When all the servers have weight 0 they are reset to their initial values.

Weighted Least Connection (WLC): Similar to LC, but server weight is also considered for scheduling decision. If W_i is the weight of server i and C_i is the current number of connections to it, then the scheduler picks server j for the next request such that, $C_j/W_j = min\{C_i/W_i\}$, i=$[1, .., n]$.

In rest of the paper, we use the terms RR and WRR (LC and WLC) synonymously since the concept of weight is often implicit. Apart from these there are also load balancing policies that are based on source address of a client. In this case, an appropriate hash function is used to map the source address to a server. Often load balancing policies also consider geographical location of a client and forwards the request to the server farm that is closest to the client location. Such policies are typically implemented by a DNS based load balancing, where the same domain name maps to different IP addresses.

Once a client session has been established, the load balancer remembers which server is processing requests originating from that client. In HAProxy, the session mapping is performed by inserting a cookie $SERVER\_ID$ in the HTTP header. When HAProxy finds the cookie in a new request, it directly forwards the request to that server instead of computing a destination. In LVS, a connection table is maintained which maps a given client to a server. If an entry exists in this table, LVS uses the same server for sending subsequent requests. Note that this is required since most web servers maintain sessions and relevant application state locally.

6. MITIGATING INTERFERENCE USING MIDDLEWARE RECONFIGURATION

Application performance has been and remains one of top five concerns since the inception of cloud computing. A primary determinant of application performance is multi-tenancy or sharing of hardware resources in clouds. While some hardware resources can be partitioned well among VMs (such as CPUs), many others cannot (such as memory bandwidth). In this chapter, we focus on understanding the variability in application performance on a cloud and explore ways for an end customer to deal with it. Based on rigorous experiments using CloudSuite, a popular Web2.0 benchmark, running on EC2, we found that interference-induced performance degradation is a reality. On a private cloud testbed, we also observed that interference impacts the choice of best configuration values for applications and middleware. We posit that intelligent reconfiguration of application parameters presents a way for an end customer to reduce the impact of interference. However, tuning the application to deal with interference is challenging because of two fundamental reasons — the configuration depends on the nature and degree of interference and there are inter-parameter dependencies. We design and implement the IC^2 system (Interference-aware Cloud application Configuration) to address the challenges of detection and mitigation of performance interference in clouds. Compared to an interference-agnostic configuration, the proposed solution provides up to 29% and 40% improvement in average response time on EC2 and a private cloud testbed respectively.

6.1 Motivation

In the brief history of cloud computing, unpredictable application performance has been one of the two key issues preventing widespread adoption of the cloud paradigm.

In a recent survey of IT buyers, about 40% cited application performance as a key concern [55]. Operational support for critical applications was another key concern with 40% IT buyers, making performance-related issues two of the top five concerns for cloud customers.

6.1.1 The Problem

Performance issues in the cloud are often attributed to misconfigurations of virtual machines (VMs), storage, and networks [47, 56, 57]. Another key reason for performance issues, which has not received adequate attention, is imperfect isolation of hardware resources across multiple VMs. Some resources, such as CPU and memory can be partitioned among VMs with little interference. However, current hypervisors do not isolate low level hardware resources, such as cache and memory bandwidth. Contention for these shared hardware resources leads to variable performance across VMs—a situation commonly referred to as performance interference (or interference in short). In the previous chapter, we presented a formal definition of interference and showed how much performance degradation can be seen during interference (refer Section 5.2). Partitioning low-level hardware resources in software (hypervisor) will introduce significant overheads and we do not envision that processor caches and memory bandwidth will be isolated on a per-VM basis in the foreseeable future. We, therefore, need practical solutions to deal with interferences when such situations arise.

6.1.2 Existing Solutions

Existing work on handling interference in clouds are driven primarily from a private cloud perspective. The key idea is to either schedule interfering VMs at different points in time on the same host (e.g., [14]) or place interfering VMs on different hosts (e.g., [15, 16]). The first approach is limited in terms of the choices of VMs available on a host that can be co-scheduled. The second approach requires frequent live mi-

grations, which is very resource intensive, especially when the source server is highly loaded [46]. Live migration in such a scenario is often long drawn and fails frequently. Further, it significantly impacts application performance during the migration. So, it is not suitable to deal with short-lived interference, which we observe is prevalent in EC2. Finally, these approaches are application-oblivious and cannot accurately judge the real impact on application performance.

6.1.3 Our Solution Approach

In this chapter, we present a complementary approach of handling interference by application reconfiguration. We argue that an application can mitigate the ill effects of short-term interference — rise in response time and drop in throughput — by deploying an intelligent configuration manager. This configuration manager continuously monitors for interference and when it is observed, reconfigures the application and/or middleware (e.g., web server, database) to reduce contention for the bottlenecked resources. Our solution gives power in the hands of the application owners, and does not rely on the infrastructure provider making prompt changes to help the application with its periods of interference. This is also important because interferences in public clouds are often short-lived, less than a minute, and therefore application reconfiguration, which can be more agile than infrastructure reconfiguration, is particularly well suited.

In this chapter, we make the following key contributions:

1. We rigorously study the performance variability of web-based applications in a public cloud environment. In this study, we run the CloudSuite [19] benchmark in Amazon's EC2 for 100 hours over a 5-day period. We then compare the statistics obtained from these runs with sample runs of CloudSuite in a private cloud testbed. We observe that CloudSuite has much longer response time distribution in EC2 (ranging up to 55X of median) than in the local testbed (up to 4X of median) with identical resource configurations. This validates our hypothesis, that public clouds have high

degree of performance uncertainty.

2. We conduct a study to understand if applications can be configured to deal with interference. We observed that an ideal operating configuration for Apache web server depends on the type and degree of interference. Further, parameters in different elements of the software stack depend on each other and the inter-dependency changes with the degree of interference; and finally, the application performance curves with the configuration values are discontinuous in places, making traditional control-theoretic approaches for parameter tuning [20] ineffective. Specifically we found three parameters corresponding to the degree of concurrency and the time to live of existing connections to be particularly significant.

3. We present a *simple, heuristic-driven* configuration manager, IC^2, to reconfigure the application upon interference. IC^2 solves three key challenges for dynamic reconfiguration—first, it presents a machine learning based technique for detecting interference; second, it uses a heuristic-based controller for determining suitable parameter values during periods of interference; and finally, it reduces the cost of reconfiguration of standard Apache distributions by implementing an online reconfiguration option in the Httpd server. A prototype implementation of IC^2 was deployed both in EC2 and our private testbed. The experiments show that IC^2can recapture lost response time by up to 29% in EC2 and 40% in our private testbed.

The rest of the chapter is organized as follows. In Section 6.2, we verify the presence of interference in cloud platforms and present a quantitative evaluation of performance degradation. We next show the impact of interference in a private cloud testbed and identify how interference changes optimal configuration values for applications. The design of our proposed solution and performance improvement achieved by it are highlighted in Sections 6.4 and 6.5. Finally, we conclude the chapter by discussing few shortcomings of the proposed solution and highlight future research directions.

6.2 Is Interference Real?

We performed an experimental study to see if the performance concerns due to interference are real. Our objectives here are to answer two questions: i) Does an application suffer from unpredictable latencies in EC2? ii) What happens when a co-located VM starts accessing memory very fast? To answer the first question, we ran an application benchmark on Amazon EC2 with a constant workload setting and collected periodic performance data over 100-hours. We then analyzed the collected data to detect outliers and see how much performance variability there is. The application benchmark we selected for our experiments is CloudSuite, a popular web application benchmark [19]. CloudSuite internally uses Olio, a social event calendar application as the base package[1]. The web server and database of CloudSuite were installed on separate EC2 VMs each of type m1.large instances (equivalent to 2 vcpus or 4 EC2 compute units, and 7.5GB memory).

Observations. We see that as a result of interference, there is significant variance in the performance of Olio on EC2 with regard to the response time (Fig. 6.1(a)), and correspondingly, the throughput. The median response times in EC2 and the private testbed were found to be 0.10s and 0.12s respectively. The histogram in Fig. 6.1(a) is plotted such that the value represents response time between the two marks on the X-axis, e.g., there are 539 measurement intervals with response times (RT) between 0.5 and 1 sec (which is more than 5X of median RT). In contrast, for a similar experiment on local testbed (measurements taken over 60 hours) we found the response time was always < 0.5s (i.e. < 4X of median). The response time distribution in EC2 has a much longer tail indicating periods of unpredictable performance. In EC2, we also measured the duration of interference using an outlier detection method as shown in Equation 6.1. Our results indicate that there are several instances when interference lasted for 30s or longer, the longest duration being 140s. While these interference instances are a small portion of the total number of requests, there are two condi-

[1]In this dissertation, we use the terms "CloudSuite," "Olio," and "Application benchmark" interchangeably.

tions that suggest we need to deal with them—they are unpredictable and therefore, worst-case provisioning for performance critical applications suggests we must put in place mechanisms to deal with them; when interferences do happen, they cause pathologically poor behavior of the application and may push the application into a "death spiral". Evidence of death spiral in applications due to transient degradation has been given in the past, such as, due to overfilling of application queues [58].

$$| P_i - P_{N/2} | > C \times median(| \{P_i\}_{i=1}^{N} - P_{N/2} |) \tag{6.1}$$

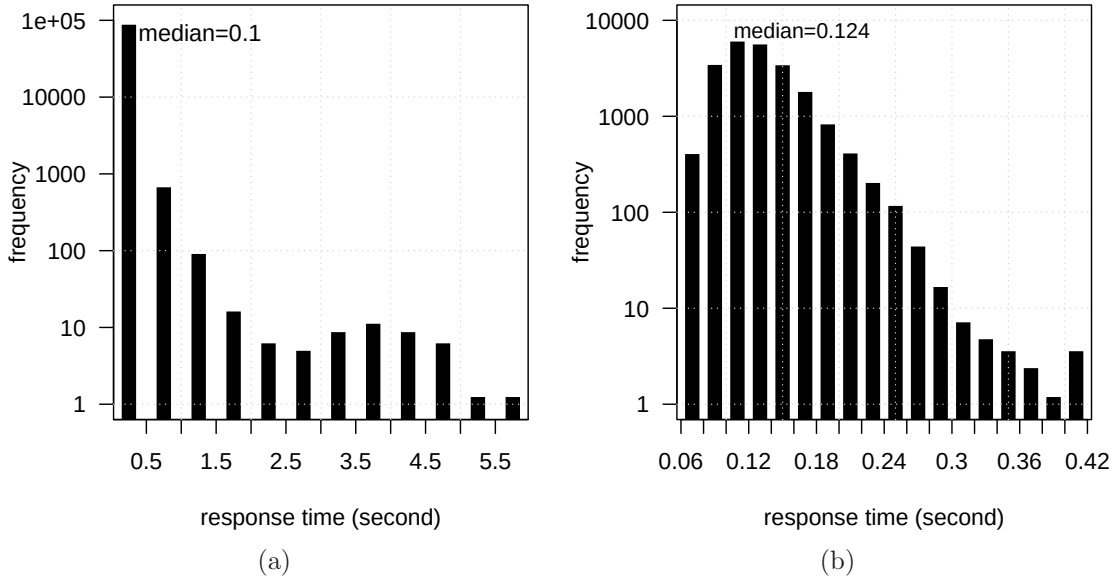


Fig. 6.1.: Distribution of response times of Olio running on (a) Amazon's EC2 (b) Private cloud. VM resource settings and workload intensity are identical in both cases. The longer tail in EC2 (ranging up to 55X of median in EC2 compared to 4X in private cloud) indicate presence of interference.

To answer the second question raised earlier, we ran another set of experiments both on EC2 and private cloud testbed. The results indicate cache-intensive interference from co-located VMs can increase response time of a web server by an order of magnitude. Our experiments in following sections substantiate this point.

6.3 Interference Impacts Optimal Configuration Values

In this section, we endeavor to understand the relationship between optimal configuration values of middlewares and interference. To do so, we ran an extensive set of experiments with Cloudsuite in a private cloud testbed. We first describe our experimental setup and then highlight our findings.

6.3.1 Experimental Setup

Hardware. Our private cloud testbed consisted of three Poweredge 320 servers with Intel Xeon E5-2440 processors. Each server has 6 cores (or 12 hardware threads with hyperthreading enabled), 15 MB cache and 16 GB memory. We installed the KVM hypervisor on these machines. We co-located our custom interference VMs on the same host as the web server, while database VM was run on a separate physical machine. In this work we focus only on web server performance and over-provisioned the DB VM to eliminate any DB bottleneck. The database (approx. 1.6GB) was also loaded in memory to reduce disk contention. The third machine of our setup was used to run the benchmark driver and rest of the client emulators. All the computers were connected via a dedicated 1 Gbps switch. Table 6.1 lists the values of different configurations for each experiment presented in this chapter. It is to be noted that, we never created contention for CPU and memory on the physical server. With two interference threads running and 4 vcpus for WS, the physical server's CPU utilization was at the 50% mark or lower for all experiments. Similarly, memory utilization of the host was never an issue. Our maximum WS memory utilization was well below 3GB for all workloads.

Application Benchmark. The application benchmark we selected for our experiments is CloudSuite, a popular web application benchmark [19]. In our setup, we hosted Cloudsuite on a multi-threaded Apache server (*apache-worker* v2.4) and used Php Fastcgi Process Manager (*php-fpm*) for dynamic content generation. Our

setup closely resembles a typical three-tier application with *php-fpm* v5.3 as the business logic (BL) tier. We use identical CloudSuite setup in all our experiments—homogeneous VMs with Ubuntu 12.04/Apache 2.4/php-fpm 5.3/Java 1.7. CloudSuite uses the Faban harness to emulate clients. Client emulation is done using a pre-defined distribution (negative exponential) of think times and operation mixes as defined in [59]. Workload size is given in terms of `#concurrent_clients`.

Interfering Application. We emulated interference from co-located VMs by running two different benchmarks–LLC-Probe and Dcopy–on two VMs (also referred to as interference VMs). Dcopy is an application under the BLAS [60] benchmark suite, which copies contents of a source array to a destination array. LLCProbe [13] creates an LLC (Last Level Cache) sized array in memory and then accesses each cache line very frequently. Both Dcopy and LLCProbe are cache intensive, however, rate of cache access is higher in LLCProbe than in Dcopy. Moreover, by using Dcopy with a large array size we can also emulate memory bandwidth contention. Interferences of this type may arise in reality if a co-located VM runs data mining applications like Hadoop or even under periodic consolidation operations. Earlier work has shown [13, 46] that such interferences are a routine occurrence in present-day cloud infrastructures.

Parameter Selection. In our experiments and subsequent evaluations, we consider three key configuration parameters – `MaxClients` (MXC) and `KeepaliveTimeout` (KAT) from Apache web server[2] and `pm.max_children` from Php runtime. These parameters greatly impact Apache's web application performance [20].

`MaxClients` captures the maximum number of parallel threads the web server employs to serve requests. This is typically configured based on the workload intensity, number of hardware threads available on the physical server, and its RAM capacity.

`KeepaliveTimeout` indicates how long a web server would keep an idle client connection in its connection pool (typically occupying a thread).

[2]We use the terms Apache and Httpd synonymously to identify the Apache web server.

`pm.max_children` defines the maximum number of threads used by the Php interpreter. We refer to `pm.max_children` as `PhpMaxChildren` in this dissertation.

It is pertinent to note that these parameters are generic thread-pool management parameters and have their counterparts in most commercial server distributions making our study applicable to most enterprise middleware (e.g. thread pool size in Glassfish).

Metric Collection. CloudSuite (Olio) uses Faban harness to emulate clients and generates high level benchmark metrics for each run (Response Time and Throughput). For each data point in our plots (i.e. a given setting of configuration values or workload) we consider an average of three runs. Each run lasted for 10 minutes (excluding ramp-down) of which last 5 minutes were considered as steady state and reported. The experimental VMs were rebooted after each run to clear any state. For monitoring hardware performance counters we started oprofile [61], a low overhead profiler, on the hypervisor of the web server VM. We use the observation that a guest VM in KVM is represented as a qemu process in the hypervisor. We used oprofile to monitor the hardware events corresponding to the qemu process of the WS VM. We next report some of our key experimental results.

6.3.2 Impact of Interference on Middleware Configurations

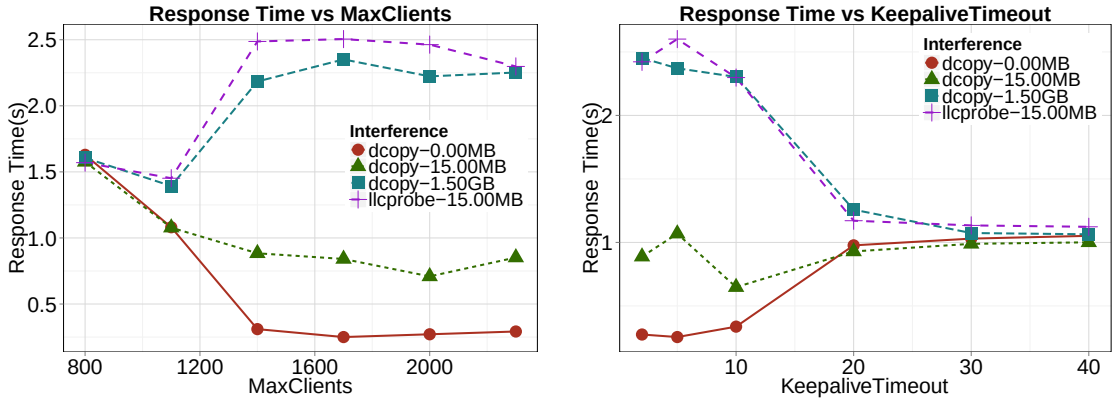
In this experiment, we evaluate the impact of interference on the choice of optimal values for the three parameters—`MaxClients`, `KeepaliveTimeout` in Apache and `PhpMaxChildren` in Php-fpm. For each of these parameters, we ran the web server with different interference intensity - LLCProbe with array size of 15MB and Dcopy with array sizes 15MB and 1.5GB. Due to its fast cache access, LLCProbe emulates a strong interference, while Dcopy 15MB emulates a low interference. With Dcopy size of 1.5GB we emulate contention for both cache and memory bandwidth, and its overall effect is that of a moderate interference. Here a Dcopy size of 0.0MB implies

Table 6.1.: Summary of WS VM config. and parameters during different experiments. Values with asterisk(*) are reconfigured with IC^2.

Experiment	**# Vcpus**	**Memory(GB)**	`MaxClients`	`KeepaliveTimeout`	`PhpMaxChildren`	**Load Size**
Sec. 6.3.2	4	4.5	Variable	5	1000	1500
Sec. 6.3.2	4	4.5	1700	Variable	1000	1500
Sec. 6.3.3	4	4.5	Variable	Variable	1000	1500
Sec. 6.2,6.5	2	7.5	650*	5*	50*	550

a run where no interference benchmark was run (baseline). For each interference intensity, we varied one parameter of Apache while the other was set to an observed good value. Run configurations for each experiment can be seen from Table 6.1. For all the experiments, we kept the workload intensity (`#concurrent_clients`) to a fixed value of 1500 which was found to be lower than the saturation point of the web server[3]. Note that although we have a constant number of concurrent clients, Faban may generate bursty traffic in some intervals due to is stochastic "wait time".

Effect on `MaxClients`



(a) Response Time vs. MXC

(b) Response Time vs. KAT

Fig. 6.2.: Choice of optimal parameter values with varying Dcopy and LLCProbe. For all experiments, `#concurrent_clients` is 1500, chosen default values are $MXC = 1700$, $KAT = 5$, and $PHP = 1000$. In each experiment, one of the parameters are varied while others are kept constant at their default values.

Figure 6.2(a) show the choice of optimal `MaxClients` (MXC) values for different interference intensities. In the baseline case (Dcopy 0MB), best response time can be obtained by setting MXC to 1700. However, the optimal value reduces to 1100 for Dcopy-1.5GB and LLC-15MB. Interestingly, with a smaller interference of Dcopy-15MB, the optimal value increased to 2000 (although the gain in response time was

[3]We define saturation point to be the minimum workload intensity when the web server exhausts its cpu or memory capacity.

small compared to at 1700). Even though all the curves show concave nature before saturation, they diverge from each other significantly clearly highlighting a change in the operating environment.

It can also be seen that interference causes the response time of WS to go up from fraction of a second (< 0.5s) to several seconds. If we keep MXC constant at the baseline optimal value of 1700 (refer Fig. 6.2(a)), with LLCProbe it increases up to 2.5s. However, with a different MXC value (1100), this degradation can be limited to only 1.5s. One may argue that we can always keep MXC fixed at 1100, but this wastes server resources (e.g. baseline throughput at 1100 is 13% lower than that at 1700). A better alternative is to configure it for the dominant case (no-interference) and to reconfigure when interference is detected.

Effect on `KeepaliveTimeout`

We found similar results for variable `KeepaliveTimeout` (KAT) which suggests different optimal KAT values for varying interference intensity (refer Fig. 6.2(b)). For this experiment, we kept the MXC value fixed at the optimal baseline MXC value of 1700 and varied KAT from 2 to 40 seconds. The curves show very different patterns with varying interference intensity. In the baseline case, increasing KAT beyond 5s increases response time. On the other hand, with strong interferences increasing KAT reduces response time significantly. Based on this, one may argue that we can always keep KAT fixed at a high value (e.g. 20). We see from the plot that such a choice is suboptimal for no-interference; it also shows poor throughput. As a general rule we found that interference from co-located VMs increases the optimal KAT value. This emphasizes that a web application needs to reconfigure its KAT value in the presence of interference and finding the optimal is a non-trivial problem.

Due to space limitations we only present the key findings of varying `PhpMaxChildren` (PHP). In general, increasing PHP had almost no impact in no-interference response time. We therefore choose a low PHP value (100) with lower memory footprint

as the no-interference optima. With interference, optimal response time is seen for `PhpMaxChildren`= 800 or higher although the performance improvement is smaller compared to `MaxClients`. Interested readers may find the details in [62].

Table 6.2 presents a summary of our observations about optimality of parameters and the relationship with interference. Each cell in this table summarizes the impact of interference on the optima of a given parameter (whether it increases or decreases) and the degree of impact this parameter has on performance (high or low).

Table 6.2.: Summary of our experiments on evaluating the impact of interference on optimal parameter values.

Application runtime	Apache and Php		
Operating context changes	Cache, memory bandwidth pressure		
General impact on optimal configuration values			
Context	`MXC`	`KAT`	`PMC`
No interference Initial value	High	Low	Low
With Interference Performance Impact	Decrease High	Increase High	Increase Low
Memory Pressure Performance Impact	Decrease High	Increase High	Decrease High

6.3.3 Change in Inter-parameter Dependency

In this section, we answer a commonly asked question on configuration management– are two parameters independent? We verify this with the specific example of two parameters that had the most profound effect on the performance of our benchmark applications, namely, `MaxClients` and `KeepaliveTimeout`. We find that dependency does exist between these parameters and it changes with interference.

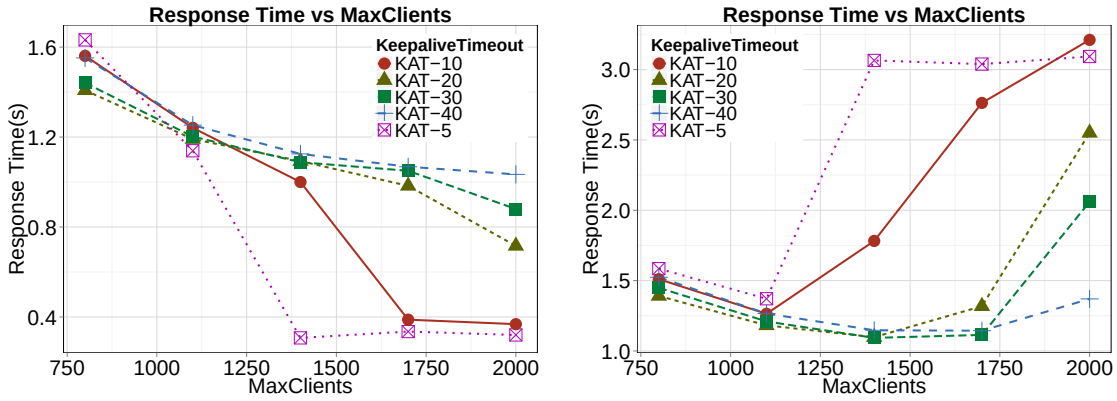
For this experiment, we varied both MXC and KAT for the Apache server under two scenarios. In the first, we ran the web server with no interference, while in the second, we ran it with LLCProbe-15MB. We found that the nature of curves changed significantly across these experiments (refer figures 6.3(a) and 6.3(b)). For the case with no interference, the curves generally have a negative slope, while with

interference, the curves display both positive and negative slopes. Choice of optimal KAT for a given MXC is significantly different in the two. As a general observation, we find that lower KAT is better at baseline while higher KAT is better during interference.

One may argue that the following simple equation suffices to determine KAT value for a given MXC:

$\texttt{KAT} = \texttt{MXC}/\texttt{\#new\_connections/sec}$

However, this does not work well during interference. For example, during interference, if we reduce MXC then according to this formula we should also reduce KAT to maintain a constant connection rate. But such an action would further increase load on the server. Due to shorter KAT, a larger fraction of established client connections would time out, necessitating new connection establishment. A better alternative is to be aware of interference and select a different value for `#new_connections`. This emphasizes the need for the tuning algorithm to be context aware. Depending on the presence or absence of interference it must select a different optimal KAT value for a given MXC.



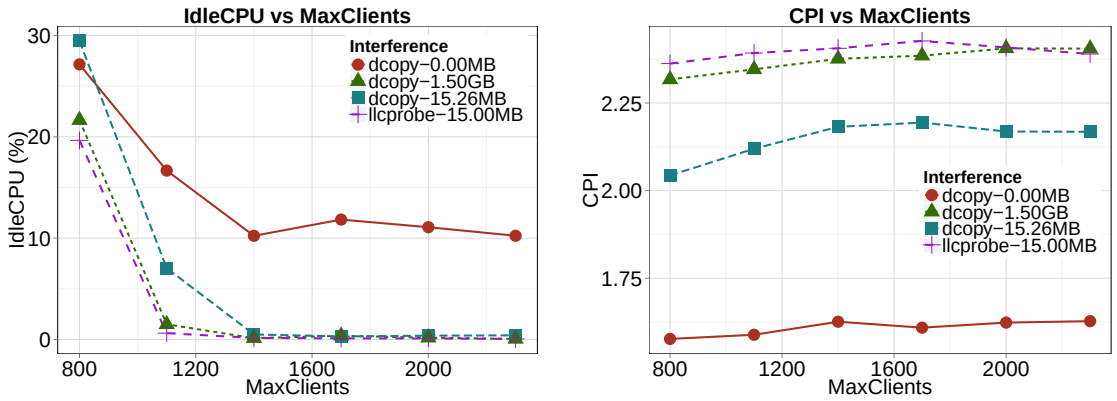
(a) No interference

(b) With LLCProbe

Fig. 6.3.: (a–b) Response Time vs. MXC with varying KAT. Dependency between `MaxClients` and `KeepaliveTimeout` changes with interference.

6.3.4 Interference and Web Server Capacity

In the previous section, we found that interference has a significant impact on the response time of a web server. In this section, we ask ourselves what is the root cause for such increase? To answer this question, we analyzed the system metrics obtained from the previous experiments (Section 6.3.2). We also evaluated the impact of interference with varying workload sizes. Our observations are presented below.



(a) Rise in CPU utilization

(b) CPI of WS-VM

Fig. 6.4.: Effects of interference. Here we identify system level bottlenecks that causes response time to increase by an order.

Interference increases CPU utilization of WS-VM. We found that for a memory allocation of 4.5GB, the WS was never constrained for memory. But the cpu utilization (Fig. 6.4(a)) values showed significant bottleneck. It can be seen that for a given choice of `MaxClients`, the IdleCPU values for with-interference curves are lower than baseline. Note that the IdleCPU values are virtual utilization measured inside the WS VM. Intuitively, for a constant workload this should remain fixed irrespective of the functioning of a co-resident VM. To understand this behavior, we measured the CPI (cycles per instruction retired) values for the WS VM with varying degrees of interference. Due to the large number of cache misses induced by the interference VM, the WS VM uses more of its cpu cycles fetching data from memory to cache and consequently the CPI increases. It can be seen from Fig. 6.4(b) that

the CPI values for the WS with interference is between 2 and 2.25, whereas, baseline CPI is only 1.5. It implies that, on average, a WS thread takes longer time to finish execution. The overall effect is that a larger fraction of the WS VM's time slice is occupied by some busy thread. This is reflected as increased cpu utilization inside the guest VM.

Interference increases active memory of WS-VM. Similar to Section 6.3.4, we found active memory of the web server increased during interference. This happens since with interference, Apache threads are active for a longer duration on average (higher response time). Note that an active Apache thread has larger memory footprint than in an idle one (in Apache terminology an active thread includes the `request pool`, a large block of memory for storing the request and response data, in addition to `server` and `configuration pools`), therefore longer response time implies increased active memory. We found that this observation becomes even more significant if the web server is under memory pressure. In such a case, if a web server's memory footprint is just below capacity in a baseline case, with interference it is likely to start swapping. This again has catastrophic impact on performance. We verified this hypothesis by running the WS VM with 2GB RAM in a separate experiment and found evidence of swapping with interference even though no swapping happened in a baseline run. Details of this experiment may be found in [62]. Based on Fig. 6.4, we conclude *interference reduces the capacity of a web server*.

6.4 Design and Implementation

In the previous section, we found that the choice of optimal configuration values for web services middleware depend significantly on interference created by co-located VMs. Here we propose the design of a configuration manager that is aware of the *operating context* [47] of the web server VM. Although most of our implementation focuses on mitigating impact of cache-intensive interference, the same principles can be applied for other types of interferences (e.g. network). To design an interference-

aware configuration manager we need to answer three important questions:
i) How do we detect a web server is suffering from interference?
ii) Which parameters can be configured to mitigate interference?
iii) For the parameters determined in step (ii), how should their values be set as a function of the degree of interference?
We answer each of these questions in rest of this section.

Fig. 6.5 presents a high level system architecture of the proposed solution. IC^2 consists of two primary modules: a) *Performance monitor*, and b) *Config manager*. For all the VMs that are part of a web application (e.g. web server, database and mail server) and managed by IC^2, performance monitor collects performance data at three levels. At the application level it collects aggregate response time and throughput measurements, whereas at system level, it collects utilization values for CPU, memory, IO, and network. If hardware performance counters are available (on our local testbed, but not on AWS), it also collects CPI (cycles per instruction) and CMR (last level cache-miss rate) data for the monitored VMs. Based on the collected data, config manager can detect if any system context has changed. This can either be a change in workload, VM resource allocation or presence of interference. There are several existing solutions that can handle workload and resource changes [20, 63] and these can run concurrently with our solution.

A high level functioning of IC^2 is shown in Fig. 6.6. After collecting metrics, IC^2 tries to detect if the web server is under interference. Based on the detection result it maintains a state machine for the web server. The state machine, in turn, is used to decide when reconfigurations are needed. Finally, the config controller actuates the reconfiguration action. Details of this configuration loop is presented below.

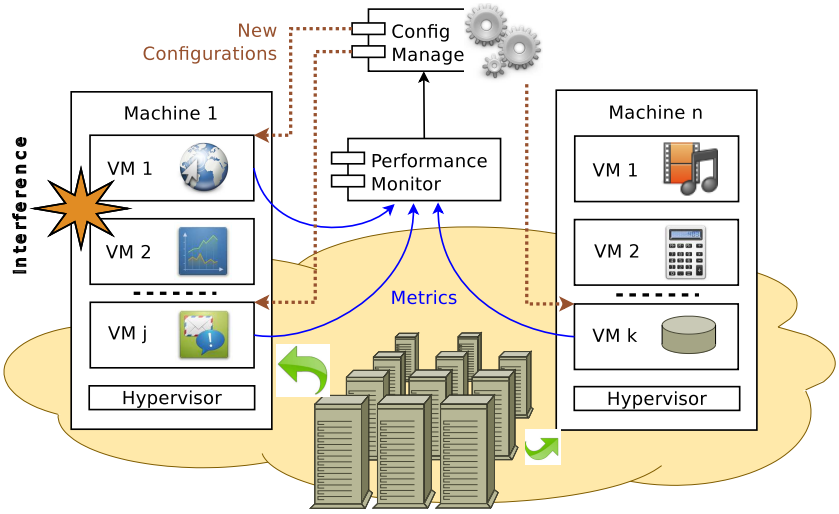


Fig. 6.5.: System architecture of IC^2

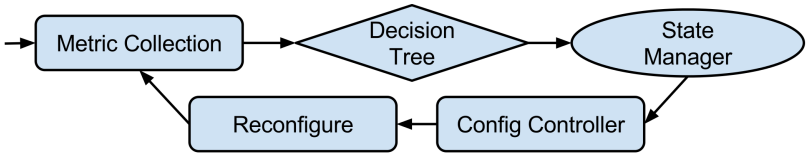


Fig. 6.6.: High level functioning of IC^2

6.4.1 Interference Detection: Metrics Used

Any interfering VM that is accessing large amounts of memory, such as our two experimental interference VMs running DCopy or LLCProbe, will ultimately cause a pressure on the shared cache on the physical machine. We find empirically that a sharp increase in CMR and CPI is a *leading* indicator of interference. Since in this dissertation, we focus on cache intensive interference, CMR was chosen as a representative trigger in the local testbed. We determined two thresholds for CMR as $CMR_{low}^{thres} = max(\text{CMR of Cloudsuite no-interference})$ and $CMR_{high}^{thres} = min(\text{CMR of Cloudsuite with interference})$. CMR_{high}^{thres} detects when interference is in effect and CMR_{low}^{thres} detects when interference has gone away. This approach, however, cannot be used in public clouds due to the policy of disallowing access to hardware counters.

For our experiments on EC2, we used a sharp rise in CPU, reduction in throughput (THPT) , and increase in response time (RT) of the application VM as secondary evidence of interference.

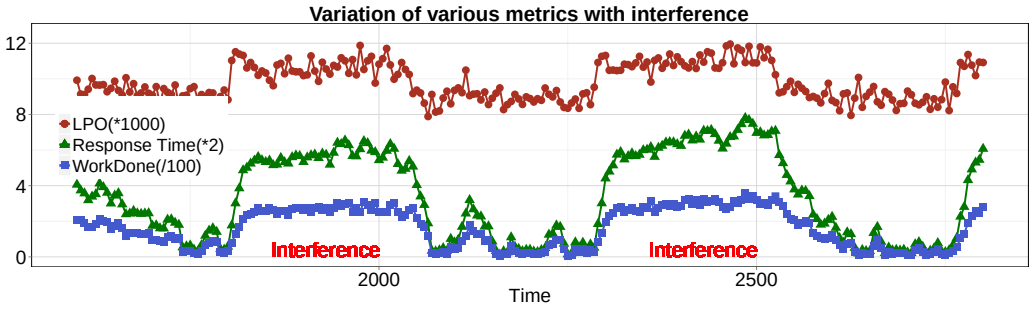


Fig. 6.7.: Interference impacts load per operation (LPO) and work done (WorkDone) by a web server. These, together with response time, can be used as metrics for detecting interference. The values are normalized by the factors shown in figure for better visualization.

Instead of using raw CPU utilization which may show sharp fluctuations due to stochastic nature of request arrivals, we use a normalized metric Load Per Operation (LPO). LPO is defined as $LPO = \frac{CPU_{util}}{Throughput}$. We also define another derived metric $WorkDone = RT * THPT * CPU_{util}$. Intuitively, Workdone approximates the number of CPU cycles spent to serve all the requests during current measurement interval. Without interference, assuming the server is not saturated, Workdone is small since $RT < 1s$ even though throughput is high. With interference, however, Workdone is large as response time increases significantly even though throughput reduces. To determine applicability of LPO and Workdone for interference detection, we ran CloudSuite in EC2 for multiple 1-hour runs. During these runs we periodically start Dcopy on a co-located VM at fixed intervals of 8 minutes (4 minutes of interference followed by 4 minutes of no-interference). The collected metrics for one 1-hour run is shown in Fig. 6.7. It can be seen from the figure that both LPO and Workdone form distinct clusters with and without interference. It may be argued that interference detection based on CPU utilization may fail to detect small interferences that does not increase utilization above threshold. In our experiments, we found such

interferences have minimal impact on response time. Here, our primary focus is to detect pathological cases that saturate server resources.

6.4.2 Decision Tree for Detecting Interference

To detect interference in EC2 we built a Decision Tree classifier using the attributes LPO, Workdone, and Response time. A decision tree generates a finite set of "tests" on attribute values to determine the class of a given sample. Our choice of decision tree is due to its two key advantages: i) simplicity—it is easy to visualize the rules in a decision tree ii) customizability—an administrator can manually change the thresholds of various attribute values based on expert knowledge or QoS requirements. Our classifier consists of 3 classes: Interference, No-interference, and Transient. The Transient class is introduced to capture temporary fluctuations in performance (e.g. immediately after starting or stopping of emulated interference). Based on current observation values the classifier tries to predict if the web server is suffering from interference or interference has gone away.

A key challenge in building the decision tree is to deal with changes in parameter values. When IC^2 reconfigures a web server during interference, its performance metrics also change. A decision tree trained with baseline runs (with and without interference, but default parameters only), doesn't work well to capture parameter changes. For example, during interference, when IC^2 reduces ResponseTime and LPO by modifying parameters, the decision tree incorrectly classifies this as no-interference. A possible solution for this is to collect metrics with various combination of parameter values, with and without interference. However, collecting training data for all combinations of parameter values is time consuming and may even be impractical. We therefore select an alternate bootstrapping approach where the classifier is trained in 3 phases. Each training phase consists of 10-hour run of Cloudsuite and is done offline. In phase I, we run Cloudsuite with baseline optimal setting (IC^2 disabled),

periodically generating interference. The collected data is used for training the phase I (base) classifier. In Phase II, we repeat the experiment with IC^2 enabled and use the base classifier for interference detection. The metrics collected approximate measurements with random parameter combinations. We use 50% data from Phase I (thereby, still biasing it towards default parameter values) and 50% data from phase II measurements to train the phase II classifier. Finally, in phase III we use phase II classifier and collect more data with IC^2 enabled. The data collected in phase III is used for training the final classifier. Note that, during training we use only Dcopy with varying array sizes and intensity (`#dcopy threads`) as our interference benchmark, while in evaluations we use both Dcopy and LLCProbe to test our detection module. We used the Weka [64] toolkit to create the decision tree.

6.4.3 Configuration Controller

IC^2 internally uses a simple state machine to keep track of current operating context of the web server and generate reconfiguration triggers (Fig. 6.8). In local testbed, the state machine consists of only two states and interference detection is merged with the state machine. We use response time in the trigger to ignore cases where response time was within QoS values, this prevents the server from incurring reconfiguration overheads during less-intense interferences. Self-loops in the state diagram are the negations of the trigger conditions on outgoing edge. In EC2, however, we use a 5-state machine, two representing interference and two representing normal (no-interference) runs, and one for the transient phase described in the previous paragraph. The transition labels are classifier outputs based on recent observations. Our choice of 5-states instead of two serves two purposes: i) Due to ambient interference in EC2, state changes may be short lived. Reconfiguring frequently in such cases may impact throughput. Our design forces IC^2 to reconfigure only after it has seen two successive periods under interference or no-interference (assuming the current phase

will last a while). ii) This hides classifier false positives. For example, if the server is under no-interference but the classifier predicts interference, it would take at least three successive misclassifications for a reconfiguration (No-interference → Transient → Interference → Interference), the probability of which is much smaller than the classifier error rate. IC^2 performs reconfiguration actions when the server enters the states `I2` or `NI2` as shown in Fig. 6.8.

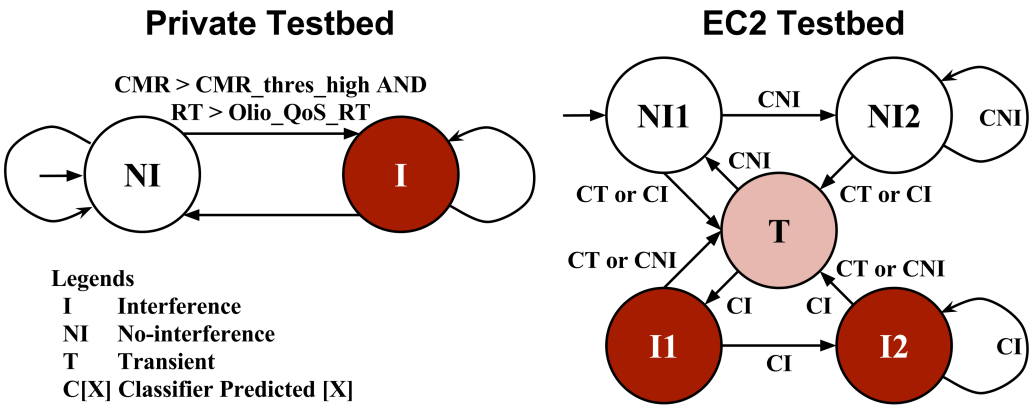


Fig. 6.8.: State transitions of IC^2. In EC2, reconfiguration is done when the server enters I2 or NI2.

6.4.4 Reconfiguration Actions

Our reconfiguration actions in IC^2 are currently implemented as a heuristic backed by a knowledge base (refer Table 6.3). This knowledge base directs IC^2 *which* parameters to reconfigure when a trigger is detected. It does not include precise values of the parameters but instead specifies a set of rules. The knowledge base can be created in two ways: i) with the help of a domain expert, ii) analyzing performance logs from training runs. Note that most commercial web applications go through load testing phase before going to production. A systematic variation of critical middleware parameters (as in Section 6.3.2) during these tests can generate insights about application performance. Our current implementation deals with row 3 of Table 6.3,

i.e. increased CMR. Our earlier experiments suggest that the actions MXC↓, KAT↑, and PHP↑ can improve application performance during phases of cache interference. We reconfigure all three parameters simultaneously.

Table 6.3.: Knowledge base for web server reconfiguration

Context Change	**Config. Heuristic**
Increased Workload (High Idle Memory)	MXC↑ and PHP↑
Increased Virtual/Physical CPU ratio	MXC↓ and KAT↑
Increased LLC Miss Rate	MXC↓, KAT↑, PHP↑
Increased Host Memory Contention	MXC↓ and PHP↓
Increased Page Faults (Active Memory Low)	PHP↓

Algorithm 1 Parameter update functions for IC^2

1: **procedure** RECONFIGURE_FOR_INTERFERENCE()
2: $\delta_{MXC} \leftarrow ((MXC * \frac{LPO - LPO_{nointf\_median}}{LPO}))$
3: $\delta_{MXC} \leftarrow checkBounds(\delta_{MXC})$
4: $\delta_{KAT} \leftarrow (\delta_{response} * C_{KAT})$
5: $\delta_{KAT} \leftarrow checkBounds(\delta_{KAT})$
6: **update_params**$(\delta_{MXC}, \delta_{KAT}, (400 - PHP))$
7: **end procedure**

6.4.5 Update Functions

The quantitative update functions for the three parameters are shown in Algorithm 1. The update objective for MXC is to reduce CPU demand of the web server. We therefore decrease it proportional to the increase in CPU utilization (approximated by δ_{LPO}). We restrict the new value to be within a min-max bound so that throughput does not degrade alarmingly. A similar objective function can be realized for a memory constrained web server by considering memory utilization. An underlying assumption here is that the server's CPU utilization is dominated by the Apache Httpd server. In our setup, though Php-fpm was used for dynamic content generation, we found the impact of `PhpMaxChildren` on response time/throughput was much

smaller than `MaxClients`. This likely indicates that the effect of `PhpMaxChildren` on CPU utilization of the VM was marginal.

On the other hand, increase in response time implies the server's average request cycle time (response time + wait time) is increased. We increase KAT proportional to the $\delta_{response\_time}$ to offset increased cycle time, i.e., to keep a connection alive for longer since the server is taking longer time to respond to client requests. During experiments in Fig. 6.2(b), it was found that the increase in optimal KAT value ($KAT_{opt}^{intf} - KAT_{opt}^{nointf}$) during interference is several times larger than $\delta_{response\_time}$. Therefore, a constant multiplicative factor (C_{KAT}) is used with $\delta_{response\_time}$ to come up with the change in the KAT value. We empirically determined the value of C_{KAT} to be 3. For `PhpMaxChildren`, we found performance improvement beyond a certain value (400, for a VM with 2vcpus) is negligible. We therefore select two constant values of PHP for interference and no-interference scenarios.

6.4.6 Implementation

IC^2 currently has been implemented as a Java application which combines the functionalities of Performance Monitor and Config Manager described in Fig. 6.5. One instance of IC^2 is designed to handle an application group as shown in Fig. 6.5, e.g., an application group may consist of web server, database server, and e-mail server. In current implementation, we focus on managing the web server. IC^2 uses remote scripts to fetch performance metrics from various levels of the monitored systems. It uses Faban logs to collect application level metrics (in periodic intervals of 5s), and uses `sysstat` utilities (inside WS VM) for cpu and memory utilization. In the local testbed, we also collect hardware counters from the hypervisor. Based on the collected data and configured threshold values, it detects if an interference has started or stopped. It then sends reconfiguration commands to the Apache and Php servers. A separate program was implemented to start and stop the interference

benchmarks in periodic intervals.

Redesigning Httpd. During initial testing with IC^2, we found that CloudSuite had significant increase in response time and decrease in throughput immediately after a reconfiguration. This transient phase lasted between 30-60s and was determined to be a limitation of Apache Httpd server. In order to update configuration values, the server has to restart all child processes. This is essential because Httpd internally assumes that configuration values are never changed (read-only)—doing so allows it to avoid synchronization overhead during request processing.

To avoid the penalty of restarting Httpd, we implemented an online reconfiguration option for Httpd. The online reconfiguration option enables Httpd to gracefully change over from old parameter values to new parameter values without needing to shut down and restart all worker processes. We noted that `MaxClients` is used only in Httpd master process to control the number of worker threads. The children processes (workers) are oblivious of MXC. Therefore MXC can be updated in master (and subsequently propagated to children) without requiring restart. KAT value is read at the end of every request processing, therefore any change to it is reflected in the next request. Assuming a relaxed consistency model, we can modify KAT in master and propagate the changes to children later.

We implemented a custom signal handler (`SIGUSR2`) and an online reconfiguration command (`apachectl reconfigure`) in the Httpd server (`worker mpm`) to initiate online reconfiguration of these two parameters. The signal is delivered to the master process by `apachectl` and later propagated to children via Httpd's Pipe of Death (POD) implementation. We also updated the Scoreboard structure to store runtime values of MXC and KAT. The reconfiguration decisions are implemented in `server_main_loop()` in master and `child_main()` in children. Our implementation involved adding/modifying 500 lines of code in current Apache codebase (v2.4.3). With our implementation of online Httpd, the server showed significantly less over-

head of reconfiguration as explained in the next section. The modified version of Apache can be downloaded from [65].

6.5 Evaluation

In this section, we evaluate the effectiveness of IC^2 in detecting and remediating interference. The high level objective here is to reduce the response time for the web server during periods of interference. Therefore, if the average response time after reconfiguration is lower than that before reconfiguration, we consider IC^2 to have achieved its objective. More specifically, we ask ourselves the following questions, individually for the local testbed and Amazon EC2:

i) Can IC2 successfully detect interference?

ii) How much improvement in response time can be obtained by running IC^2?

iii) What is the overhead of reconfiguration in IC^2?

6.5.1 Setup

To quantify the performance change due to IC^2, we compare IC^2 with the performance of an interference-agnostic controller. We assume that an interference-agnostic controller is able to achieve optimal parameter setting under normal runs and does not react to interferences. We found that for resource configurations equivalent to EC2 `m1.large` instances, the optimal parameter values for no-interference runs were $< MXC = 650, KAT = 5, PMC = 50 >$. For all the experiments, we consider a CloudSuite workload with 550 concurrent clients which is below the saturation point of the web server.

Interference Emulation. To emulate interference we star-ted the interference benchmarks with varying array sizes at different instants in time. To simplify implementation, we consider a periodic interference behavior as opposed to a stochastic

behavior. Due to transient behavior of httpd-basic immediately after reconfiguration, we found it difficult to precisely evaluate benefits of IC^2 with a bursty interference. For our evaluations the interference benchmarks are on for 240s followed by an off period of 240s. We selected emulated interference to evaluate IC^2 instead of natural interference in EC2 primarily because of two reasons: 1) Interferences occur infrequently enough to make statistically significant results difficult within a reasonable experimental time. 2) The nature (intensity and duration) of interference may change every time making it hard to draw comparable results. We run LLCProbe with an array size of 20MB, Dcopy with 20MB (also referred to as Dcopy-low) and Dcopy with 1.5GB (Dcopy-high). On EC2, this synthetic interference happened in addition to ambient interference in the environment. On the local testbed, we ensured that no other VM, extraneous to our experiment, was running.

Co-location in EC2. In order to evaluate IC^2 in EC2 we needed to co-locate some of our VMs on the same machine as the WS VM. This is necessary to emulate interference on the web server. We iteratively started 10 EC2 instances in batches (as described in [52]) and were able to successfully co-locate 2VMs after some trial and error. We found that the co-located instances had sequential `domid`s and were able to pass messages among themselves using `xenstore` (write in one VM and read from another). We used this as verifying evidence that co-location was achieved. Our results, in themselves, are also secondary validation of co-location since we found noticeable impact of interference on WS performance. The co-located VMs on EC2 were hosted on a Xeon-2650 machine having 8(16) physical(logical) cores and 20MB L3.

Baseline Formation. To form baseline observations for both private testbed and EC2 we first configured their corresponding web servers to the no-interference optimal settings. These settings, with IC^2 disabled, emulate an interference-agnostic controller. We then used Faban to generate client requests for a 1-hour run. During the run we started our interference controller described above to generate periodic

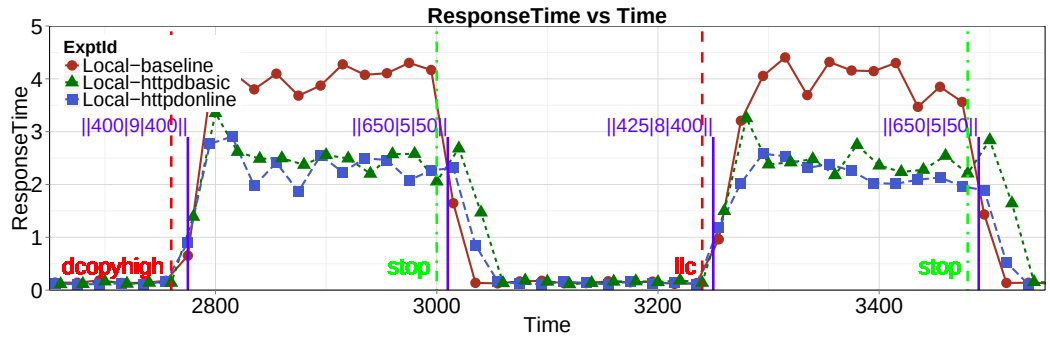
interferences. The application metrics for CloudSuite (response time and throughput) were collected at intervals of 5s. These metrics when plotted against time axis represent performance of one baseline run. In general, we found that interference had more performance impact in EC2 than in local testbed. For this experiment, we reconfigured the WS VM on the local testbed to match Amazon EC2's m1.large instances. To achieve noticeable impact, we had to use 4 threads of the interference benchmark on the local testbed compared to 2 in EC2. We found that with this utilization of the local server (6 of 12 hardware threads) the effects of interference in local and EC2 were of comparable magnitude.

Similar to baseline measurements, we also ran CloudSuite with IC^2 enabled. In both testbeds, we evaluate IC^2 under two scenarios: one where Apache is reconfigured with traditional `apachectl -k graceful` command (httpd-basic) and the other where our instrumented version of Apache is reconfigured online (httpd-online). We iteratively start 1-hour of baseline run followed by 1-hour of IC^2 with httpd-online, and 1-hour of httpd-basic. This was repeated 16 times for a total runtime of 48 hours (3×16). We restart the web server between each 1-hour run.

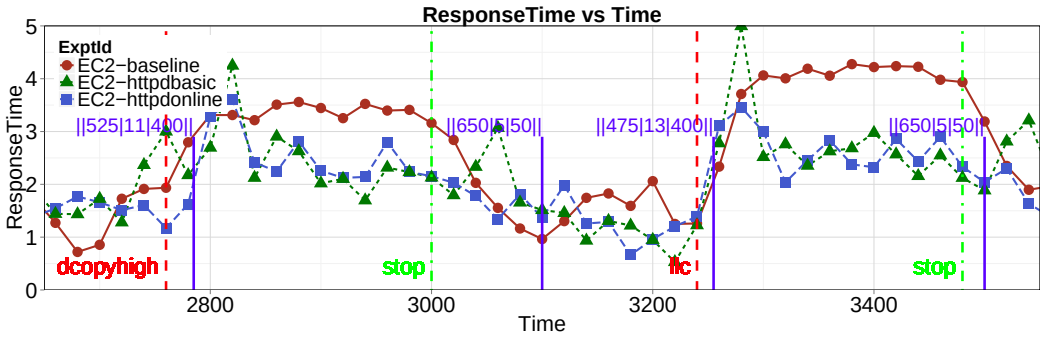
6.5.2 Results

Improvement in Response Time

Fig. 6.9 shows the variation in Response time with Time in local testbed and in EC2 for a set of representative runs. In each plot, red vertical lines show the point on time axis when an emulated interference is started and green vertical lines show when interference is stopped. The blue vertical lines show the point when IC^2 reconfigured with httpd-online. New parameter settings at each reconfiguration point is annotated as the three tuple $|MXC|KAT|PHP|$. It can be seen that in general both httpd-online and httpd-basic are able to reduce response time during interference. In case of httpd-basic, there is a spike in response time following a reconfiguration, an indication that Apache is restarting all of its child processes. With



(a) On Private Testbed.



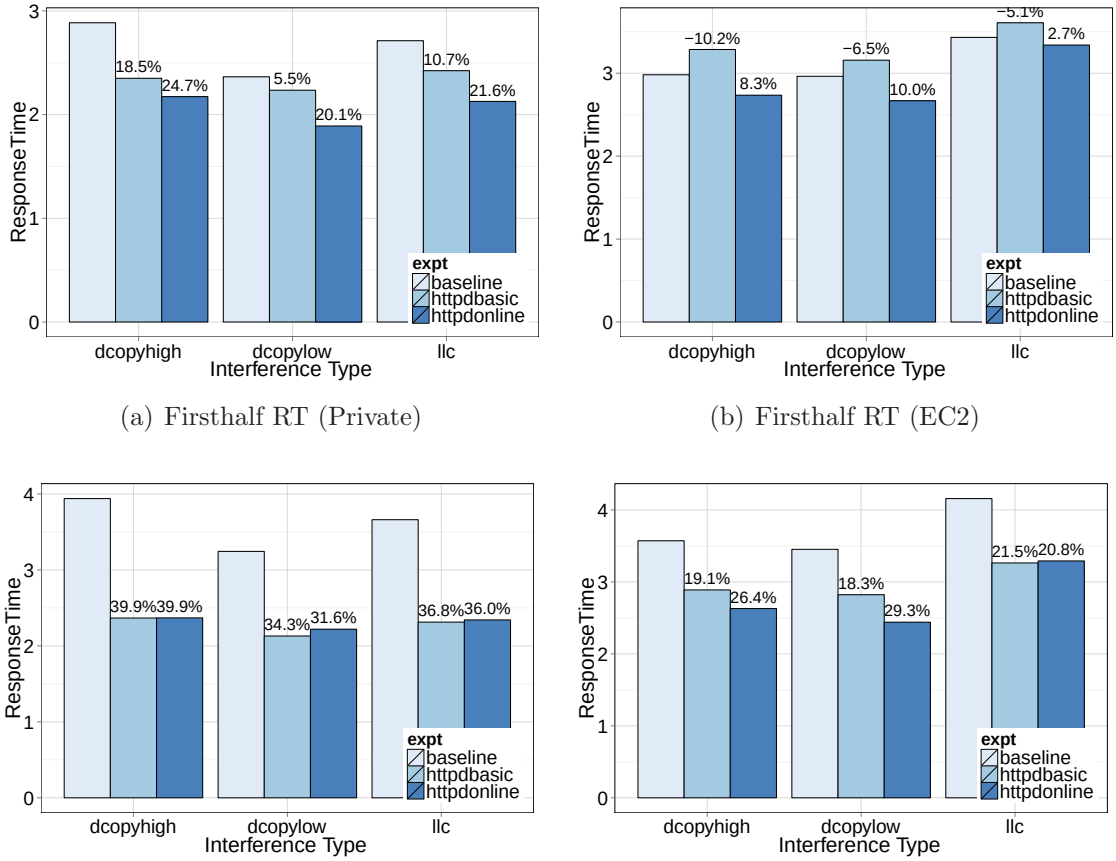
(b) On EC2 Testbed.

Fig. 6.9.: IC^2 improves response time of a web server during phases of interference. Red vertical bars show when an emulated interference is started and green vertical bars show when interference is stopped. The blue vertical bars show the point when IC2 reconfigured with httpd-online. New parameter setting at each reconfiguration point is annotated as the three tuple $|MXC|KAT|PHP|$. Baseline run implies IC^2 is disabled.

httpd-online this spike is nearly eliminated, although, some overhead remains due to updating of `PhpMaxChildren`. Interference detection is faster in Local than EC2 since we use cache miss rate. Another interesting fact is that the effect of interference persists longer in EC2 even after emulated interference is stopped. This happens for two reasons, i) ambient interference in EC2, ii) max throughput in EC2 is lower than in the local testbed, hence queued requests persist for longer in EC2.

To quantify improvement in response time, we analyze response time during interference in two halves—a) From onset of interference (red line in Fig. 6.9) up to 60 seconds is considered first half. This is the period when interference detection and reconfiguration take place effectively showing overhead of IC^2, specially in case of

httpd-basic. b) From 60s after interference to stopping of interference (green line in Fig. 6.9) is considered the second half. This is the steady state performance of IC^2 during interference.



(a) Firsthalf RT (Private)

(b) Firsthalf RT (EC2)

(c) Secondhalf RT (Private)

(d) Secondhalf RT (EC2)

Fig. 6.10.: Response time with IC^2 for various interferences. Numbers represent percentage improvement from baseline RT.

We found that, across different interference types in EC2, httpd-basic degraded response time by 5-10% in the first half, but httpd-online improved response time by 3-10%. This proves that the online version of Apache is able to reconfigure faster. In local testbed, during the first half, httpd-basic showed improvement between 5 and 19% while httpd-online showed improvement between 20 and 25%. The measurements are better in the local testbed compared to EC2 since interference detection happens

faster. In steady state or second half (60-240s from onset of interference), httpd-online showed improvements of 21-29% in EC2 and 32-40% in the local testbed (refer Fig. 6.10). The numbers for httpd-basic are 18-22% in EC2 and 34-40% in Local. The steady state performance of httpd-basic and httpd-online are comparable in the local testbed, although httpd-online outperforms httpd-basic in EC2. Overall IC^2 showed higher improvement in response time in the local testbed since it was able to compute δ_{MXC} and δ_{KAT} more precisely (no ambient interference as in EC2). We find that the response time improvements are significant considering the simplicity of our controller. It further establishes our point that, in a cloud deployment, an application configuration manager must be interference-aware. A summary of our results can be found in Table 6.4.

Detection Latency

From the collected metrics we also measured how long it takes for IC^2 to detect interference in EC2 and in the local testbed. We define detection latency to be time from the starting or stopping of an emulated interference to the time when IC^2 reconfigures Apache server. In Fig. 6.9 these are the times between a red line and the next blue line (we call this interference detection latency) or time between a green line and the next blue line (no-interference detection latency). We found that in local testbed, median values for interference and no-interference detection latencies are 15s and 10s respectively. In comparison, IC^2 detects interference in EC2 with a median latency of 20s. Detection of no-interference in EC2 takes much longer—a median value of 65s. This happens since effect of interference persists much longer in EC2 as described in the previous section (Fig. 6.9(b)). Our future work includes finding ways to reduce detection latency even further in both testbeds.

Table 6.4.: Summary of IC^2 Results. Response time numbers are %change from baseline runs across interference benchmarks. FH:=first half, SH:=second half, INTF:=interference, NI:=no-interference

	Response Time (%change)				Detection Latency	
	httpd-online		httpd-basic			
	FH	SH	FH	SH	INTF	NI
Local	20-25↓	32-40↓	6-19↓	34-40↓	15s	10s
EC2	3-10↓	21-29↓	5-10↑	18-22↓	20s	65s

Classifier Accuracy

To measure the accuracy of our classifier we apply it on the data collected from our experiments in Section 6.5. For each experiment type (httpd-online and httpd-basic), we create a test set comprising measurements collected by IC^2 in that experiment. Due to space constraints, we present only the first type here. We label the test data based on its timestamps and our knowledge of when an emulated interference is started and stopped. Data from the start(stop) of an interference up to 30s is labeled Transient, rest are labelled according to which interval it is (Interference or No-interference). Note that this labeling does not take into account ambient interference in EC2 and therefore may manifest as poorer precision, although the classifier works well in practice as seen in results from Section 6.5.2. We found the Transient class had significant overlap with both Interference and No-interference in the training data, as a result it had very low precision. But since IC^2 does not perform any reconfiguration in this state, the cost of misclassification is zero. We therefore ignore the results for Transient and focus primarily on interference detection.

During the training phase, it was found that with default cost for misclassification,

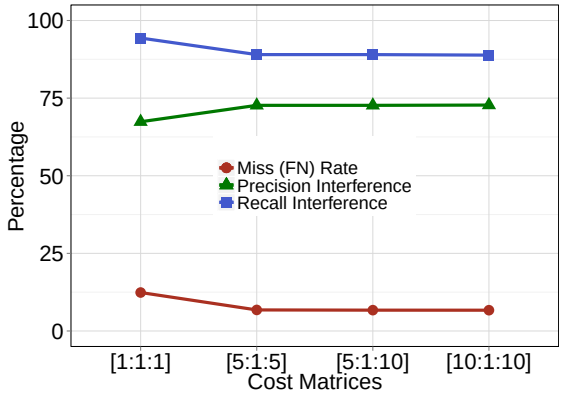


Fig. 6.11.: Accuracy of Interference Detection with varying cost matrices. The cost values $5:1:10$ are used in production.

the decision tree had significant number of False Negatives (FN). This causes IC^2 to perform badly, e.g. using baseline parameter setting while in interference may have

significant performance degradation and vice versa. We tried several combinations of cost matrices and selected the one with lowest miss rate. Fig. 6.11 shows the interference detection rate of our classifier with different cost matrices for httpd-online test data. Here a cost value of $\{a : b : c\}$ represent a 3×3 cost matrix, a is the cost of No-interference (NI) detected as Interference (I), b is the cost of Transient misclassified as any other class, c is the cost of (I) classified as (NI), and cost of correct prediction is 0. We define Miss Rate as (I classified as NI + NI classified as I)/Total Samples. It can be seen that, initially with default cost of $1 : 1 : 1$, miss rate was 12%, but with higher cost values miss rate reduced to 6%. We used the cost values of $5 : 1 : 10$ in our production runs based on the fact that response time penalty for misconfiguration in (I) is much higher than cost of misconfiguration in (NI). The largest percentage of FNs (98.7%) arise from NI being detected as I (with cost $10 : 1 : 10$). This happens since impact of interference persists longer in EC2 as seen in Fig. 6.9(b) (but our labeling does not account for this). This also manifested as lower precision of (I) and lower recall of (NI). In general, our interference detection achieved 89% recall and 73% precision.

Cost of IC^2

The cost of IC^2 can be defined in terms of two metrics—(a) Apache performance immediately after a reconfiguration, and (b) execution cost of IC^2. We already found in Section 6.5.2 that IC^2 with httpd-online improves response time during first half, both in EC2 and in private testbed. This indicates httpd-online is able to reduce cost of reconfiguration significantly (compared to httpd-basic, response time improved by up to 17%). Note that IC^2 is trained offline, therefore it does not have any runtime cost for building the classifier. Since the classifier has only 3 attributes, the tree has a simple structure and classification decision is made in the order of 10 comparisons. This is insignificant compared to our measurement period of 5s, which is also the

frequency at which the classifier is invoked. Therefore, the execution cost of IC^2 is negligible.

6.6 Discussions

How generic is the knowledge base in IC^2?

Even though we work with Apache, most parameters we work with are common to thread-pool based middleware. Parameters like `MXC`, `PHP` capture concurrency, whereas timeout parameters capture state preservation. Hence, we believe the knowledge base (KB) in Table 6.3 to be applicable to other thread-pool based servers (e.g. Glassfish, WebSphere, DB2, MySql etc.). In a small scale experiment with Glassfish, we found that it has sensitivity to thread-pool size (similar to MXC). Our KB, however, is not applicable to event-driven architectures (e.g. Nginx).

How expensive is it to generate the knowledge base?

The knowledge base in IC^2 can be created empirically by systematically varying important parameters as described in Sec. 6.3.2. It can be done in parallel with the load testing phase of web applications. Note that the KB does not include precise values of parameters, rather IC2 can figure out the parameter values depending on runtime conditions, including interference. Once created, it can be used for a given application and middleware distribution irrespective of deployment (assuming similar architectures, e.g. x64 or x86).

Can IC^2 handle other types of interference?

Network interference is another major problem that seriously affects performance of cloud applications. IC2 can also be useful in mitigating some of the effects of network interference through application reconfiguration. In our preliminary experiments, we simulated an environment where bandwidth available to the WS-VM became constrained (by up to 20%) due to a co-located VM using up a major share of the network. We observed, as the level of network interference increases (i.e. the available bandwidth to WS-VM reduces) the response time of the webserver sharply degrades.

IC2 can improve response time by employing an *admission control* mechanism, which is equivalent to reconfiguring the `MaxClients` parameter in Apache to a lower value. We empirically verified that optimal MXC setting with network interference is lower than no-interference optima [62]. IC2 can be trained to use response time along with packets pruned from send-buffer as a trigger to detect such network interference.

6.7 Directions for Future Research

In this chapter, we presented the design and implementation of IC^2, an autonomous middleware configuration engine for mitigating performance interference in clouds. IC^2 has the distinct advantage that it can be used by website administrators who usually do not have access to the hypervisor in public clouds. In such cases, the simple heuristic-driven controller in IC^2 is able to reduce the response times of web server VM(s). However, the design of IC^2 requires substantial expert knowledge in determining which parameters should be reconfigured during interference and how their values should be changed (increased or decreased). We generated this knowledge base (KB) for popular web services middleware such as Apache Httpd and Php-fpm engine by running an extensive set of experiments. The KB tells us how the optimal values of various parameters change with and without interference. In further experiments, we found that the knowledge base is applicable to a specific class of middleware: e.g., in our setup the IC^2 knowledge base was applicable to thread-pool based middleware (such as Apache web server and Darwin streaming server). However, for event-based middleware (such as Nginx) we didn't find any change in optimal parameter values with interference. Further research is required to automatically find application parameters that are performance-critical.

Below we present some future research directions for enhancing IC^2.

1. *Automatically Finding Performance-critical Parameters in IC^2:* Finding performance critical parameters for applications and especially finding optimal parameter values has been a long-standing "open" problem in the systems com-

munity. The difficulty of the problem arises from three facts: i) the exploration space for configuration values is huge, ii) there are dependency among parameters and often this dependency spans across machines in the server cluster, iii) it is very difficult and time-consuming to gather sufficient training data for all possible combinations of parameter values. Significant research is required to address these challenges and find optimal values for performance-critical parameters in a completely autonomous and efficient manner.

2. *Handling Other Types of Interference:* In IC^2, we detect and mitigate interferences due to cache contention. A possible direction for future research is to extend IC^2 to handle other types of interferences such as network or IO bandwidth. Such a solution would require designing new modules for detecting these interferences and finding which parameters can mitigate their ill effects. We highlight a possible solution in Section 6.6.

7. ICE: AN INTEGRATED CONFIGURATION ENGINE FOR CLOUD SERVICES

Performance degradation due to imperfect isolation of hardware resources such as cache, network, and I/O has been a frequent occurrence in public cloud platforms. A web server that is suffering from performance interference degrades interactive user experience and results in lost revenues. Existing work on interference mitigation tries to address this problem by intrusive changes to the hypervisor, e.g., using intelligent schedulers or live migration, many of which are available only to infrastructure providers and not end consumers. In this chapter, we present a framework for administering web server clusters where effects of interference can be reduced by intelligent reconfiguration. Our controller, ICE, improves web server performance during interference by performing two-fold autonomous reconfigurations. First, it reconfigures the load balancer at the ingress point of the server cluster and thus reduces load on the impacted server. ICE then reconfigures the middleware at the impacted server to reduce its load even further. We implement and evaluate ICE on Cloudsuite, a popular web application benchmark, and with two popular load balancers - HAProxy and LVS. Our experiments in a private cloud testbed show that ICE can improve median response time of web servers by upto 94% compared to a statically configured server cluster. ICE also outperforms an adaptive load balancer (using least connection scheduling) by upto 39%.

7.1 Motivation

Performance issues in web service applications are notoriously hard to detect and debug. In many cases, these performance issues arise due to incorrect configurations or incorrect programs [66]. Web servers running in virtualized environments also

suffer from issues that are specific to cloud, such as, interference [50, 67] or incorrect resource provisioning [47]. Among these, performance interference and its more visible counterpart performance variability cause significant concerns among IT administrators [55]. These concerns are justified since a slow website almost always implies customer dissatisfaction and missed revenues. Performance interference also poses a significant threat to the usability of Internet-enabled devices that rely on hard latency bounds on server response (imagine the suspense if Siri took minutes to answer your questions!). For other latency sensitive tasks such as web search, delayed tasks are simply discarded resulting in wasted computation. Existing research shows that interference is a frequent occurrence in large scale data centers [50, 68]. Therefore, web services hosted in the cloud must be aware of such issues and adapt when needed.

7.1.1 The Problem

Interference happens because of sharing of low level hardware resources such as cache, memory bandwidth, network etc. Partitioning these resources is practically infeasible without incurring high degrees of overhead (in terms of compute, memory, or even reduced utilization). Existing solutions primarily try to solve the problem from the point of view of a cloud operator. The core techniques used by these solutions include a combination of one or more of the following: a) Scheduling, b) Live migration, c) Resource containment. We presented a high level overview of these techniques and their shortcomings in Chapter 5. Our principal objective here is to mitigate effects of interference *without* requiring modification of the hypervisor.

7.1.2 Improving IC^2

In the previous chapter, we saw how IC^2 mitigates interference by reconfiguring web server parameters in the presence of interference [67]. The parameters considered are `MaxClients` (MXC) and `KeepaliveTimeout` (KAT) in Apache and

`pm.max_children` (PHP) in Php-fpm engine. We found that IC^2 could recapture lost response time by upto 29% in Amazon's EC2 and 40% in private cloud testbed. However, the drawback of this approach is that web server reconfiguration usually has high overhead (the web server need to spawn or kill some of the worker threads). We found that with a standard Apache distribution, the response time immediately after a reconfiguration shot up by 1s and the negative effects lasted for about 30s. This limits how frequently the web server can be reconfigured. Moreover, IC^2 alone cannot improve response time much lower without drastically degrading throughput. We observe that the key goal in IC^2 is to reduce the load on the web server (WS) during periods of interference. The reconfiguration actions to reduce MXC and to increase KAT help in achieving this objective. However, this is an indirect way of implementing load reduction. We also observe that most deployments have multiple instances of web servers placed behind a load balancer. Therefore, implementing admission control at the gateway (load balancer) is a more direct way of reducing load to the affected web server. An out-of-box load balancer is agnostic of interference and therefore treat the WS equallly irrespective of whether it is suffering from interference or not. We aspire to make this context aware, and evaluate how much performance gain can be achieved.

7.1.3 Solution Approach.

Our solution approach relies on leveraging the admission control functions of load balancer to make the system more agile. We observe that existing load balancers (e.g. HAProxy with least-connection scheduling) can mitigate the effects of interference to some extent. However, if we can detect and mitigate interference sooner, it can improve system stability and reduce response time even further. The proposed solution, called ICE, uses hardware counter values to detect the presence of interference and therefore provides fast detection. Access to hardware counters does not always require hypervisor (root) access as these may be virtualized [69]. Our evalu-

ations showed that ICE could detect and reconfigure the WS cluster with a median latency of 3s. This is better than existing techniques which would incur a detection latency of 15-20s [67].

When interference is detected, ICE performs two-level reconfigurations of the web-server cluster. The first level, geared towards agility, is to reconfigure the load balancer in such a manner so that fewer requests are forwarded to the affected WS VM. We found that this provides the maximum benefit in terms of reduced response time. The second level of reconfiguration, which configures Apache and Php-fpm as described in IC^2, is activated only if the interference lasts for a long time. This prevents the server from incurring unnecessary reconfiguration overheads if the interference is short. The second level reconfiguration ensures that the WS does not incur overhead of idle threads (note that since the LB has been reconfigured this WS VM will receive fewer client requests). Moreover, it also works as a fail-safe for situations where the WS VM still receives lot of requests because of a sudden spike in client load at the LB.

Our key contributions in this chapter are as follows:

1. We present the design and implementation of a two-level reconfiguration engine for web server clusters to deal with interference in cloud. Our solution, called ICE, includes algorithms for: a) detecting interference quickly (primarily cache and memory bandwidth contention), and b) predicting new weights for impacted server(s) to reduce load on them.

2. We deploy our solution on a popular web application benchmark called CloudSuite. Our evaluation experiments show that on a combination of Apache+Php+HAProxy middleware, ICE can reduce median response time of web servers by upto 94%. We evaluate ICE for two different scheduling policies (weighted round-robin, and weighted least-connection) in HAProxy and find that it improves response time across both scheduling policies (upto 94% and 39% respectively). Median interference detection latency was 3s.

3. To evaluate the generalizability of our framework, we also ran some experiments with the Darwin media streaming server running with LVS load balancer. Our results

show that reconfiguring server weight in LVS can be used to reduce the inter-frame delay for an affected Darwin VM. We also found that the optimal `num_threads` parameter in Darwin is vastly different with and without interference indicating our WS reconfiguration is also applicable to Darwin.

The rest of the chapter is organized as follows. In Section 7.2, we show that interference can degrade response time of a WS VM even in a load-balanced setup. It is followed by a detailed description of the design of ICE. Section 7.4 shows the performance improvement of ICE over a simple load-balanced WS cluster and IC^2. Next, we show the generalizibilty of our solution approach by illustrating how ICE can be applied in a media streaming server cluster. We use the Apple Darwin media streaming server and LVS load-balancer for this work. We finally discuss the shortcomings of ICE and conclude the chapter.

7.2 Interference Degrades Performance of Web Servers

In this section, we present an empirical evaluation of the impact of interference on web server (WS) performance. The metric of interest in this dissertation is the average response time of the WS, which is a dominant factor for customer satisfaction.

7.2.1 Experimental Setup

Testbed. Our experiments were conducted in a private cloud testbed consisting of three poweredge 320 servers. Each server is equipped with an Intel Xeon E5-2440 processor consisting of 6 cores (or 12 threads with hyperthreading), 15MB L3 cache and 16GB main memory. The hypervisor used to manage this cloud testbed is the popular KVM hypervisor. Each of our experimental virtual machines (VM) were configured with 2 vcpus and 3GB RAM except the DB VM. The DB VM was provisioned with 4 vcpus and 4GB RAM to eliminate any DB bottleneck. All the machines were connected by a 1 Gbps switch.

Application Benchmark. For our experiments with web servers, we use the popular CloudSuite [19] web application benchmark. CloudSuite emulates a social event calendar application written in PHP. It is an example of a dynamic web application, where most of the responses are generated by executing PHP scripts and running database queries. The benchmark also includes a custom workload generator based on real-life distribution of user requests.

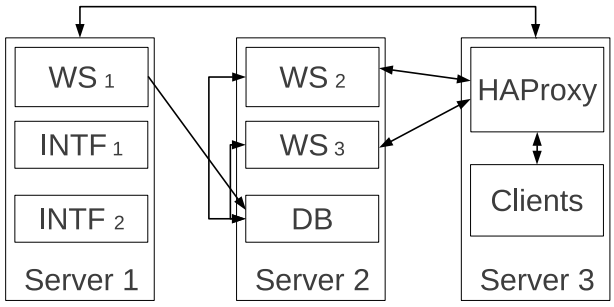


Fig. 7.1.: Layout of virtual machines in private cloud testbed.

Virtual Machines and Middleware. We used a multi-threaded Apache web server (*Apache-worker*) to host CloudSuite, while PHP scripts were interpreted using the PHP-Fastcgi Process Manager (*php-fpm*). Note that one instance of Apache and php-fpm runs in a single virtual machine (called WS VM). We replicated the WS VM three times to create a web server cluster with 3 virtual servers. These were distributed across 2 physical machines (refer Fig. 7.1). The third machine was used to emulate clients. Note that the physical machine with one WS VM was chosen to run interference to limit the impact of interference across the cluster. Unless otherwise specified, we refer to *this* WS VM as the monitored web server (or impacted web server, also denoted WS_X). All our measurements and analysis below pertain to this single WS VM instance (WS_X). This also makes sure that our measurements are free from WS-WS (intra-cluster) interference. The CloudSuite database was hosted in MySql and placed on the same physical machine as the other two WS VMs. Note that the combined utilization of these VMs (2WS VM + 1DB VM, for a total of 8 vcpus) was

below the total capacity of the phycial machine (12 vcpus) and the physical machine never contended for CPU or RAM. We used the popular HAProxy load balancer to distribute requests equally among the WS VMs. HAProxy was deployed in a VM on the same physical machine as the client emulator. This was done to reduce network overhead. We found that the CPU and Memory utilizations of HAProxy and Client VM were always well below their provisioned capacities.

Interference Benchmarks. We use the LLCProbe and Dcopy benchmarks described in [67] for generating interference on the monitored WS VM. LLCProbe creates an LLC Sized array in memory and then accesses each cache line very frequently. On the other hand, Dcopy from the linear algebra suite (BLAS) copies contents of one array to another. While LLCProbe is an example of a cache read benchmark, Dcopy is an example of cache read+write benchmark. We found that in general LLCProbe has greater cache access frequency than Dcopy and therefore emulates a stronger interference. Interferences of these types occur frequently in cloud platforms during computation of data mining tasks [50], or during periodic consolidations [47].

Configuration Parameters. We consider several parameters in Apache, PHP and HAProxy for reconfiguration tasks in *ICE*. The HAProxy parameter that *ICE* manages is the `server weight` parameter which determines what fraction of all client requests go to a particular WS VM. In Apache, we automatically reconfigure the `MaxClients` (MXC) and `KeepaliveTimeout` (KAT) parameters for improving response times. While MXC indicate the maximum number of worker threads that an Apache server can spawn, KAT determines how long a client connection is persisted in idle state before terminating it. Both these parameters have large impact on response time of an Apache server. In Php, we consider the `pm.max_children` (PHP) parameter for autonomous reconfiguration. Similar to MXC, PHP indicates the maximum number of worker threads in the Php-fpm engine. Note that although our current experiments are geared towards Apache+Php+HAProxy setup, similar

parameters exist for most threadpool-based web services middleware and load balancers. Therefore, our design principles can be easily adapted for other web servers. We show this with an example of the Darwin media streaming server with LVS load balancer in Section 7.5.

7.2.2 Interference Increases Response Time

In this experiment, we run CloudSuite with a given load size (`#concurrent_clien-ts`) for several 1-hour runs. Note that all the clients direct their requests to the HAProxy load balancer which distributes the requests equally among WS VMs. The scheduling policy selected by HAProxy for distributing client requests is round-robin in this experiment. During each run, LLCProbe benchmarks were started and stopped in the interference VMs periodically. We show the response times of each WS VM over time in Fig. 7.2. It can be seen that with interference the response time of the impacted WS VM (WS_X) increases sharply from 10x ms to 100x ms. This clearly shows that interference can degrade performance of few WS VMs, although, we found that rest of the cluster remained under-utilized. The reason behind this sharp rise in response time is that interference reduces capacity of the WS. Therefore, an impacted WS may saturate with fewer number of clients than in a no-interference scenario. For a detailed treatment on how interference impacts performance of WS, the readers are referred to [67].

This shows that a load-balancer which has no knowledge of which WS VM is suffering from interference, would continue to send the same number of requests to the impacted server(s), as to the other WSs (assuming same default weight). We therefore need to make the load-balancer aware of dynamic situations (interference) and make intelligent scheduling decision.

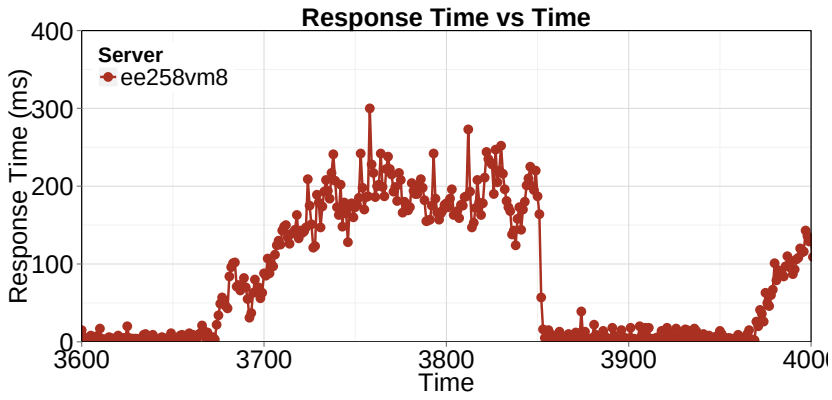


Fig. 7.2.: The plot shows response time of the impacted WS VM over time. Response time here indicate the average time the web server takes to serve a single URL. Interference increases WS response time even though the server is behind a load balancer. The scheduling policy used here is round-robin.

7.2.3 Interference vs. Load

In a separate experiment, we ran CloudSuite on a standalone WS VM (no HAProxy) with varying workload sizes and interference types. Unlike the previous experiment, each run here indicates a 15-minute run of CloudSuite with a specific combination of (workload x interference). The interference benchmark is executed for the entire duration of a run. The results from our experiments is displayed in Fig. 7.3. Note that each data point in the plot indicate the average of all observations during a 15-minute run for a given combination of workload and interference. An interference of type Dcopy-0.0MB indicate no interference was run. It can be seen that the rise in response time due to interference, for a given interference, varies with workload size. For example, with very low workload (< 1000), the impact of interference is negligible and all the plots overlap with each other. However, as the workload size increases (> 1000), the slopes of the curves increase upto a saturation point (2000) beyond which the slope decreases again. In other words, the rise in response time (RT) depends on the server load. Therefore, if the load of a WS can be reduced sufficiently (in this experiment < 1000 clients), then the impact of interference can be reduced (or nearly eliminated). This observation provides a key motivation in the

design of ICE.

In a load-balanced WS setup, reducing load on an impacted WS VM implies for-

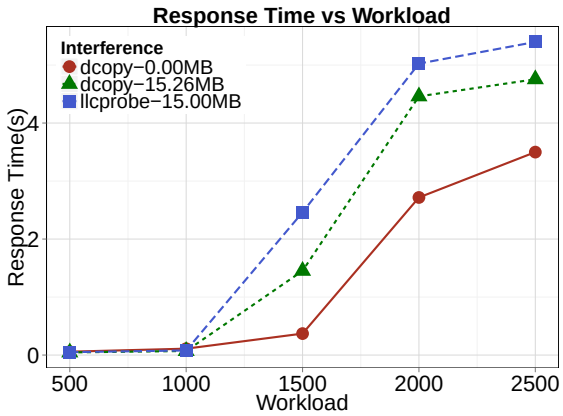


Fig. 7.3.: Increase in response time during interference with varying workload sizes. The X-axis here indicate number of concurrent client connections. The response times shown here are the times to finish an operation (a sequence of URL requests) as described in Cloudsuite. The plot shows interference has greater performance impact with larger workloads and the impact varies across interference benchmarks.

warding fewer client requests to that server. This can be achieved by reducing the scheduling weight of the corresponding server in the load-balancer configuration. To validate this hypothesis, we ran another simple experiment on our Apache+HAProxy setup. Here, we run CloudSuite with periodic interferences as described before. However, the scheduling weight of the monitored WS (WS_X) is set to 100 and 80 (100 being the default value) in two separate runs. It can be seen from Fig. 7.4 that the average response time of WS_X reduces quite significantly while using a lower weight during interferences. In the following section, we describe how we automate the reconfiguration actions in ICE.

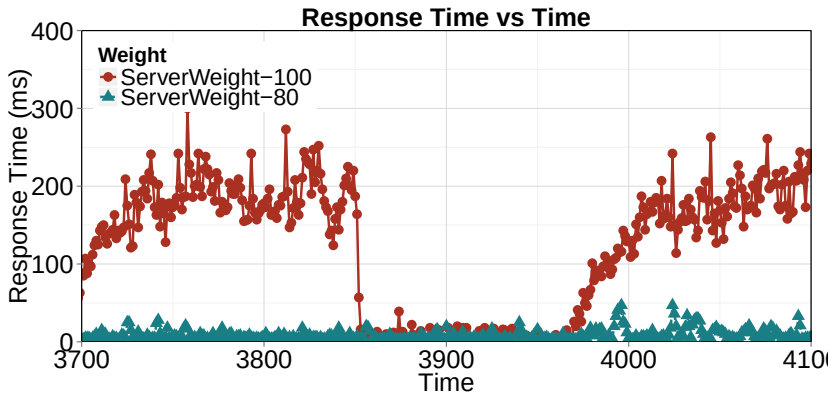


Fig. 7.4.: The plot shows response time of an affected WS VM over time. Changing the Server Weight parameter in the load balancer can reduce its response time significantly during interference.

7.3 Design and Implementation

7.3.1 Overview

ICE primarily consists of four components: i) monitoring engine (ME), ii) interference detector (DT), iii) load balancer configuration engine (LBE), and iv) web server configuration engine (WSE) as shown in Fig. 7.5. These components are coordinated by ICE-Core. Fig. 7.6 shows the high level functioning of this configuration engine. First the monitoring engine collects performance metrics from the web server cluster. These are then fed as inputs to the subsequent modules. In the second stage, the interference detector analyzes observed metrics to find which WS VM(s) are suffering from interference. Once the affected VM(s) are identified, the load-balancer configuration engine then reduces the load on these VMs by diverting some traffic to other VMs in the cluster. We chose the LBE as the first level configuration engine since reconfiguring load balancer has much less overhead than reconfiguring web servers (the first one involves updating global variable(s), whereas, the second one involves creation or destruction of processes). If interference lasts for a long time, the web server configuration engine reconfigures the web server parameters (MXC, KAT, and PHP) to improve response time further. Finally, when interference goes away the load balancer and web server configurations are reset to their default values. Fig. 7.5

shows how these components are distributed across the web server cluster and their dependencies. Note that the exact deployment of a given component is flexible and choices are driven by the need to reduce network and computation overhead. As an example, our *ICE* core is deployed in the hypervisor of the monitored WS VM. We present the details of each module in the following sections.

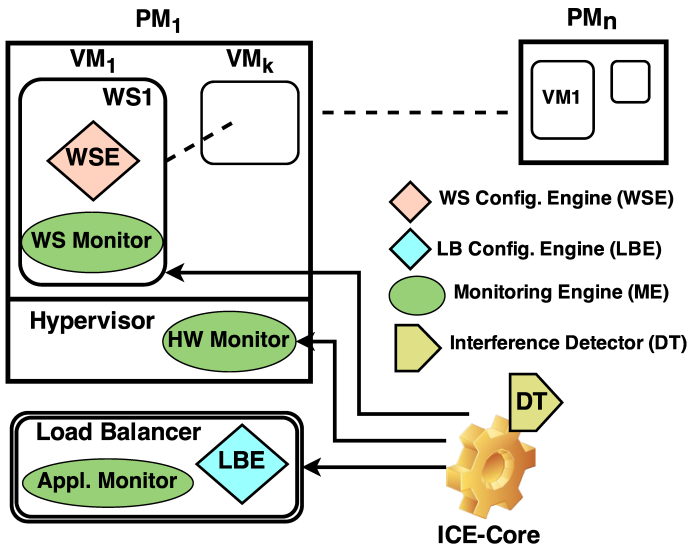


Fig. 7.5.: Components of *ICE* and their deployment.

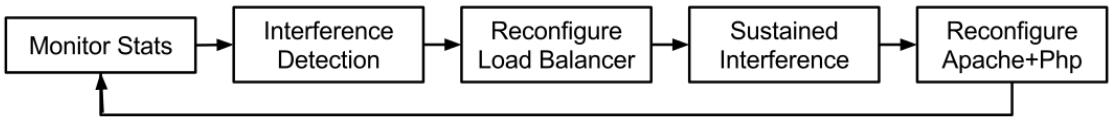


Fig. 7.6.: *ICE* control loop. This figure shows the sequence of steps performed for two-level reconfigurations in *ICE*.

7.3.2 Monitoring

ICE monitors the performance metrics of WS VMs at three levels of the system. The sensor placements are shown in Fig. 7.5. At the hypervisor level, the hardware counter sensor collects values for cycles per instruction (CPI) and cache miss

rate (CMR) for all monitored VMs, whereas, CPU utilization (CPU) of each VM is monitored inside the VM. The application level sensor at HAProxy gives us the average response time (RT), and requests/second (RPS) metrics for each VM. While the hardware counter values are primarily used for interference detection, system and application metrics are used for reconfiguration actions. We chose a monitoring interval of 1 second for all the metrics. During training, we found that system and application metrics give better accuracy with an interval of 5 sec (refer paragraph on Estimating $\xi()$). Therefore, we also maintain a periodic average of 5 sec for those metrics. In the later sections, we discuss how using different measurement intervals help in improving responsiveness and accuracy of various modules.

7.3.3 Interference Detection

We found that the CPI and CMR values of affected WS VMs increase significantly during phases of interference. This is shown in Fig. 7.7 where the red vertical lines (start of interference) are followed by a sudden rise in CPI and CMR. Both these metrics are leading indicators of interference (as opposed to response time which is a lagging indicator) and therefore allow us to reconfigure quickly and prevent the web sever from going into a death spiral (i.e., one where the performance degrades precipitously and does not recover for many subsequent requests). Prior work has also shown that distribution of CPI values can be used to detect interference in web server clusters [50]. In our experiments, we found that during interference CPI values had larger variance than CMR, hence, the latter provided better accuracy for interference detection. This matches with our intuition that for cache intensive interference such as LLCProbe or Dcopy, CMR is a good indicator of interference.

Interference detection in ICE is performed by a Decision Tree (DT) classifier which uses CPI and CMR as the attributes. Our choice of classifier is primarily due to the simplicity and understandability of the clasification rules generated by DT. The classifier is trained using sample runs of the Cloudsuite benchmark. We postpone the

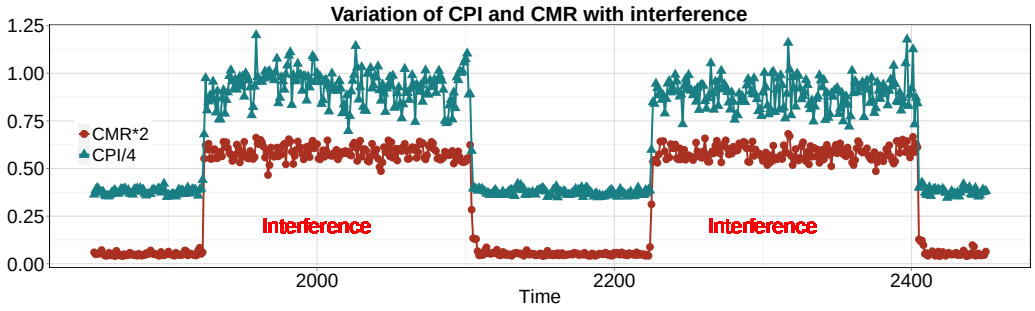


Fig. 7.7.: Variation in CPI and CMR with interference. Metrics are measured with a periodic interval of 1 sec. Actual values are scaled with the factors shown in lables for clarity.

description of training runs until the next section since the same data is also used to train the LB reconfiguration engine. It was found that although both CPI and CMR were used as attributes for training the DT, the final classifier included thresholds on CMR only. This happens because CMR alone is able to classify most samples correctly. The classifier showed a detection accuracy (TP+TN) of 99.12% on the training data using 10-fold cross-validation. In contrast, a DT constructed with CPI only, showed detection accuracy of 98.15%. We used the first classifier (with CMR) for our evaluation experiments.

7.3.4 Load-balancer Reconfiguration

A key component of ICE is the load-balancer configuration engine, which reduces load to an impacted web server by forwarding traffic to other servers in the cluster. In Section 7.2, we found that the impact of interference (rise in response time) on a web server is a function of its load. During high server load, interference creates greater performance degradation since it can easily saturate the CPU [67]. Therefore, the goal of ICE is to reduce traffic to the affected WS so that it does not cross a CPU utilization setpoint (u_{thres}). We achieve this by updating the weight of the corresponding WS VM in the load balancer configuration. Note that, $\#requests$ forwarded to a WS is a function of its scheduling weight. Formally,

$$r = f(w, T),$$

where r is the requests per second (RPS) received by a WS, w is the scheduling weight, and T is total requests received at front-end (HAProxy).

Similarly, it is traditionally known that the CPU utilization (u) of a WS is a function of its load, i.e., $u = g(r)$. [67] also showed that u depends on interference. The degree of interference on a WS can be approximated by its CPI measurements (c). Therefore, $u = h(c)$. We approximate the dependence of u on r and c with the empirical function

$$u = \xi(r, c)$$

During our experiments we also found that CPU utilization u_t at time t is often dependent on the utilization u_{t-1} at time $t-1$. This happens because execution of a task may often last multiple intervals. For example, a request that started executing just before taking measurement for interval t may be served in interval t+1. Therefore, the measurements in intervals t and $t+1$ are not independent. This dependence is captured by taking into account the cpu utilization at the previous interval (we denote this as Oldcpu or o) in our empirical function. The final function for estimating u is represented as

$$u = \xi(o, r, c)$$

Our LB reconfiguration engine works as shown in Algorithm 2:

Assume that CPI, RPS and CPU values at time t are c_t, r_t, and u_t. When the DT detects interference it sends a reconfiguration trigger to LBE. The LBE then computes the predicted CPU utilization ($\hat{u_t}$), given the current metrics c_t, r_t, and u_{t-1}. Notice that we use the estimated CPU utilization $\hat{u_t}$ to compare with setpoint u_{thres} since rise in u often lags behind rise in c. If $\hat{u_t}$ is found to exceed the setpoint, we predict a new RPS value $\hat{r_t}$ s.t. the CPU utilization falls below setpoint. To predict a new load-balancer weight we first compute the percentage reduction in RPS (δ) that is required to achieve this. This is then used to reduce weight w proportionately. Note that during periods of heavy load, the estimated δ may be very high, practically marking the affected server offline (very low w_{new}). To avoid this, we limit the change

in w within a maximum bound (40% of its default weight). In the following sections, we present how to train the estimator function $\xi()$.

Algorithm 2 LB reconfiguration function for ICE

```
1: procedure UPDATE_LB_WEIGHT()
2:     if DT detected interference then
3:         Estimate û_t ← ξ(u_{t−1}, r_t, c_t)
4:         if û_t > u_thres then
5:             Estimate r̂_t, s.t. u_thres = ξ(u_t, r̂_t, c_t)
6:             Compute δ ← (r_t − r̂_t)/r_t
7:             Set weight w_new ← (1 − δ)w_current
8:             Check Max-Min Bounds (w_new)
9:         end if
10:    else
11:        Reset default weight
12:    end if
13: end procedure
```

7.3.5 Collecting Training Data

To collect training data for estimating the function $\xi()$, we ran CloudSuite under various scenarios. Each run involved running the Faban workload emulator for 90 minutes with a constant number of concurrent clients. For each run, we collected application and system metrics as decribed earlier. During a run, we periodically started various interference benchmarks (LLCProbe and Dcopy) every 5 minutes. The interference benchmark was run for the first 3 minutes of this interval followed by no-interference for 2 minutes. To emulate various degrees of interferene, we run LLCProbe and Dcopy with varying number of interference threads (between 2 and 4) and varying array sizes (between 8MB and 750MB per thread). This gave us sufficient variance in the values of CPI and CMR. To collect measurements with varying server loads, we changed the HAProxy weights of the monitored server between 100, 80 and 60 across runs. This allowed us to collect measurements with sufficient variance in RPS values. The benchmark was run for a total of 9 runs amounting to observation

data collected over 13.5 hours. Note that we used this data for training both the Decision Tree and the Estimator. For building the DT, we labeled the observations based on whether an interference benchmark was running at that time or not (we use the class labels IF and NI for Interference and No-interference respectively). Since the WS has different performance characteristics during interference and no-interference, we use only the data collected during interference for training the estimator (note that the estimator is used only when interference is in effect).

7.3.6 Estimating $\xi()$

We approximate $\xi()$ using multi linear regression on variables r, c, and o, where o is a time-shifted version of the vector u (observed CPU utilization). We found that although RPS values were tightly centered around the median there were a few observations when RPS varied widely from the median. We removed these outliers with standard IQR (inter-quartile range) measures. The thresholds for the selected obeservations were chosen as $r > 1Q - 1.5 * IQR$ and $r < 3Q + 1.5 * IQR$, where $1Q$ and $3Q$ are first and third quartile values and $IQR = 3Q - 1Q$. For other metrics, we found the values to have much fewer outliers. The metrics were also scaled to a value between 0 and 1 by normalizing them with their range ($max - min$). The final dataset was fed to the `lm` regressor in R. We measure the accuracy of a regression model by its coefficient of determination or R^2 score. The R^2 score is defined as

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}},$$

where SS_{res} is the residual sum of squares and SS_{tot} is the total sum of squares (proportional to variance of the dataset). Intuitively, R^2 measures how much of the variance in the dataset is captured by the predicted model. Therefore, a higher R^2 score implies a better model. Initially, we found that the metrics collected with 1 sec interval had high variance in RPS and CPU and therefore resulted in a lower R^2 score. On the other hand, using a larger measurement interval (5 sec) averages most of the noises, thereby, giving a better model fit. Based on this observation, we used

the measurements collected at 5 sec period as input to the estimator.

To check if a higher degree polynomial in o, r, and c produces a better fit for the training data, we also ran regression with various degrees. The R^2 scores for models with varying degrees of polynomials are shown in Fig. 7.8. We found that models that include dependence of u with past CPU utilization (i.e., $u \sim (o, r, c)^i$) have better fit than those not considering this dependence ($u \sim (r, c)^i$). It can also be seen from Fig. 7.8, although higher degrees give a better fit, the improvements in R^2 values for degrees > 1 are very low. Moreover, in a linear model, the estimated values of u and r can be computed easily and the model can be updated over time with relative ease. We therefore choose a linear model in (o, r, c) as our estimator. The final prediction function $\xi()$ was computed as:

$$u = 0.06 + 0.64 * o + 0.21 * r + 0.24 * c$$

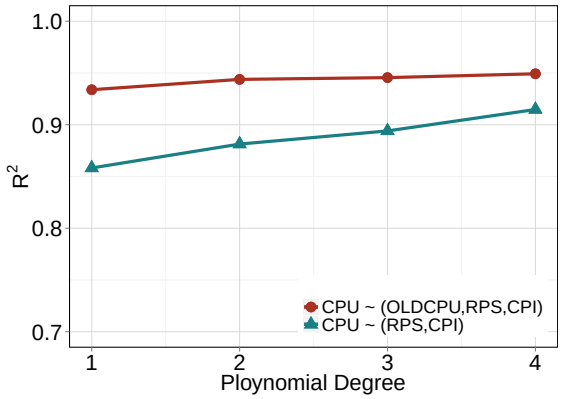


Fig. 7.8.: Accuracy of regression with varying degrees of polynomials. The metrics used for regression are shown in the plot labels.

7.3.7 Web Server Reconfiguration

The final level of reconfiguration in ICE is performed by updating parameters of web server middleware (Apache, Php-fpm, Darwin etc.) hosting an application. Note that reconfiguring middleware parameters is costly since it often involves pro-

cess creation or destruction. Also, the effects of parameter changes are only visible after a lag (usually 5-10 sec). Due to this, ICE performs Web Server reconfiguration only if an interference lasts for a long time (> 20 sec). Our WS reconfiguration decisions are based on a knowledge base described in earlier research. [67] found that during interference the optimal parameters of Apache and Php changes significantly. More specifically, the following reconfiguration actions are necessary to improve WS response time during interference: $MXC \downarrow, KAT \uparrow, PHP \uparrow$, i.e., the optimal values of KAT and PHP increases during interference, while the optimal MXC values decreases. New values for the parameters are computed based on proportinal increase in CPU utilization (MXC) or response time (KAT). We use the controller described in [67] as the WS reconfiguration engine in ICE.

ICE is currently implemented as a Java program (ICE-Core) and a collection of shell scripts distributed across the WS cluster (refer Fig. 7.5). Logically, ICE-Core combines the functionalities of DT, LBE, and WSE in a multi-threaded application. We used the Weka toolkit for generating the DT, while R was used to compute the estimator function $\xi()$. Monitoring and reconfiguration actions are performed with the help of shell scripts and are also driven by ICE-Core.

7.4 Evaluation

We hypothesized earlier that the response time of a web server can be improved during periods of interference by taking two measures: a) by diverting traffic away from the impacted VM and b) by reconfiguring the web server parameters. This is the core design principle of ICE. Therefore, if we find that the average response time of a WS VM during interference is reduced by using ICE, we may conclude that our hypothesis is validated. More specifically, to evaluate the benefits of ICE, we ask the following questions:

i) How much improvement in response time is achieved by using ICE over a statically configured load-balancer?

ii) Does the performance of ICE vary across selection of scheduling policies (WRR and WLC)? If so, how much?
iii) How fast can ICE reconfigure in presence of interference?
We answer each of these questions in the following sections.

Setup. We use the same CloudSuite setup described in Sec. 7.2 to run our evaluation experiments. Only one of the VMs in the web server cluster was subjected to interference as before and we use the metrics collected from the impacted VM to perform our analysis. To quantify the response time improvement with ICE, we compare it against two scenarios: i) when the WS cluster is statically configured (i.e. the LB weight and WS parameters never change), we call this the baseline run, and ii) when the LB is statically configured, but the WS VMs are reconfigured using IC^2. While the improvement over baseline shows the overall benefit of ICE, its comparison against IC^2 shows how ICE outperforms existing solutions. The experiments are repeated with different scheduling policies at HAProxy to answer question (ii) earlier.

Interference Emulation. We emulate interference by running Dcopy and LLCProbe in a periodic manner. The array sizes chosen for our evaluation were LLCProbe (32MB), Dcopy-Low (30MB), and Dcopy-high (3GB) with 4 threads of interference. Note that only a subset of these interferences (LLCProbe 32MB and Dcopy-high 3GB, 4 threads) were present in our training set. To analyze results across interfernce types, only one type of interference is run at any given point in time. Each run of CloudSuite is an emulation lasting 1 hour, during which interfernces are run with a period of 8 minutes. Within each period, an interference benchmark runs for 4 minutes followed by 4 minutes of no-interference.

Data Collection. Note that our experiments involve measuring response time across various combinations of scheduling policies (WRR, WLC) and configuration controllers (Baseline, IC^2, and ICE). Each such combination is considered one experiment. Within one experiment, we run CloudSuite as described above for 5 X 1

hour runs. This gives us enough measurements for each interference type (LLCProbe, Dcopy-Low, and Dcopy-High) to have statistically significant results. The cumulative runtime of the evaluation experiments was more than 30 hours. The metrics observed by ICE during each run are stored in a log file. These are analyzed offline to extract measurements during a given interference run (using the timestamps when interferences are started and stopped). Below, we present the results from our experiments.

7.4.1 Improvement of Response Time due to ICE

The performance of ICE compared to baseline and IC^2 is shown in Fig. 7.9. The plots here show the response times of the monitored WS VM during periods of interference. The starting(stopping) points of interferences are shown with red(green) vertical lines. The magenta(blue) vertical lines show the points when HAProxy(WS) are reconfigured. The texts associated with the blue lines indicate new parameter values for reconfiguration. It can be seen from Figures 7.9(a) and 7.9(b) that the WS response times show very different characteristics depending on scheduling policy. With round-robin, response time of the impacted WS always goes up and the degradation lasts as long as the interference. On the other hand, with least connection scheduling, response time shows occassional spikes. Since least connection scheduling implicitly accounts for server state (`#busy_connections`), an out-of-box load balancer with least connection can mitigate the effects of interference to a large extent. However, in both cases, ICE outperforms a baseline run. With round-robin scheduling, this improvement is markedly visible. With dynamic weight assignment and WS reconfiguration, ICE can reduce response time to the same level as (or better than) a baseline least connection run. It can also be seen in Fig. 7.9(b) that although the response time spikes in baseline are not very high or long-lasting, they persist throughout the entire duration of interference. In contrast, with ICE, after LB reconfiguration these spikes are much fewer. The performance of IC^2 falls midway

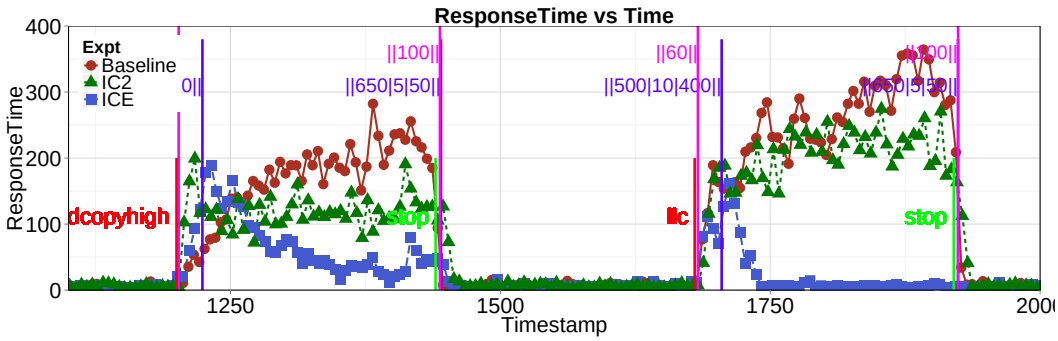
Table 7.1.: Summary of ICE Results. Numbers indicate % change in median response time from baseline runs for different interference benchmarks. The arrows indicate whether response time decreases (↓) or increases (↑).

Scheduling Policy	LLCProbe		Dcopy-High		Dcopy-Low	
	IC^2	ICE	IC^2	ICE	IC^2	ICE
WRR	29%↓	94%↓	18%↓	92%↓	59%↑	82%↓
WLC	5%↑	39%↓	15%↑	21%↓	35%↑	25%↓

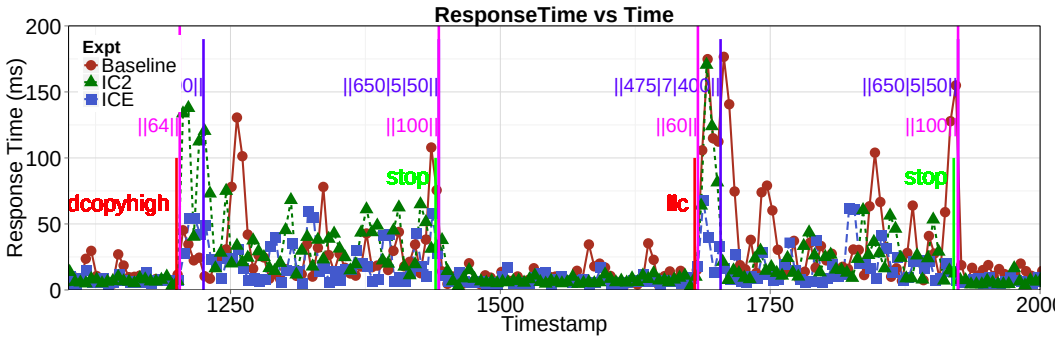
between a baseline run and ICE. Although IC^2 is able to reduce the response time during stronger interferences, the reduction is not as significant as ICE. Moreover, a baseline least-connection run outperforms IC^2 with round-robin. This is because of the fact that reducing MXC as in IC^2 is an indirect and milder way of reducing load on an impacted WS, while using load-balancer is a more direct way of preventing server overload.

Fig. 7.10 shows the median response time of the monitored WS VM across interference types with various scheduling policies. We find that, when using a WRR scheduler, ICE shows an improvement of 82-94% over baseline response time while improvement with IC^2 is 20-30%. Across interference types, LLCProbe, which is the strongest interference, shows the most performance benefit. Interestingly, with Dcopy-Low IC^2 shows performance degradation.

With WLC scheduling the performance benefits are significantly less compared to round-robin. WLC by itself is able to direct some traffic away from the congested WSs and therefore help mitigate interference to a significant extent. However, using ICE alongside WLC improves response time even further. Across interference types this improvement varies between 21 and 39%. However, using IC^2 alongside WLC shows poorer performance compared to baseline (degradation of 5-35%). This is primarily beacuse of the fact that, even though the WS is reconfigured to handle fewer clients the LB has no knowledge of it and it continues sending nearly the same number of requests to the WS as before. This results in increased overhead of queuing or connection dropping, thereby, increasing processing times over existing connections.



(a) Scheduling policy is round-robin



(b) Scheduling policy is least connection

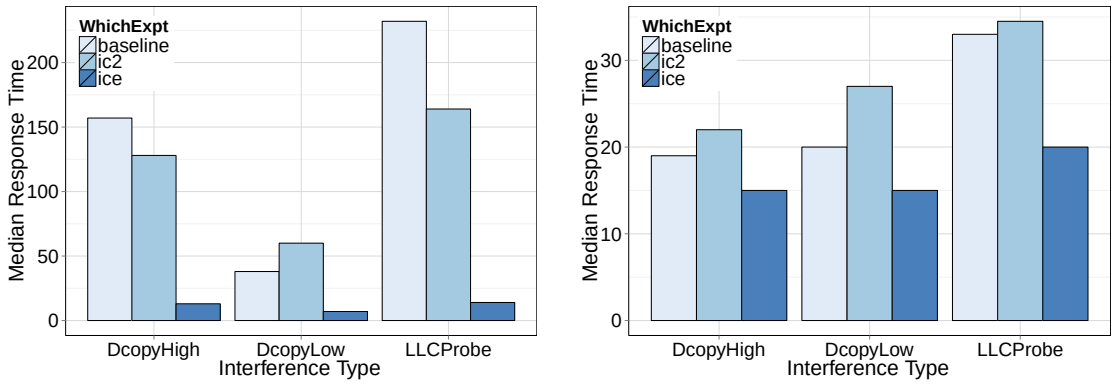
Fig. 7.9.: Response time over time. ICE improves response time significantly compated to baseline and IC^2 with round-robin scheduling. With least connection the lines are not clearly distinguishable, however, median response time is best with ICE.

This effect is most prominent with WRR scheduling and DcopyLow benchmark (58%) in IC^2, where the effect of interference is much less than stronger interferences. The response time values during our experiments may be seen in Table 7.1.

To summarize, we found that using WLC as a scheduling algorithm instead of WRR can mitigate much of the performance degradation during interference. ICE outperforms baseline LB with both types of scheduling (WRR and WLC). It also outperforms IC^2 by a large margin.

7.4.2 ICE has Low Detection Latency

To measure how fast ICE can mitigate the effects of interference, we computed the detection latency of ICE from our evaluation runs. Note that detection latency



(a) Load balancer scheduling policy is round-robin

(b) Load balancer scheduling policy is least connection

Fig. 7.10.: Median Response time with IC^2 and ICE for various interferences against a statically configured load balancer (baseline). Note that baseline LC is able to reduce response time significantly compared to baseline RR. Response time with IC2 increases in LC (also with Dcopy-Low) due to overhead of dropped connections.

defines the duration between the onset of an interference and the first reconfiguration action at HAProxy in response to that interference. This includes the performance of the DT classifier (how quickly and accurately it can detect interference) as well as any lag between detection and the corresponding reconfiguration action. Since HAProxy is only configured in the runs with ICE enabled, only these samples are used for measurement of detection latency. We found that across interference types, ICE had a median detection latency of 3 sec (the maximum being 4s). We observe that this is a very fast response considering most interferences in public clouds last for 10s of seconds. This is also much faster compared to IC^2 where reconfiguration happened with a delay of 15-20 sec. In IC^2, the authors didn't have access to HW counter data and they had to rely on CPU Utilization and Response Time which increases with a delay, whereas, in ICE, use of hardware counters allows us to have much lower detection latency.

7.4.3 Performance of Classifier

To evaluate the accuracy of the classifier we also ran it on the data collected during our evaluation experiments. The collected data was labeled based on whether an interference benchmark was running. We found that the decision tree showed an accuracy (TP+TN)/(Total Samples) of 99.6%. Notice that the decision tree generated in our evaluations of *ICE* has ranges on a single attribute (CMR) (7.3). It is therefore identical to a threshold-based detection, therefore the cost of classification is negligible.

We found that *ICE* incurred about 10% cpu overhead in the HAProxy load balancer for calculating average response time. Note that the total cpu utilization of the HAProxy was less than 60%. Therefore, our monitoring overhead does not skew the results. This can be eliminated by moving the response time from HAProxy to a dedicated *ICE* VM. The monitoring overhead of other metrics (sysstat + ocount) was negligible.

So far, we have described the implementation of *ICE* only in the context of CloudSuite setup. However, the general concepts used here can also be used to adapt to other server setups and other types of workloads. We validate this by showing a set of simple experiments on the Darwin streaming media server in the next section. Darwin and Apache+Php are two very different server frameworks (the first one optimized for delivery of media streams, i.e. static content, while the second one is optimized for generating dynamic responses) and they have different workload characteristics (the former has long client sessions, while the latter is usually short). Therefore by validating *ICE*'s design principles on both of them, we provide strong evidence of generalizability.

7.5 Streaming Server Evaluation

We evaluated *ICE*'s capabilities for widely used open source **Darwin streaming server** [70] (originally developed by *Apple*), which shares the same code base as **QuickTime** streaming server.

7.5.1 Monitoring and Performance Metrics

For video streaming server, *ICE* uses sensors in each VM to monitor CPU utilization and CPI and CMR measurements are collected from the hypervisor. Video streaming servers do not follow a strict request-response pattern during its operation. Typically clients requests for videos and some protocol messages are exchanged before the server starts streaming the videos to client devices. In the absence of strict request-response pattern, we define a metric called *frame-delay* to quantify the performance of the streaming server. While streaming, the server sends video frames to multiple clients. Based on the *frames per second (fps)* playing rate of the videos and clients' remaining buffer size, the server calculates an *expected* sending time for each frame. The time difference between when a frame was actually sent by the server to its expected sending time, is called the frame-delay. Under normal circumstances, this delay should be non-positive, i.e. the server will send the frame no later than its expected time for sending. When the server is overloaded or in the presence of interference, server will not be able to sustain its performance and frame-delay will increase. In our setup, the streaming server was instrumented to output average frame-delay encountered by all the clients in every second and we use that to quantify the performance of the streaming server.

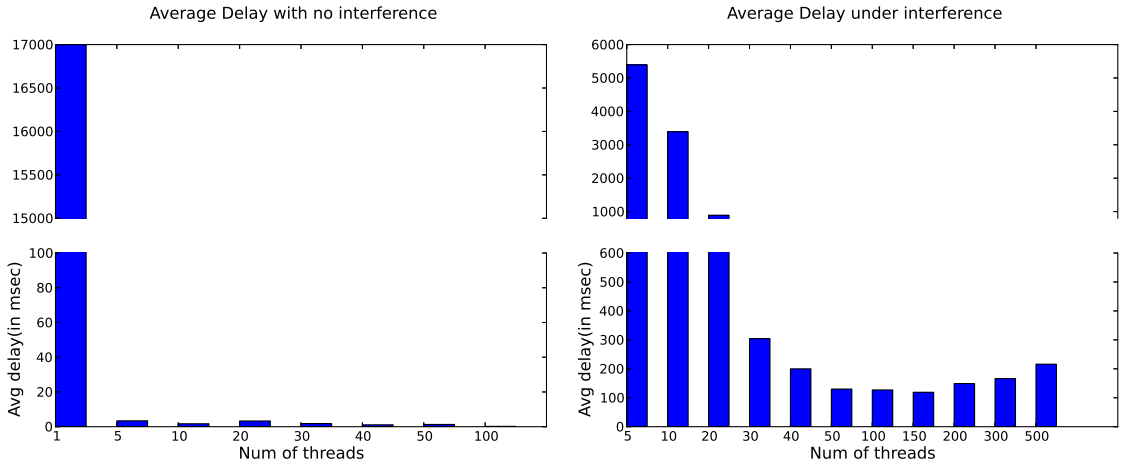
7.5.2 Experimental Setup

In our private-cloud, we set up two Streaming Server(SS) VMs in two physical machines and put a load-balancer in front. Each SS VM (we also call real-servers) was

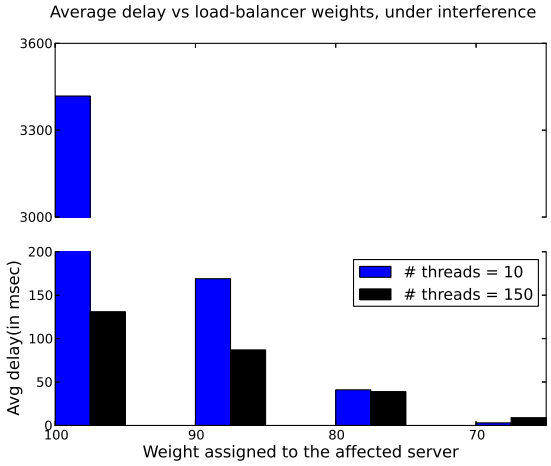
allocated 2-vcpus and 2GB of RAM. SS typically communicates using RTSP protocol and sends data using RTP transport protocol. Since majority of the RTP implementation is built on top of UDP, we use Linux Virtual Server (LVS) as our load-balancer as it can handle both UDP and TCP traffic. Specifically we used the Direct Routing (DR) configuration for LVS for our load-balancing purpose. We used CloudSuite's media streaming benchmark [71] to test the performance of media streaming servers. CloudSuite's media streaming benchmark clients emulates real-world user requests for different videos using Faban engine. Benchmarks clients request various videos at different resolutions. Some videos are requested more frequently by the clients than others to emulate popular videos. Client also stops watching (by closing the connection) after different duration - a real-world usage pattern found in many studies [72] [73]. Using CloudSuite-benchmark we emulated 10,000 clients in a separate physical machine connected to the same LAN as the load-balancer. We started with a weight of 100 to each machine so that the LVS load-balancer divides the client load equally. Thus each SS gets request for videos from 5000 clients. In all the experiments, we used default values for all the configuration parameters for Darwin streaming server, except `maximum_connections` (maximum number of allowed RTSP connections) and `maximum_bandwidth` (maximum allowed bandwidth usage), which we set to `-1`, i.e. removing any restrictions. For each experiments we used average values from 3 sets of experiments.

7.5.3 Results

a) Variation of frame-delay with number of threads: without interference In this experiment, we wanted to find the optimum number threads that is sufficient for handling 5,000 clients connected to each SS. We modified the `run_num_threads` (number of threads created by the server to handle concurrent connections) configuration parameter. We used 600 seconds of steady state period for the Faban client emulation engine. We found that memory foot-print of the SS is not significant (less



(a) Frame-delay - normal condition (b) Frame-delay - with LLC interference



(c) Frame-delay vs load-balancer weights

Fig. 7.11.: Two replicas of media Streaming Servers behind a load-balancer is serving 10,000 clients simultaneously. (a) Shows the variation of frame-delay with number of threads when no interference is present. With just 1 thread, the server gets extremely overloaded hence it calculates a very high expected delay for the frames. With 10 optimum number of threads, the server shows negligible delay. (b) Shows the variation of frame-delay with number of threads with LLC interference. With 150 optimum number of threads, the server shows minimum frame-delay. (c) Shows how frame-delay of the affected server changes with load-balancer weights - when one real-server is under LLC interference. Two plots show how delay changes for optimum number of threads calculated for both interference and non-interference cases

than 1 GB) and it is mostly CPU bound. As Fig. 7.11(a) shows, we found when only 1 thread is used, the server experiences massive frame-delay. But the delay quickly drops and we found 10 is the optimum number of threads giving almost negligible

delay. For higher number of threads, frame-delay remained negligible and we did not observe any significant increase due to context switching overhead.

b) Variation of frame-delay with number of threads: with LLC interference

We wanted to find how the streaming server behaves under cache-interference and what would be the optimum number of threads under such situation. Even if memory foot-print of Darwin streaming server is not significant we found LLC interference causes a significant disruption in the service. We created LLC interference by running LLC-probe benchmark on another co-located VM (similar to WS experiments). As shown in Fig. 7.11(b), even with optimum number of threads, i.e. 10, found under normal usage, the SS experiences significant frame-delay (3,200 msec). As we increase the number of threads, we found 150 is the optimum number where we observed the least amount of frame-delay. Beyond that we observed slight increase in frame-delay. Thus, in the presence of interference, the optimum number of threads differs from normal operating environment.

c) Variation of frame-delay with reconfiguration of the load-balancer

Similar to WS, interference in media streaming servers can be detected early from the change in CPI and CMR. Once interference is detected, weights in the load-balancer can be dynamically reconfigured for reducing frame-delay. In this experiment, we show how frame-delay improves as we decrease the relative weight of the affected SS. We ran 2 sets of experiments, one with optimum number of threads (10 threads) found under normal condition and another with optimum number of threads (150 threads) found when the Streaming Server's performance is affected by interference. As can be seen in Fig. 7.11(c), as we decrease the relative weight of the affected SS (i.e. move the clients to the other server), the frame-delay metric improves. For relatively higher weights (100 and 90), using optimum number of threads for interference scenario gives better performance. But with further decrease in weight, performance of the Streaming Server converges for both type of optimum thread counts. With a relative weight of 70, the frame-delay of the affected SS becomes almost negligible. In fact, at this point, performance with 10 threads is marginally better than performance

with 150 threads. We interpret that such behavior is due to higher context switching overhead with 150 threads. Thus, the same design principle of ICE applies to two very different kinds of servers (web server and media streaming server) to improve on the metrics of interest.

7.5.4 Discussion: Advanced Streaming Techniques

While naive load-balancing technique by redirecting new requests for videos away from the affected server will bring down the frame-delay in a highly dynamic (i.e. lot of new requests coming in every second, and old streams end) streaming service. The responsiveness of the mitigation mechanism would work best if a significant percentage of the streams are short lived, which is the case in reality [73]. However, our technique will work best (for both short and long lived streams) if state-of-the-art streaming techniques such as Dynamic Adaptive Streaming over HTTP (DASH) [74] is used. By augmenting basic DASH protocol (as done by [75]), it is possible to request the next *slices* of videos from a different server which is not affected by interference. Thus, even for long running streams we can effectively migrate the load to a different server which in turn will improve the frame-delay of the affected server as shown by Fig. 7.11(c). We leave implementation of such augmented DASH based load-balancing as our future work.

Summary

From the experimental evaluations in Sections 7.4 and 7.5, we have the following three key take-aways:

1. An off-the-shelf Load Balancer can mitigate the effects of interference in VMs significantly by using the Least Connections scheduling strategy.

2. ICE improves the WS performance in the face of interference further, by detecting interference quickly and dynamically adjusting the weights of the Load Balancers.

3. These results are generalizable to a very different kind of server (media streaming server) with a different metric of interest.

7.6 Directions for Future Research

In the previous sections, we presented the design and implementation of ICE, a two-level reconfiguration engine for web server clusters, which mitigates interference by reconfiguring load-balancers as well as web services middleware. We found that ICE showed significant improvement in web service response time compared to IC^2. However, the effectiveness of ICE is dependent on the length of user sessions. In Chapter 5, we showed that once a client session is established on a server X, all subsequent requests in the same session go to server X irrespective of load-balancer policy. This is especially bad for media streaming servers where a client's stream can be active for a long time. Moving such long-lasting sessions from one server to another during interference is part of our future work as described below.

Improving Effectiveness of ICE for Long-lasing Sessions: To be able to move client sessions from one WS VM to another without interruption we need to store session specific data in a distributed datastore such as memcached. In case of media streaming server, the server must also support advanced streaming techniques like DASH (refer Section 7.5.4). We are currently finding techniques for integrating DASH with Darwin streaming server.

8. RELATED WORK

In this chapter, we present some of the earlier research work related to this dissertation. We have already introduced few of these solutions in Chapters 2 and 5 to build the motivations for our work. Here we revisit those in greater details and add comparison of our work with other related research domains. We organize the chapter into a set of research topics so that it may be easily compared with our work in earlier chapters. While Section 8.1 is primarily related to our work in Chapter 3, Sections 8.2–8.3 are related to our work in Chapter 4. In Sections 8.4–8.5, we present research related to our work in Chapters 6 and 7.

8.1 Operating System Reliability

The goal of research in software reliability analysis has been to classify software errors, as well as to characterize various properties of failures. Such characterization enables us to not only assess the effects of failures but also to prevent and detect new bugs. Reliability literature over the years contains the results of many research efforts directed at analyzing bug reports for popular operating systems [5, 6, 76–78]. In one of the early works on OS reliability, Sullivan et al. [76] analyzed defects of the MVS operating system based on empirical failure records documented by field IBM staff. This work categorized defects as overlay (errors that corrupt memory) and regular (those that do not corrupt memory). The frequency and effects of both types of errors were analyzed. Chou et al., in [5], presented their finding on OS errors by compiler attachments, which checks code for certain types of bugs, and counts bug density. The work discovered the correlation of different types of bugs with directories, function size, and file age. Our work looks into similar problem with different scope: instead

of compiler attachments, our research is based on the developers' view and how bugs are fixed.

In another work on software reliability, Chandra et al. [78] contradicted a popular belief that most application faults can be tolerated using generic techniques such as process pairs. They categorized faults into 3 types: environment-dependent transient, environment-dependent non-transient, and environment-independent. It was found that only 5-14% of the faults were triggered by transient conditions and 72-87% of faults were environment-independent. A similar trend is observed in our analysis where an overwhelming majority of the bugs (more than 90%) are permanent.

In [77], Liang et al. highlighted the failure characteristics of BlueGene/L supercomputer by correlating data obtained from event loggers. Applying similar concept of failure event loggers, Cinque et al. [7] performed one of the few pieces of work that focuses on mobile OS reliability. In this work, the authors attached fault event loggers to a set of 25 Symbian OS based mobile phones to record failure events and panics (kernel-generated warnings for Symbian). Through this, the authors unveiled characteristics of the panics (burst, etc.) and the relation between panics and user-visible failures. However, since Symbian was not open-sourced at the time of this analysis, their research was limited to the manifestation of failures. Our work in Chapter 3 extends the scope beyond these findings by classifying failures according to their root causes in the source code (for Android). We also present an analysis on customizability and complexity of mobile OSes which is distinct from previous work.

8.2 Robustness Testing

Robustness evaluation of software systems is broadly categorized into functional and exceptional testing. Functional testing [79] employs generation of expected test inputs with the intention of checking the functionality of a software module, while exceptional testing employs generation of specially crafted test inputs to crash the system in order to check its robustness. Generated input test data can be random,

a pure fuzz approach [80], or semi-valid (intelligent fuzzing) [81, 82]. UNIX utilities were first fuzzed by Miller et al. [80] by feeding random inputs to show that 25-33% of utility programs either crashed or hanged on different versions of UNIX. This simple technique has caught a variety of bugs like buffer overflows, unhandled exceptions, read access violations, thread hangs, memory leaks, etc. A later work by the authors [83] showed that robustness of UNIX utilities improved little over five years. A study [84] of similar nature on Windows NT and Windows 2000 showed their weakness against random Win32 messages, while, blackbox random testing on MacOS [85] reported a considerable lower failure rate (7%). Our research extends these works to a mobile platform where we fuzz the ICC of Android and show a variety of exception handling errors. In terms of knowledge about the target application (i.e. whitebox [82, 86] vs. blackbox testing [87]), our tool takes a combined approach (blackbox for explicit Intents and whitebox for implicit Intents).

Fuzz tools reported in literature can also be classified based on their input generation techniques and their intrusiveness. The input data produced by a fuzzer tool may be either generation based or mutation based [88]. Generation based fuzzers generate test inputs based on specification of a protocol or an API to be tested while mutation based fuzzers rely on capturing and replaying a mutated version of valid input. Our tool (JJB) in Chapter 4 falls under generation-based fuzz tools, as it generates input data, i.e., Intents conforming to Android Intent API specifications. JJB is also intelligent in that it has knowledge of Android APIs (e.g. known `Action`, `Category`, and `Extras` strings) and partial knowledge of the target applications (e.g. Intent-filters). Fuzzing tools typically produce input received across trust boundaries [89], i.e., Runtime-OS and Application-Runtime boundary. At a lower layer, fuzzing can be done at Runtime-OS interface as shown by [90]. Another similar work, Ballista [81], identified ways to crash operating systems with a single function call at Runtime-OS boundary. At a higher layer, fuzzing can be done at Application-Runtime boundary where runtime is responsible for validating data. In this work, we

fuzz at Application-Runtime boundary with the aim of crashing Android runtime by fuzzing *Intents* that are passed between application components.

8.3 Smartphone Reliability and Security

A malformed Intent delivered to a receiver through ICC exposes attack surfaces as pointed out by [36], example vulnerabilities being triggering of components that are unintentionally exported by a developer (i.e., an Intent spoof) or unauthorized receipt of an implicit Intent by malicious component. ComDroid [36], a *static* analysis tool, detects these two vulnerabilities in Android applications. We narrow down these attack surfaces to a set of input validation errors by *runtime* testing, however, actual exploit of these errors may require combining these with other vulnerabilities (e.g. improper permission assignment). Our approach discovers vulnerabilities in the application components, but, we do not provide exploits to use these vulnerabilities from an external source, i.e., we do not show external requests that will generate malformed Intents for actually exploiting these vulnerabilities. That is part of our ongoing work.

Other work on Android security looked at permission assignment of applications, misuse of sensitive information [91], and provided future directions for application certification [92]. Our work does not directly detect privacy leaks, but can be used for giving insight to good application design practices (specially input validation). These practices in turn can be incorporated in an application certification process that is geared towards improving application robustness.

8.4 Autonomous Configuration Management

Configuration management of complex software systems has been an area of active research over the last decade leading to many interesting solutions to the *correctness* problem [93–97]. Most of these approaches try to identify causality between a configuration event and a detected failure and then predict (correct) the error [93–95]. Yin

et al. presented a wealth of real-world examples of configuration errors from both commercial and open-source applications (CentOS, MySQL, Apache, and OpenLDAP) in [66] which served as a key motivation of our work. The authors found that parameter mistakes accounted for 70-85% of the configuration errors. However, all of this work focuses on correctness errors and in non-virtualized settings, whereas we focus on performance configuration in a virtualized setting.

In one of the early works on tuning of Apache servers [98], Liu et al. showed that `MaxClients` exhibits a concave upward behavior on response time—an observation that led to the design of a tuning agent using hill-climbing algorithms. Our results also highlight this behavior of `MaxClients`, but we find that the gradients of the curves change frequently due to interference and dependence on other parameters. In another work [20], Diao et al. presented a multi-input multi-output (MIMO) feedback control for optimizing web server performance, however, their controller only considers stabilizing CPU and Memory utilization with changes in workload intensity—one of the many challenges we present in our work. A more recent work [99] looks into automatic generation of configuration files in multi-tier web servers. The authors considered dependency with a different set of configuration parameters, however, their work is in a non-virtualized setting where reconfiguration events are infrequent (addition of physical nodes).

The question of how to performance tune applications that are executing in virtualized environment has been addressed by several prior works. Such approaches were applied to configuration of software systems like Apache server [94], application server [100], database server [101] and online transaction services [99, 102]. Some of these consider coordinated tuning of resources allocated to VMs and associated application configurations [101, 103]. We note that doing so is disadvantageous for two reasons—i) it increases the exploration space, ii) it underutilizes a wealth of research on cloud hypervisors. In our work, we assume the cloud provider already employs sophisticated provisioning algorithms and we use resource changes as a trigger for reconfiguration.

8.5 Performance Interference in Clouds

The issue of interference in virtualized environments has been pointed out by several researchers [12,13,104,105] and some efforts have been made for providing better resource isolation [12,105,106]. However, due to the intrusive nature of these changes and the impact on the performance, today's production virtualized environments still do not provide isolation for cache usage and memory bandwidth, which are relevant to the results that we presented here.

Effect of interference on application performance has been evaluated by many researchers over the years [11, 13, 48, 49, 106]. In terms of contended resources, the papers can be classified as either related to memory interference (cache, memory-bandwidth) [13,48,49], or network interference [11,106,107]. While most researchers have looked at performance isolation (or the lack of it) in a security oblivious manner [11,48,49,106], others have pointed this as a source of vulnerability in the public cloud [13,52]. In most cases, the performance degradation observed is severe enough to encourage implementation of sophisticated mitigation strategies. Among the mitigation strategies, better scheduling [14, 16, 17] and live migration [15, 49] are most common. However, each of these solution strategies has its shortcomings. Firstly, a VM's resource usage pattern may change over time, often unpredictably. A consolidation manager cannot foresee such usage changes without knowing of the applications running within the VM, and that is usually considered too intrusive and hence, not made available to the consolidation manager. Secondly, both types of solutions operate at a level which is beyond the scope of an end-customer. Solutions that use live migration for avoiding interference have their drawbacks as well. In [46], the authors show that VM live migration is very resource intensive, especially when the source server is highly loaded. Live migration in such a scenario is often long drawn and fails frequently.

In this work we try to mitigate the effects of performance interference using intelligent reconfiguration of load balancers and web servers. We observe that Live migra-

tion is an example of infrastructure reconfiguration and usually takes several minutes to complete. In comparison, our work *ICE* is an example of application adaptation and may be finished in seconds. Our work is also partially related to research on building adaptive web servers using intelligent configuration engines [20, 98–101, 103]. However, most of these are evaluated in a non-virtualized environment and the reconfiguration actions are performed in the order of minutes. This is unsuitable in a cloud environment, where transient interferences may render the choice of a given parameter value sub-optimal.

9. LESSONS LEARNED

To answer the questions raised in Section 1.2, we have taken several steps to evaluate and improve the dependability of smartphones and cloud-based applications. We have described our design, implementation, and results in Chapters 3–7. Here, we present a summary of our findings from earlier Chapters.

9.1 Study of Failures in Android and Symbian

In Chapter 3, we presented a measurement based failure analysis of two operating systems—Android and Symbian—by studying publicly available bug databases. The key findings are: (1) Most of the bugs (more than 90%) in both these platforms are permanent in nature, suggesting that the codebases are not yet mature. (2) The Kernel layer in both the platforms is sufficiently robust, however, much effort is needed to improve the Middleware layer (Application Framework and Libraries in Android). (3) Development tools, Web, Multimedia, and Build failures are most prevalent in both the platforms. This suggests the necessity for better mobile application development tools and need for caution in using third-party libraries. (4) Android offers a great degree of customizability in both the build and the execution processes. This customizability comes at a cost for a significant fraction of bugs—between 11% and 50% (assuming all of Modify settings, Add/modify cond, Preprocess changes, and Major changes are due to customizability). At present, the percentage of build errors is also high in Symbian (38.6%). (5) According to our analysis, a significant minority of the bugs in Android (22%) needed major code changes. Among various types of code modifications, fixing configuration parameters and control flow update (adding if-else clause) are most widespread.

9.2 Evaluation of Robustness of Android ICC

In Chapter 4, we have successfully conducted an extensive robustness testing on Android's Inter-component Communication (ICC) mechanism by sending a large number of semi-valid and random Intents to various components across 3 versions of Android. Our learnings from this fault injection campaign are many, most prominent ones being: 1) Many components in Android have faulty exception handling code and `NullPointerExceptions` are most commonly neglected, 2) It is possible to crash Android runtime by sending Intents from a user-level process in Android 2.2, 3) Across various versions of Android, 4.0 is the most robust so far in terms of exception handling; it, however, displays many environment dependent failures.

Based on our observations, we have highlighted the guideline that any component that runs as a thread in a privileged process should be guarded by explicit permission(s). We have also proposed several enhancements to harden implementation of Intents; of these, subtyping in combination with Java annotations can be easily enforced.

9.3 Mitigating Interference using Middleware Reconfiguration

In Chapter 6, we investigated one of the major sources of performance variability in clouds, namely, interference and presented ways in which an end-customer can mitigate its ill-effects. More specifically, we evaluated the frequency and impact of interference in public clouds like Amazon EC2. Our experiments suggest that performance anomaly due to interference is a reality. We designed and evaluated an interference-aware application configuration manager (IC^2), which is able to detect interference and find suitable parameter values during these phases. Through an extensive set of experiments, we also identify the challenges associated with finding optimal application parameter values during interference.

IC^2 solves three key challenges for dynamic reconfiguration—first, it presents a machine learning based technique for detecting interference; second, it uses a

heuristic-based controller for determining suitable parameter values during periods of interference; and finally, it reduces the cost of reconfiguration of standard Apache distributions by implementing an online reconfiguration option in the Httpd server. Our solution reduced response time of CloudSuite, a dynamic web application benchmark, by upto 29% in EC2 and 40% in a private cloud testbed during periods of cache interference.

9.4 Handling Interference by Two-level Reconfiguration

In Chapter 7, we presented a two-level configuration manager for web server clusters which can mitigate effects of performance interference in cloud. Our solution called ICE consists of a decision engine, a load-balancer configuration engine, and a web server configuration engine. The decision engine detects interference with the help of a decision tree (trained on hardware performance counter measurements). We found that the decision engine can detect and reconfigure very fast ($\sim$ 3s). The load balancer configuration engine uses an estimator to predict how much reduction in request rate is required to improve response time.

We observed that, an off-the-shelf Load Balancer can mitigate the effects of interference in VMs significantly by using the Least Connections scheduling strategy. However, ICE improves the WS performance in the face of interference further, by detecting interference quickly and dynamically adjusting the weights of the Load Balancers. The proposed configuration controller (ICE) improves the median response time of the WS by upto 94% compared to a static configuration. We find that ICE also gives better resposne time by upto 39% compared to an adaptive load balancer (least-connection). We also show the applicability of ICE to a media streaming server (Darwin). This shows that ICE can be generalized to other cloud services that do not use Apache+Php middleware.

REFERENCES

REFERENCES

[1] Gartner, "Gartner says worldwide traditional pc, tablet, ultramobile and mobile phone shipments to grow 4.2 percent in 2014," http://www.gartner.com/newsroom/id/2791017, July 2014.

[2] IDC, "Idc: 87% of connected devices sales by 2017 will be tablets and smartphones," http://www.forbes.com/sites/louiscolumbus/2013/09/12/idc-87-of-connected-devices-by-2017-will-be-tablets-and-smartphones/, September 2013.

[3] Google Play Store Crashes or Refuses to Open, http://forums.androidcentral.com/google-nexus-7-tablet/246334-google-play-store-crashes-refuses-open.html.

[4] B. Treynor, "Todays outage for several google services," http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html, January 2014.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 73–88. [Online]. Available: http://doi.acm.org/10.1145/502034.502042

[6] W. Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," in *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, 2003, pp. 459–468.

[7] M. Cinque, D. Cotroneo, Z. Kalbarczyk, and R. K. Iyer, "How do mobile phones fail? a failure data analysis of symbian os smart phones," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 585–594. [Online]. Available: http://dx.doi.org/10.1109/DSN.2007.54

[8] R. Golijan, "Fridge magnet poses security threat to ipad 2," http://www.technolog.msnbc.msn.com/technology/technolog/fridge-magnet-poses-security-threat-ipad-2-119905, April 2012.

[9] Details on Facebook Outage 2010, https://www.facebook.com/note.php?note_id=431441338919&id=9445547199.

[10] Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region, http://aws.amazon.com/message/680587/.

[11] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, ser. MMSys '10. New York, NY, USA: ACM, 2010, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/1730836.1730842

[12] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. USENIX Association, 2007, pp. 1–18.

[13] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: improve your cloud performance (at your neighbor's expense)," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 281–292. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382228

[14] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms," in *Proceedings of the 3rd international conference on Virtual execution environments*, ser. VEE '07. New York, NY, USA: ACM, 2007, pp. 126–136. [Online]. Available: http://doi.acm.org/10.1145/1254810.1254828

[15] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *USENIX ATC*, 2013.

[16] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 77–88. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451125

[17] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 237–250. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755938

[18] R. Olio, "Olio deployment," 2010, http://radlab.cs.berkeley.edu/wiki/Olio_Deployment.

[19] EPFL, "CloudSuite," http://parsa.epfl.ch/cloudsuite/cloudsuite.html.

[20] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury, "Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server," in *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pp. 219–234.

[21] "Dalvik virtual machine," http://www.dalvikvm.com/, 2008.

[22] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual machine showdown: Stack versus registers," *ACM Trans. Archit. Code Optim.*, vol. 4, pp. 153 – 163, January 2008.

[23] "What is android?" http://developer.android.com/guide/basics/what-is-android.html.

[24] Symbian OS on Wikipedia, http://en.wikipedia.org/wiki/Symbian_OS.

[25] Symbian System Model, http://developer.symbian.org/wiki/index.php/Symbian_System_Model.

[26] Android Bug Listing, http://code.google.com/p/android/issues/list.

[27] http://developer.symbian.org/bugs/.

[28] T-Mobile G1 Forum, http://forums.t-mobile.com/tmbl/?category.id=Android.

[29] Android Code Review, https://review.source.android.com/Gerrit#all,merged,n,z.

[30] Android Source Code Repository, http://android.git.kernel.org/.

[31] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, Nov. 1992. [Online]. Available: http://dx.doi.org/10.1109/32.177364

[32] Understand 2.0: A source code analysis tool, http://www.sci-tools.com/products/understand/.

[33] Symbian Source Code Repository, http://developer.symbian.org/main/source/packages/index.php.

[34] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *Computers & Security*, vol. 26, no. 3, pp. 219–228, 2007.

[35] Android-Central, "Why are my google play services crashing after i did an update?" http://forums.androidcentral.com/general-help-how/445114-why-my-google-play-services-crashing-after-i-did-update.html, October 2014.

[36] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 239 – 252.

[37] "Intent fuzzer," http://www.isecpartners.com/mobile-security-tools/intent-fuzzer.html.

[38] "Intent class overview," http://developer.android.com/reference/android/content/Intent.html.

[39] T. Register, "Amazon's cloud on track for $2bn in revenue in 2013," http://www.theregister.co.uk/2013/04/26/aws_revenue_analysis/.

[40] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *ACM SIGMETRICS Performance Evaluation Review*, vol. 35, no. 2, pp. 42–51, 2007.

[41] C. A. Waldspurger, "Memory resource management in vmware esx server," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 181–194, 2002.

[42] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. Usenix NSDI*, 2005.

[43] R. Koller, A. Verma, and R. Rangaswami, "Generalized ERSS Tree Model: Revisiting Working Sets," in *IFIP Performance*, 2010.

[44] L. Cherkasova and R. Gardner, "Measuring cpu overhead for i/o processing in the xen virtual machine monitor," in *Usenix ATC*, 2005.

[45] C. A. Waldspurger, "Memory resource management in vmware esx server," in *Proc. Usenix OSDI*, 2002.

[46] A. Verma, G. Kumar, R. Koller, and A. Sen, "Cosmig: Modeling the impact of reconfiguration in a cloud." in *IEEE MASCOTS*, 2011.

[47] B. Sharma, P. Jayachandran, A. Verma, and C. Das, "Cloudpd: Problem determination and diagnosis in. shared dynamic clouds," in *Proc. DSN*, 2013.

[48] A. Verma, P. Ahuja, and A. Neogi, "pmapper: Power and migration cost aware application placement in virtualized systems," in *Proc. Middleware*, 2008.

[49] R. Koller, A. Verma, and A. Neogi, "Wattapp: An application aware power meter for shared data centers," in *ICAC*, 2010.

[50] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, 2013, pp. 379–391. [Online]. Available: http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Zhang_2.pdf

[51] N. Rameshan, L. Navarro, E. Monte, and V. Vlassov, "Stay-away, protecting sensitive applications from performance interference," in *Proceedings of the 15th International Middleware Conference*, ser. Middleware '14. New York, NY, USA: ACM, 2014, pp. 301–312. [Online]. Available: http://doi.acm.org/10.1145/2663165.2663327

[52] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687

[53] HAProxy, "The Reliable, High Performance TCP/HTTP Load Balancer," http://www.haproxy.org/.

[54] W. Zhang *et al.*, "Linux virtual server for scalable network services," in *Ottawa Linux Symposium*, vol. 2000, 2000.

[55] R. P. Mahowald and M. Rounds, "It buyer market guide: Cloud services," in *IDCReport*, July, 2013.

[56] K. T. (PCWorld), "Thanks, Amazon: The Cloud Crash Reveals Your Importance," 2011, http://www.pcworld.com/article/226033/thanks_amazon_for_making_possible_much_of_the_internet.html.

[57] R. J. F. Inc.), "More details on today's outage," 2011, https://www.facebook.com/note.php?note_id=431441338919&id=9445547199.

[58] Oracle, "Jdk-6558100 : Cms crash following parallel work queue overflow," 2011, http://bugs.sun.com/view_bug.do?bug_id=6558100.

[59] Olio, "The Workload," http://incubator.apache.org/olio/the-workload.html.

[60] Basic Linear Algebra Subprograms. http://www.netlib.org/blas.

[61] OProfile, "OProfile," http://oprofile.sourceforge.net/about/.

[62] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, "IC2: Interference-aware Application Configuration in Clouds," May 2014, technical Report, School of Electrical and Computer Engineering, Purdue University, http://docs.lib.purdue.edu/ecetr/.

[63] C.-Z. Xu, J. Rao, and X. Bu, "Url: A unified reinforcement learning approach for autonomic cloud management," *Journal on Parallel and Distributed Computing (JPDC)*, vol. 72, no. 2, pp. 95–105, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2011.10.003

[64] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1656274.1656278

[65] A. K. Maji, "Httpd with Online Reconfiguration," 2014, https://github.com/amaji/httpd-online-2.4.3.git.

[66] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 159–172. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043572

[67] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, "Mitigating interference in cloud services by middleware reconfiguration," in *Proceedings of the 15th International Middleware Conference*, ser. Middleware '14. New York, NY, USA: ACM, 2014, pp. 277–288. [Online]. Available: http://doi.acm.org/10.1145/2663165.2663330

[68] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408794

[69] PAPI, "PAPI on Virtualization Platforms," http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:PAPI_on_Virtualization_Platforms.

[70] MacOSforge, "Darwin Streaming Server," http://dss.macosforge.org/.

[71] EPFL, "CloudSuite Media Streaming Benchmark," http://parsa.epfl.ch/cloudsuite/streaming.html.

[72] A. Rao, A. Legout, Y.-s. Lim, D. Towsley, C. Barakat, and W. Dabbous, "Network characteristics of video streaming traffic," in *CoNEXT*, 2011.

[73] A. Finamore, M. Mellia, M. M. Munafò, R. Torres, and S. G. Rao, "Youtube everywhere: Impact of device and infrastructure synergies on user experience," in *IMC*, 2011.

[74] T. Stockhammer, "Dynamic adaptive streaming over http –: Standards and design principles," in *MMSys*, 2011.

[75] R. K. P. Mok, X. Luo, E. W. W. Chan, and R. K. C. Chang, "Qdash: A qoe-aware dash system," in *MMSys*, 2012.

[76] M. Sullivan and R. Chillarege, "Software defects and their impact on system availability-a study of field failures in operating systems," in *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, 1991, pp. 2–9.

[77] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "Bluegene/l failure analysis and prediction models," in *Proceedings of the International Conference on Dependable Systems and Networks*, ser. DSN '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 425–434. [Online]. Available: http://dx.doi.org/10.1109/DSN.2006.18

[78] S. Chandra and P. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 2000, pp. 97–106.

[79] B. Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Verlag John Wiley & Sons, Inc, 1995.

[80] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, pp. 32 – 44, December 1990.

[81] P. Koopman and J. DeVale, "The exception handling effectiveness of posix operating systems," *Software Engineering, IEEE Transactions on*, vol. 26, no. 9, pp. 837 – 848, sep 2000.

[82] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[83] B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of unix utilities and services," University of Wisconsin-Madison, Tech. Rep., 1995.

[84] J. E. Forrester and B. P. Miller, "An empirical study of the robustness of windows nt applications using random testing," in *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*. Berkeley, CA, USA: USENIX Association, 2000.

[85] B. P. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of macos applications using random testing," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 78 – 86, January 2007.

[86] J. DeMott, "The evolving art of fuzzing," http://www.vdalabs.com/tools/, June 2006.

[87] P. Godefroid, "Random testing for security: blackbox vs. whitebox fuzzing," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, ser. RT '07. New York, NY, USA: ACM, 2007.

[88] P. Oehlert, "Violating assumptions with fuzzing," *Security Privacy, IEEE*, vol. 3, no. 2, pp. 58 – 62, march-april 2005.

[89] J. Neystadt, "Automated penetration testing with white-box fuzzing," http://msdn.microsoft.com/en-us/library/cc162782.aspx, February 2008.

[90] A. Johansson, N. Suri, and B. Murphy, "On the selection of error model(s) for os robustness evaluation," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, june 2007, pp. 502 –511.

[91] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011.

[92] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 235 – 245.

[93] Y.-Y. Su, M. Attariyan, and J. Flinn, "Autobash: improving configuration management with operating system causality analysis," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 237–250. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294284

[94] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: Finding the needle in the haystack," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, vol. 6, 2004, pp. 1–14.

[95] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924960

[96] L. Keller, P. Upadhyaya, and G. Candea, "Conferr: A tool for assessing resilience to human configuration errors," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, June, pp. 157–166.

[97] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 17–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251271

[98] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. Parekh, "Online response time optimization of apache web server," in *Proceedings of the 11th international conference on Quality of service*, ser. IWQoS'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 461–478. [Online]. Available: http://dl.acm.org/citation.cfm?id=1784037.1784071

[99] W. Zheng, R. Bianchini, and T. D. Nguyen, "Automatic configuration of internet services," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 219–229. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273020

[100] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 287–296.

[101] L. Wang, J. Xu, and M. Zhao, "Application-aware cross-layer virtual machine resource management," in *Proceedings of the 9th international conference on Autonomic computing*, ser. ICAC '12. New York, NY, USA: ACM, 2012, pp. 13–22. [Online]. Available: http://doi.acm.org/10.1145/2371536.2371541

[102] I.-H. Chung and J. K. Hollingsworth, "Automated cluster-based web service performance tuning," in *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*. IEEE, 2004, pp. 36–44.

[103] X. Bu, J. Rao, and C.-Z. Xu, "Coordinated self-configuration of virtual machines and appliances using a model-free learning approach," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 4, pp. 681–690, 2013.

[104] A. Gulati, A. Merchant, and P. J. Varman, "mclock: handling throughput variability for hypervisor io scheduling," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–7.

[105] A. Shieh, S. Kandula, A. Greenberg, and C. Kim, "Seawall: performance isolation for cloud datacenter networks," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud)*. USENIX Association, 2010, pp. 1–6.

[106] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in xen," in *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*. Springer-Verlag New York, Inc., 2006, pp. 342–362.

[107] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha, "Sharing the data center network." in *NSDI*, 2011.

VITA

VITA

Amiya Maji earned his B.Tech (Bachelor of Technology) and M.S. (Master of Science) degrees from the Department of Computer Science and Engineering at Indian Institute of Technology, Kharagpur, India in the years 2005 and 2008 respectively. He has been a Ph.D. student in the School of Electrical and Computer Engineering at Purdue University since then. Amiya is currently working towards his doctoral dissertation in the Dependable Computing Systems Laboratory under the supervision of Prof. Saurabh Bagchi. His research focuses on dependability aspects of large-scale distributed system. More specifically he has published several peer-reviewed papers on reliability and security of mobile devices; reliability and performance of cloud services; web application reliability; and security of Pub-Sub networks in top conferences such as ISSRE, DSN, ICAC, Middleware, and SecureComm. Prior to joining Purdue, Amiya did several projects on Telemedicine during his bachelor's and master's degrees. Amiya has also served as secondary reviewer in top reliability conferences such as DSN, SRDS and PRDC.

PUBLICATIONS

ISSRE2010 A. K. Maji, K. Hao, S. Sultana, S. Bagchi. "Characterizing Failures in Mobile OSes: A Case Study with Android and Symbian," in *21st International Symposium on Software Reliability Engineering*, ISSRE 2010, November 1-4, 2010, San Jose, California.

DSN2012 A. K. Maji, F. A. Arshad, S. Bagchi, J. S. Rellermeyer. "An Empirical Study of the Robustness of Inter-component Communication in Android," in *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN 2012, June 25-28, 2012, Boston, MA.

MW2014 A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, A. Verma. "Mitigating Interference in Cloud Services by Middleware Reconfiguration," in *15th International Middleware Conference*, MIDDLEWARE 2014, December 8-12, 2014, Bordeaux, France.

ICAC2015 A. K. Maji, S. Mitra, S. Bagchi. "ICE: An Integrated Configuration Engine for Interference Mitigation in Cloud Services," in *12th International Conference on Autonomic Computing*, ICAC 2015, July 7-10, 2015, Grenoble, France. (*Under review*)

DSN2009s A. K. Maji, K. Hao, S. Sultana, S. Bagchi. "Characterization of Failures in Android Operating System," in *39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN 2009 (Fast Abstract), June 29-July 2, 2009, Lisbon, Portugal.

OTHER PUBLICATIONS

ICAC2014 Fahad A. Arshad, Amiya K. Maji, Sidharth Mudgal, Saurabh Bagchi. "Is Your Web Server Suffering from Undue Stress due to Duplicate Requests?" in *11th International Conference on Autonomic Computing*, ICAC 2014, June 18-20, 2014, Philadelphia, PA (short paper).

SECCOM11 Amiya K. Maji, Saurabh Bagchi. "v-CAPS: A Confidentiality and Anonymity Preserving Routing Protocol for Content-Based Publish-Subscribe Networks," in *7th International ICST Conference on Security and Privacy in Communication Networks*, SecureComm 2011, September 79, 2011, London, UK.