Title
Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software

Author
Rabkin, Ariel Shemaiah

Publication Date
2012

Peer reviewed|Thesis/dissertation

Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software

by

Ariel Shemaiah Rabkin

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Randy Katz, Chair
Professor Koushik Sen
Professor Ray Larson

Spring 2012

Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software

by

Ariel Shemaiah Rabkin

Abstract

Using Program Analysis to Reduce Misconfiguration in Open Source Systems Software

by

Ariel Shemaiah Rabkin

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy Katz, Chair

This dissertation describes using program analysis to document and debug modern open source systems software. We address three problems: documenting configuration, debugging configuration errors, and improving logger configuration.

Thanks to the Cloud, developers today often have access to dozens or hundreds of nodes. Managing this hardware requires a large software stack. Often, this software is open-source and community developed. As a result of this development model and the complexities of distributed resource management, modern software systems can have dozens or even hundreds of configurable options. The open source development process makes it easy for developers to add options and easy for documentation to become stale. We offer a static analysis to identify the options present in a given program and to infer types for them. Our analysis is often more precise than the existing human-written documentation.

We offer a similar analysis to aid in debugging configuration errors. We build an explicit table matching program points to the configuration options that might have caused a failure at that point. The analysis runs quickly, taking less than an hour for programs with hundreds of thousands of lines of code. We use Hadoop and JChord as case studies to assess accuracy. For those programs, our technique diagnoses over 80% of the errors due to randomly-injected illegal configuration values. Precision is high; our analysis finds an average of 3-4 possibly relevant options for each error. Using stack traces in the analysis removes approximately a third of the imprecision as compared to error messages alone.

We also present a solution to a quite different problem: poor quality console logs. Log analysis and logging configuration are hampered by the fact that there is no way to refer unambiguously to a particular log statement. Assigning a unique identifier to every statement enables fine-grained control of which messages are printed and how they are labeled. We achieve this using program-rewriting, retrofitting statement numbers to legacy Java programs. This numbering is consistent across program runs and in the presence of software updates. We use an offline analysis to match statements across program versions. The runtime overhead of our approach is negligible.

This dissertation is dedicated to my grandmother, Lillian Rabkin (1919 - 2009). She was sure I would eventually grow up to be a scientist. Somehow, I have.

Contents

List of Figures

List of Tables

Acknowledgements

This dissertation would not have happened without the aid, support, and encouragement of a great many people along the way.

Ken Birman and Gün Sirer convinced me to go to graduate school. Both of them taught inspiring and exciting systems courses. On separate occasions, both of them pulled me aside and urged me to apply to graduate schools: Ken assured me it would be fun; Gün warned me I would need to work hard. Both, of course, were correct.

I bounced between several topics before embarking on the research that led to this dissertation. I am profoundly grateful to Randy Katz, my advisor, for giving me the freedom to explore and trusting me to come out at the other end with a substantive piece of computer science research.

Randy has taught me a great deal, over the years, about how to do science. Re-reading old papers, I am struck how very much his advice has improved my scientific writing.

Neither Randy nor I had ever worked on program analysis or software engineering before. Mayur Naik, George Necula, and Koushik Sen all supplied advice and orientation in navigating the world of software engineering in general, and applied static analysis in particular. All three were generous with their time and patient with my uncertainties and ignorance.

I am especially grateful to Mayur, whose JChord analysis toolkit was the foundation for my work. Throughout, he was understanding and accommodating to my (sometimes misguided) efforts to add the features I wanted to JChord.

This work was done under the auspices first of the UC Berkeley RADLab and then the AMP Lab projects. I appreciate the support given me by the State of California and our industry sponsors, including Sun Microsystems, Google, Microsoft, Amazon, Cisco, Cloudera, eBay, Facebook, Fujitsu, HP, Intel, NetApp, SAP, VMware, and Yahoo!.

I spent the summer of 2011 at Cloudera. I am grateful to the management and staff there for making my internship both pleasant and productive. I particularly thank Amr Awadallah, Omer Trajman, and Angus Klein, my managers while I was there, as well as Vikram Oberoi and Adam Warrington, my colleagues, who were understanding with my efforts to reconcile corporate priorities with scientific interest. I am particularly grateful for their authorization to publish the failure data that appears in Chapter 2.

The lab staff, Kattt, Sean and Jon, managed to keep everything running smoothly despite the chaos, confusion, and special needs that are inherent in major research projects. I am grateful to them for keeping the train on the tracks.

Many fellow students offered their company and support during my time at Berkeley. Michael Armbrust was always there to make sure things happened and to reassure me that he too was going through the same challenges I was. Steve D-H was a reliable source of good cheer and good sense. Jacob Burnim and Derrick Coetzee gave me much valuable advice about writing for a software engineering audience. Terry Filiba kept me sane during the lead-up to my qualifying exam.

Last, I appreciate the love and support that Sophie Litschwartz showed throughout.

Curriculum Vitæ

Ariel Shemaiah Rabkin

Education

May 2006	Cornell University A.B., Computer Science (*Summa cum laude*)
May 2007	Cornell University M.Eng., Computer Science
December 2009	University of California, Berkeley M.S., Electrical Engineering and Computer Science
June 2012	University of California, Berkeley Ph.D., Electrical Engineering and Computer Science

Biographical Sketch

Ariel Rabkin was born on April 3, 1984, in Ithaca, NY. With brief interruptions, he spent the next 23 years there. Around the time he started high school, he discovered that it was possible to make the computer do things it wasn't previously able to do. He became fascinated with programming, and then with the deeper science behind programming. This has taken him, by various stages, to UC Berkeley, where his dissertation work will, hopefully, help other programmers make *their* machines more docile.

Chapter 1

Introduction

1.1 The Big Picture

It is instructive to compare the user experience involved in setting up a popular web browser, such as Firefox, with that of setting up a popular web server, such as Apache[1]. The browser is usable immediately when installed. It has a graphical interface with a set of dialog boxes for configuration. The configuration interface shows users the scope of possible options. Invalid options can be flagged as soon as they are specified. The server, in contrast, is controlled by a textual configuration file, modified by a general-purpose text editor. No feedback is available from the editor for incorrect or even nonsensical options. Particularly for users with little experience configuring the program in question, this process can become a maddening ordeal of trial-and-error.

This is an instance of a general pattern. Consumer software applications are largely designed to work "out of the box." While word processors and web-browsers have many customizable preference options, designers usually strive to make the defaults suitable most users. Sometimes, no sensible default exists. For instance, an email client requires each user to enter their own email address. In these cases, consumer software will often pop up a dialog box, requiring the user to enter this information before using the program. Substantial care is often devoted to making configuration quick and friendly, even for novice users. Usability testing is an important part of the consumer software development process, especially for commercial software but also in open source development.

In contrast, systems software is often far more difficult to configure. Network services, distributed data-storge systems, and batch processing frameworks are designed to provide service to

[1] There are third-party graphical interfaces for configuring Apache, but here we are only concerned with the features of the service itself.

other programs, not directly to users. Performance, flexibility, and robustness, rather than ease of use, are often developers' paramount concerns.

Offering flexibility requires giving choices to users. We refer to these choices as "configuration options." In systems software, configuration typically determines such aspects of execution as where in the local filesystem to store data, which ports a server should bind to, and even which algorithms should be used in various parts of a large system.

The term "configuration" covers many different aspects of a program's environment, chosen by many actors, including configuration files, installed software, and even the underlying hardware. This dissertation describes techniques for easing two specific forms of configuration: named configuration options and logger configuration.

Named configuration is a common and particularly simple means for developers to offer users a configuration option. In this idiom, the developer gives each option a name and perhaps a default value. Users can then explicitly set the value associated with that name, perhaps via a textual configuration file, on the command line, or else by explicit `setConfiguration` API calls (in the case where "users" are really developers of additional software that runs alongside an existing system). Older readers may remember manually editing `.INI` files on Windows or DOS. Readers familiar with Unix should picture system environment variables. When we refer to "configuration" in this dissertation without clarification, we are thinking of this named configuration model.

On Unix, system environment variables can have arbitrary strings as names and these variables are kept in a flat data structure. Other design choices are possible. The Windows Registry stores options hierarchically in a tree structure. Still other systems use XML formats for configuration data, where the path to the value represents its name. In all these cases, the essential aspect for our purposes is that a configuration option can be modeled as a key-value pair.

Many programs produce semi-structured textual logs ("console logs") for debugging and administration purposes. In simple programs, the developer may simply insert `print` statements into the source code. In more substantial programs, this approach is painfully inflexible. Developers often want more-complete logging than do administrators once a system is in routine operation. As a result, developers have created a plethora of logging libraries that allow users to dynamically configure which messages are printed to the log. We use the term *logger configuration* to refer to the built-in mechanisms for altering the behavior of these libraries. Typically loggers do not use a key-value configuration model. The usual pattern, rather, is for the programmer to group the log statements in the program into a hierarchy; the library allows separate configuration at each level of the hierarchy, with lower (more fine-grained) levels overriding higher (coarser) levels.

An important motivating example, which we will return to throughout this dissertation, is Apache Hadoop. Hadoop is a prominent open source project with a large and professional development base.[2] Despite its prominence and regard, Hadoop can be fearsomely hard to configure. Current versions have over 400 named options. As we show in Chapter 3, misconfiguration is the largest class of problems referred to supporters at Cloudera, a leading Hadoop support vendor.

This dissertation describes a a set of tools and techniques to make configuration (and configu-

[2]As evidence for its prominence and maturity, we note that the Hadoop development community received the 2011 MediaGuardian Innovation Award award, for "innovator of the year"; the project also won the 2012 InfoWorld "Technology of the Year" award.

ration debugging) simpler and less error-prone. Our focus is on large processing frameworks, like Hadoop. Our goal is to develop techniques that are practical, given today's software systems and today's models for software development and support.

Hadoop, including its affiliated projects such as HBase and Hive, is among the most widely deployed "big data" systems. Techniques that worked for Hadoop and its ecosystem would be of value even if applied to no other systems. Happily, we will demonstrate that our techniques do indeed succeed on other, unrelated, systems.

Our work is timely for two reasons. First, the rise of open source development and cloud computing has encouraged developers to build complex distributed processing frameworks. Configuration is a major challenge for these frameworks. Second, trends in software development and automated program analysis have given new tools to the computer science community. The purpose of this dissertation is to apply these new tools to this increasingly critical problem.

The next two sections of this introduction will describe these two trends in more detail. Following that, we discuss our methodology and goals. The last section of the introduction will combine this technical background with our methodology to present the detailed contributions and structure of this dissertation.

1.2 The Rise of Open-Source and the Cloud

In the last 20 years, open-source software development has become common. In this development model, code is open, and any volunteers who wish to contribute can propose changes. Thanks to the Internet, it is increasingly feasible for widely distributed groups of developers to collaborate on development in this manner.

Many open source projects are run by consensus, without centralized management. Such projects can accumulate configuration options that were useful for solving some particular problem at a particular site at some point in time. Without centralized management, documentation can be sparse, out of date, or simply wrong [48]. As a result, the set of configuration options in open source systems projects can be large and incompletely documented [55, 62].

The problem is especially severe for distributed systems, which involve many coordinated machines. Coordinating the activity of many machines requires configuring each machine to know which other machines it is interacting with, to know how to communicate with them, and how to recognize the failure of a remote host. Each of these aspects of distributed computing requires some configuration. While in theory developers could pick fixed settings for each aspect, it is far more common to give this choice to users; to make them configurable.

Distributed systems, with their configuration challenges, are set to become ever more important in the coming years. Twenty years ago, such systems were rare and processor cycles were expensive. Distributed computing was a specialized problem, primarily for developers of networked services or supercomputer software. Today, distributed services are ubiquitous. The Internet, of course, is essentially distributed: users and websites are connected by a network. Behind the user-

visible web interfaces, there are typically cluster computing systems for large-scale data storage and processing.

Increasingly often, Internet services rely on *cloud computing* to supply these computing clusters.. Cloud computing is defined as a platform that allows individuals and organizations to rent computational resources as needed, spread across many nodes [8]. The availability of cloud computing changes the economics of software development. If a program will be run on a large cluster, the cost of administration can be amortized across the cluster, rather than being proportional to its size. As a result, it is cost-effective to add configuration options that allow a highly paid expert to eke out a few-percent gain in performance.

This tuning is unlikely to be error free. Past studies have shown that human administrators almost inevitably make mistakes, even when given explicit step-by-step directions [14]. Particularly given an unfamiliar program, errors and fumbling are inevitable. For widely-used programs, web search can bring up user reports of problems and solutions. But configuration debugging is still a thorny problem for specialized open-source programs. While proprietary software typically comes with manufacturer support, users of open-source systems must rely on the project's developers or volunteers. Human support may not be readily available. Developer time is a limited resource, and inspecting and diagnosing user mistakes can be a low priority.

Because volunteer support is so hit-and-miss, open source projects often are associated with of private companies that sell support. The existence of these companies is evidence that configuring and using open-source software is a problem severe enough that users will pay significant sums to solve.

1.3 Program Analysis is Now Practical

Program analysis is a catch-all term for techniques that verify properties of computer programs. Sometimes, this can be done without running the program; this set of approaches are called *static analysis*. Other techniques rely on observing the execution of the program, possibly after modifying it to add instrumentation. This is called *dynamic analysis*. In the last twenty years, particularly the last ten years, both static and dynamic analysis methods have gone from being a research curiosity to a practical tool.

Part of the reason for the upsurge in program analysis is that modern software is increasingly easy to analyze. Twenty years ago, systems software was primarily written in C or similar languages. These languages are a challenging target for analysis due to their weak type systems. To give a concrete example, there is no way to know, either statically or dynamically, just what sort of object a pointer is pointing to.

Ten years ago, the developers of SEDA had to justify their choice of Java as an implementation language [81]: the overheads of using a managed language like Java were viewed as a serious drawback. Today, the choice to use Java would be unremarkable. Java implementations have gotten more efficient, reducing the performance cost of using these languages. The machines themselves are now much larger and more capable, amortizing the overheads from garbage collection and

interpretation in managed languages. As a result, Java and C# have become the standard languages for systems programming, being used by Hadoop, Cassandra, Pastry, and Dryad. Unlike C, these languages have strong type systems. If a program asserts that a reference points to something of type `Array`, analysis can rely on this typing property, since the language implementation will enforce it.

As mentioned above, program analysis can be either static or dynamic. Of the three technical chapters in this dissertation, two are primarily about static analysis or primarily-static analysis. Only one is primarily about dynamic program transformation. For our purposes, there are two key advantage of static analysis. First, it requires no changes to the program being analyzed or to its run-time environment. This reduces the cost of deployment. There is no need for administrators to modify an existing system or to incur down-time. Related to this, static analysis cannot have side effects. Since the system of interest is not being run by the analysis, there will be no spurious network connections that would confuse a remote host, nor can the analysis process result in unpredictable changes to the filesystem. In contrast, both of these side-effects are possible with dynamic analyses if care is not taken: dynamic analysis requires modifying either the program being analyzed or underlying layers of the software stack.

Program analysis techniques can also be classified based on whether they need to analyze the whole program, or whether they can work by analyzing a piece at a time. Whole-program analyses of large software systems were vanishingly rare a decade ago. RacerX [25] was able to scale to full operating system kernels only by using a very simple and approximate model of program memory, thereby reducing the precision of the results.

Today, however, complex whole-program analyses are feasible. Over the last ten years, programming-langauge researchers have developed increasingly effective algorithms for *points-to* analysis. Points-to analysis builds a model of the data structures in a program's memory, allowing further analysis of the program's behavior. This map of program memory is an essential building block for many analyses of program behavior, including those presented in this dissertation. These algorithmic improvements are therefore another key enabler for this dissertation. (Chapter 3.2.2 has more detail about the state-of-the-art in points-to analysis.)

The last enabling technology for our work is that sophisticated program analysis techniques are now embodied in reusable frameworks. The static analysis community has, within the last five years, developed a number of flexible frameworks for carrying out analyses. These tools carry out many of the routine portions of an analysis, such as parsing Java bytecode into an intermediate representation. They also include implementations for standard analysis tasks, such as points-to analysis.

Three examples of such frameworks are WALA, developed by researchers at IBM [5], Doop, developed at the University of Massachusetts [13], and JChord, developed at Stanford and Intel Research Berkeley [57]. We opted to use JChord in our work. Doop and JChord both provide a Datalog environment in which analyses can be written in a compact declarative form. However, Doop is built around the proprietary LogicBlox Datalog solver, while JChord uses the freely available open-source `bddbddb` solver.

1.4 Philosophy and Methodology

Our goal is to demonstrate practical techniques for system management based on program analysis. This means that the techniques we develop must scale to large software systems and must be deployable.

Scale does not simply refer to the number of lines of code in a program, but also to the code's complexity. Systems software sometimes heavy use of reflection and native libraries, both of which are challenges for static analysis. Reflection allows a program to dynamically determine which code will be loaded and invoked, making it harder to statically model execution. Native code is written in a different language from the rest of the program, limiting the coverage of the analysis. Distributed systems pose their own challenges: data will flow between processes via the network in ways that analysis does not model directly. We seek analysis algorithms must handle all these features without a catastrophic loss of accuracy or exorbitant run-time penalty.

Deployability has both developer-facing and user-facing aspects. To be deployable to developers, our techniques cannot require a major commitment. Run-time on ordinary developer-quality laptops should be minutes or tens of minutes, not hours or days. This is short compared to the existing release or testing processes, which are already over an hour for Hadoop.

From a user's point of view, deployability means that the techniques we develop must not require a major commitment or unusual infrastructure in a production environment. As we discuss in Chapter 3, prior work on configuration debugging often assumes the presence of a customized kernel or runtime. This is a severe barrier in practice and one we avoid. Our technique requires no modification to the code being analyzed or its runtime environment. Asa result, experimenting with our technique does not require either developers or users to take risks or to make an expensive up-front investment of time.

Theoretical soundness and completeness guarantees are not priorities for us. Real software generally runs as part of a complex, buggy, and imperfectly-documented system environment. Analysis algorithms that come with a proof of correctness may not, in fact, retain their soundness or completeness when the software to be analyzed is embedded in a realistic system environment.

These goals shape our methodology. Without formal proofs, we rely on experiment to demonstrate that our techniques are sufficiently effective. To make our empirical claims convincing, we analyze a substantial and diverse body of program code. We show that our analysis works on big-data systems like Hadoop, HBase, and Cassandra. We also analyze single-machine software like the JChord program analysis tool, the Apache Derby database, and the `ant` build system. We use Hadoop and JChord as particular case studies. These two systems are sufficiently large and complex to be good tests. The author of this dissertation has contributed to both of these systems and is sufficiently familiar with them to understand how the analysis interacts with the program code.

We will describe real-world success with our techniques; this is a sort of "end-to-end check" that our approach works outside of a research context. The results of our analysis for Hadoop have been used at Cloudera, a Hadoop services company, to benefit both developers and supporters. Techniques similar to ours are being adopted in the Java Pathfinder community [53].

We are inspired in part by the FindBugs system [34]. FindBugs is a collection of program analyses that look for a variety of common mistakes in Java programs. The analyses often have false positives, meaning that they will sometimes warn the user about a programming pattern that is safe in that particular instance, even while being error-prone in general. FindBugs also has false negatives, meaning it does not catch all possible bugs, even of the types it does sometimes recognize. However, it catches many bugs in practice, and is therefore a widely used tool for improving software quality. Like the FindBugs developers, we consider ease of deployment and use more important than formal guarantees. Like them, we rely on practical experiences with large code-bases to evaluate our work.

1.5 Contributions and Organization

This dissertation has six further chapters. In the next chapter, we give the real-world background that informs our work. We describe the software packages we focus on in the remainder of this dissertation. We also describe the support process at Cloudera, a leading support vendor for Hadoop and related Cloud systems. This exposition serves two purposes. First, it identifies the major problems seen in practice, which motivates the contributions of this dissertation. Second, we explain the context in which our solution is intended to operate.

In Chapter 3, we present related academic work. We summarize the current state of static analysis techniques and past work on analyzing configuration work on program analysis. We give a detailed discussion of previously-proposed techniques for configuration debugging. Our analysis of this past work relies on the discussion of enterprise troubleshooting in Chapter 2. As we explain, past work makes assumptions about deployment that are more suitable for desktop environments than for the specialized open-source systems we emphasize in this dissertation.

These preliminaries pave the way for three chapters about technical contributions. Chapter 4 describes a static analysis for extracting and classifying configuration options. The result of the analysis is a list of program options. As we discuss, these listings are useful for both developers and support personnel. The listings are also annotated with inferred types. These inferred types can be used for "configuration spellcheck": verifying that a user-supplied configuration value falls within the domain that the program expects.

Chapter 5 address the challenge of configuration debugging. Our approach builds on the option-finding analysis presented in the chapter before. Given a program, we show how to build a map from program point to possible error cause. As a result, when a user is faced with an error message, a set of possible explanations can be found by simply consulting the table. This gives prompt guidance for which configuration option to inspect or tinker with. As before, we use static analysis. We refer to this approach as *static precomputation* of error diagnoses.

That chapter also presents two extensions to this basic analysis, exploiting additional information that may be available when a program fails. When programs crash, they often produce a stack trace – a listing of the methods called leading up to the failure. We show how to incorporate those stack traces to improve the precision of our results; the improvement is approximately 30%

in the case of Hadoop. We offer a second improvement: the set of configuration options read by a program before failing can be used to narrow down the set of possible root causes.

Chapter 6 presents a very different application of program analysis techniques. We use incremental program rewriting to improve the quality of generated program logs. As we show in Chapter 2, logging configuration is a significant problem industrially in the Hadoop community. Further, the need to map program points to log statements is one of the limiting steps in the analysis presented in Chapter 5. The techniques presented in Chapter 6 resolve these difficulties.

Our conclusions are summarized in Chapter 7, which also presents some possible avenues for future work. We include documentation on our software artifacts as an appendix.

Our implementation is available from the JChord source code repository, currently `http://code.google.com/p/jchord/`.

Chapter 2

Case Studies in Configuration and Debugging

This chapter explains the real-world context in which our work is situated. As we mentioned in our introduction, we evaluate our techniques by applying them to eight substantial open source projects. The first section of this chapter describes those projects. Our description of each is relatively brief; the purpose of this exposition is to demonstrate that these projects span a range of developer communities, purposes, and coding styles.

The remainder of the chapter delves into more detail describing the Hadoop ecosystem and the problems that arise. We describe experiences at Cloudera, a leading Hadoop support vendor. We go into some depth describing how support is delivered to users. This is the context and motivation for our technical contributions. Hitherto, technical support operations have not been described in detail in the computer science literature, so this discussion may be of independent interest.

2.1 A Tour of Selected Open-Source Projects

We looked for large highly configurable open-source software packages written in Java. We restricted ourselves to Java because our static analysis implementation targets Java. We refer to "packages", not programs, because each of the software systems we mention below has several different entry-points (`main` methods). Effectively, each system consists of several related programs, sharing much of their code.

We assembled a corpus of eight projects. These are summarized in Table 2.1. [1]

[1]Most of the columns in this table are straightforward. However, the size of the developer base requires some discussion. Small open-source projects can be run simply by informal coordination among developers. Larger projects

Four are networked services: Hadoop, HBase, Cassandra, and FreePastry. Another four are non-networked services: Ant, Derby, JChord, and Nachos. These programs represent a range of programmer expertise and style. Some are academic, some industrial. We describe each, below.

Name	Version	Code		Est. # core developers
		(lines *)	(kb compiled)	
Hadoop	0.20.2	167,653	5,440	25
HBase	0.89	104,781	3,149	15
Cassandra	0.5.1	36,823	1,961	10
FreePastry	2.1	175,085	6,073	15
Ant	1.8.1	201,090	3,545	40
Derby	10.6.1	1,136,718	7,903	60
JChord	SVN revision 1440 †	35,761	1,209	5
Nachos	5.0j	10,896	467	1

Table 2.1. Summary of programs studied in this dissertation.
* Including comments and white-space
†: The version of JChord used was an in-progress copy of the source repository, not a finished version.

Hadoop We begin with Hadoop, which we mentioned earlier. The project is well-established, with dozens of active developers and ample documentation, including several books. Users range from large software companies with dedicated administration teams to hobbyists attempting to configure a small cluster at home. The early development was largely managed by Yahoo!, inc. This phase of development followed traditional software engineering practices. The Hadoop development group at Yahoo! included designated project managers and quality assurance engineers[2]. Over time, the developer base grew to incorporate engineers at many other companies, notably Facebook. As a result, management became more diffuse, with decisions being made by discussion and consensus and without a single designated quality assurance organization. In this dissertation, we primarily examine Hadoop version 0.20.2. This was the primary stable version during the production of this dissertation.

require some sort of structure. One common model is to have a large and diffuse group of contributors and a smaller and more active group of committers. Typically there are several times as many contributors as committers; contributors are promoted to committer after a sufficient history of valuable contribution. Both contributors and committers can submit proposed changes to the project source code, but only committers have the authority to approve the change and commit it to the central repository. Most of the projects we discuss below follow this model.

A consequence of this contributor-committer model is that contributors can gradually become more or less active, without any formal process. Hence, it is not easy to define, much less measure, the precise size of the developer base at any one time. We estimated the number of developers in two ways, depending on the project management. For projects with identified committers, we used the number of developers with commit privileges. For those without (FreePastry and Nachos), we used the number of identified contributors.

[2]The author was an intern in that development group during the summer of 2008.

HBase Hadoop has spawned several offshoots, related projects that use some of the core Hadoop library code and that have run-time dependencies on Hadoop MapReduce or the Hadoop Filesystems. These are separate projects, however, with separate project management and distinct (though overlapping) developer communities from Hadoop proper. A particularly popular (and particularly complex) such offshoot is HBase [3], a column store modeled on Google's BigTable [17]. The developer community is scattered across a number of companies, including Facebook, Trend Micro, and Cloudera.

Cassandra Cassandra is a distributed storage service developed at Facebook [46]. The code base is comparatively small, and comparatively clean, as might be expected from a project developed by a small and focused team of professional developers.

FreePastry FreePastry is a peer-to-peer distributed hash table originally developed at Rice [65]. The code has been developed by several successive generations of graduate students. Code quality is therefore comparatively poor; there is substantial dead code and only minimal documentation. Unusually, FreePastry makes heavy use of continuations as a control structure. Since Java does not natively support this structure, FreePastry relies on a custom runtime and network library.

Ant Apache Ant is a replacement for `Make`, originally developed at Sun Microsystems as a component of the Tomcat servlet container [1]. It has become a standard build system for Java programs.

Derby Derby is an open-source database originally developed as part of IBM's Cloudscape project [2]. The developer base draws heavily from large database vendors, including IBM and Oracle, although there is also a substantial developer contingent drawn from small start-up companies.

JChord JChord is a program analysis engine originally developed at Stanford [57]. It is also the analysis engine used for much of the work of this dissertation. As a result, the author is intimately familiar with the code and the community. The development community is small; most of the code was written by a single author and only four other developers are credited with contributions.

Nachos Nachos is a model operating system environment used for undergraduate education at Berkeley [32]. It is the smallest project we examine. Java Nachos is a rewrite of a previously-developed C++ version.

The reader may notice a wide variation in the ratio between size of source code and compiled object code in Table 2.1. This is not a measurement artifact; this ratio can be swayed by various development practices. For example, a large portion of the compiled Cassandra code is RPC bindings automatically generated by the Thrift library. FreePastry copes with asynchrony and failures by using an idiosyncratic continuation-passing style of programming. This style introduces many small anonymous classes, and therefore a large overhead of class file headers and so forth.

Conversely, the Derby code-base is exceptionally well documented; an unusually large proportion of the source code is comments.

2.2 A Case Study of Support Operations

As we mentioned in the introduction, the Hadoop ecoystem is a high-profile part of the cloud or "big data" landscape today. The ecosystem includes not only the core Hadoop filesystem (HDFS) and MapReduce implementation, but a slew of other components, including HBase (a distributed table store), the Zookeeper coordination service, the Pig and Hive scripting languages, and more. All these components are complex software systems in their own right. All have rough edges, including bugs, complex configuration needs, and incomplete documentation. As a result, there is a commercial need for support.

Cloudera, inc, is a company devoted to supplying this support. We use Cloudera as a case study both of real-world misconfigurations and also for the context in which enterprise configuration support is delivered.

Our observations are based on a several month internship at Cloudera. The commercial nature of support restricts the data that can be published here. In particular, the total volume of support work and the actual time to resolve support tickets are both proprietary[3].

2.2.1 Methods and Data

At Cloudera, support is primarily delivered via a web-based trouble ticket system. Our results are based on analyzing approximately 6 months' worth of support cases ("tickets"), from February through August of 2011. We manually labelled 916 tickets from the period with a category or root cause and with the specific system component being worked on. Our analysis relies on both these manual labels plus additional statistics mined from the support ticketing system. We excluded tickets without a recorded resolution.

When a ticket is closed, supporters usually record approximately how long they spent actively working on it. Supporters do not precisely log their time spent and often omit to record it for short tickets. They also do not record time spent by engineers or other supporters who help. We therefore adjust the reported time by adding five minutes for each supporter comment on the ticket. Supporters have confirmed to us that this is a reasonable metric. Moreover, our findings are robust against various changes to the formula.

[3]The danger, in both cases, is not primarily giving technical advantage to competitors. Rather, it is the risk of giving a *marketing* advantage.

2.2.2 Overview

At a coarse level, support interactions with customers can be divided into "diagnosis" and "non-diagnostic". A diagnostic ticket is a customer-reported problem for which a root cause must be found and a solution proposed. A non-diagnostic ticket is one in which the user needs some service or information. This can range from users asking for examples of particular programming practices to seeking advice about hardware, to scheduling on-site training sessions. Non-diagnostic tickets are common, representing approximately 65% of tickets. They tend to be quick to resolve, however, and so account for only a third of supporter time.

Category	Definition	Common examples
Bug	A problem that required a patch to fix a problem in Hadoop or Cloudera source. A bug with a configuration workaround is still a bug.	Deadlock in MapReduce. HBase opening too many Zookeeper connections.
User bug	A problem resolved by a change to user code.	
System problem	A problem caused by a JVM or OS-level defect or misconfiguration not specific to Hadoop. Does not include JVM memory settings or Hadoop-related file permissions.	NFS misconfigured
Hardware problem	A problem due to a hardware defect.	Bad network cable
Operational	The user is operating the system wrongly, perhaps by starting or stopping it in the wrong way.	using the wrong start scripts. Turning off HDFS with HBase still running.
Install problems	A problem caused by a faulty installation	Stale libraries, inconsistent versions
Misconfiguration	Any diagnostic ticket that did not require a fix to code, hardware, or the underlying system software. Excludes cases where users ask for advice about configuration.	Bad classpath in configuration file. OS permissions set wrong on Hadoop storage. Out of disk space.
Clarification	The customer needed explanation or information, only. This includes most cases where problem diagnosis is unnecessary/inapplicable.	"Does version X have bugfix Y?" "What sort of hardware should we buy?"
Administrative	Questions about the support process itself, not about the underlying technology. Account management, access, etc.	"Please create account for user X"

Table 2.2. Definitions of categories of support tickets

Table 2.2 offers a more detailed breakdown of root causes. Administrative and Clarification tickets are non-diagnostic. "Other non-diagnostic" includes several other categories, such as feature requests. The remaining categories required diagnosis.

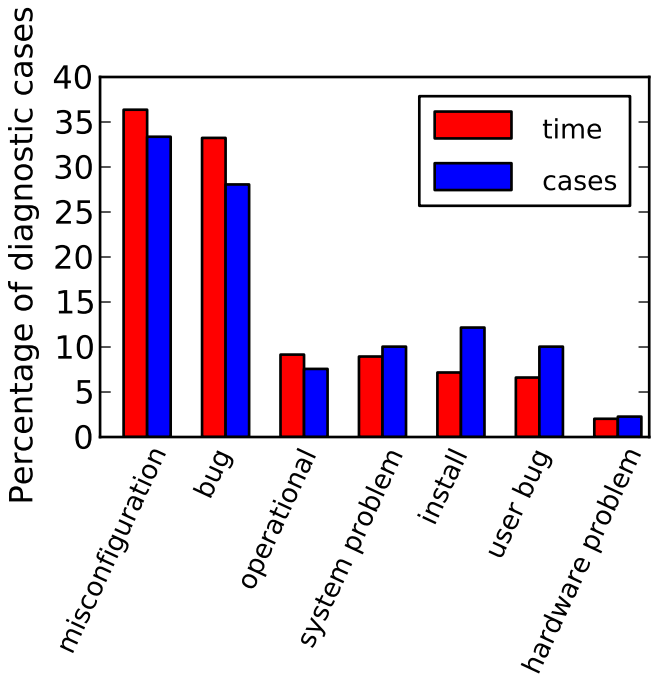

Figure 2.1. Breakdown of support tickets and time by category.

Figure 2.1 breaks down support time and tickets by root cause, excluding non-diagnostic tickets. Figure 2.2 displays average resolution time for each category. As can be seen, bugs take longer to resolve than most other failure causes. However, misconfigurations are more common and account for more total time. Hardware problems are rare, but can take a long time to diagnose.

The number of distinct bugs is smaller than the number of tickets caused by bugs. 30% of bug tickets werefound to be already-known issues slated for fixing in the next version. Over a third of issues without a definitive root cause went away after an upgrade, suggesting that they too were caused by known bugs. This evidence suggests that bugs in a given version tend to manifest quickly and at multiple sites. The measurements for this report include the beta-test period for CDH version 3, so there will have been a higher-than-normal rate of bugs and of upgrades.

In some support contexts, a handful of common issues account for a large fraction of cases.That is not the case for Cloudera. Even the most common specific issues account for no more than 2% or 3% of support cases. This is evidence that the existing process is decently good at learning from past experiences and preventing common issues. In this context, permanently preventing an issue can involve both fixes to the Hadoop platform and extra documentation to explain common problems, letting users can resolve them without additional help. Our data does not let us distinguish the relative importance of these two corrective actions.

The next subsection goes into more detail on misconfiguration problems, since those are the primary focus of this dissertation. Following that, we describe limitations of our data and methods.

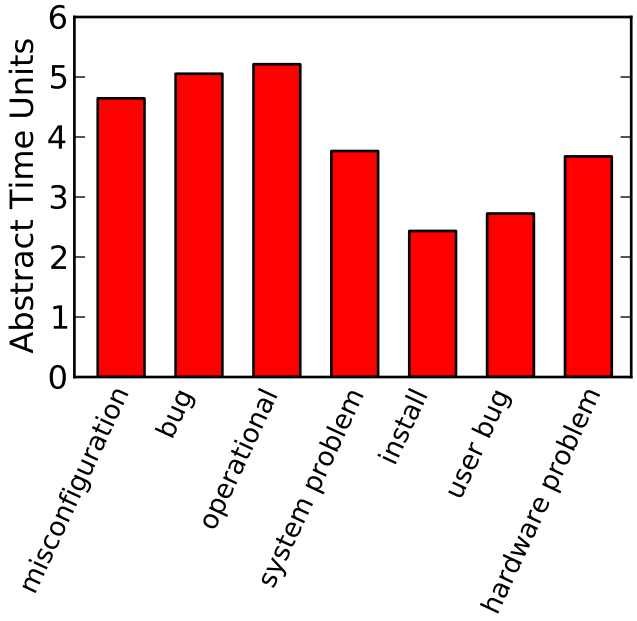

Figure 2.2. Average ticket time by category.

2.2.3 Misconfigurations

We can break down the misconfiguration category into several smaller ones. There are resource-allocation problems, where some system resource, such as memory, file-handles, or disk-space is being mismanaged. This breakdown is shown in Figure 2.5. There are permissions issues.

Approximately a third of all misconfiguration problems were caused by some form of memory mismanagement. Hadoop has many options for controlling memory allocation and usage, at several levels of granularity. Working from the bottom up, there are configurable buffers, such as the MapReduce sort buffer. Each Java process itself has a configured maximum heap size. For each daemon, there is an OS-imposed limit on the maximum amount of RAM to be used, the `ulimit`. For MapReduce, the user can tune how many concurrent tasks to execute on each host. And all tasks must fit into physical memory.

Hadoop does not check that these options form a sensible hierarchy. It is possible for the combined heap size for all the daemons on a machine to exceed physical memory, or for the JVM to request more than the OS-imposed limit. Depending whether the JVM heap size, OS limit, or physical memory is exhausted first, this will cause an out-of-memory error, JVM abort, or severe swapping, respectively.

Another common source of memory-management problems is that a normally-small datastructure becomes unexpectedly large. The Hadoop MapReduce master, the Job Tracker, keeps a summary of past job execution in memory. This history is normally small and so Hadoop does not

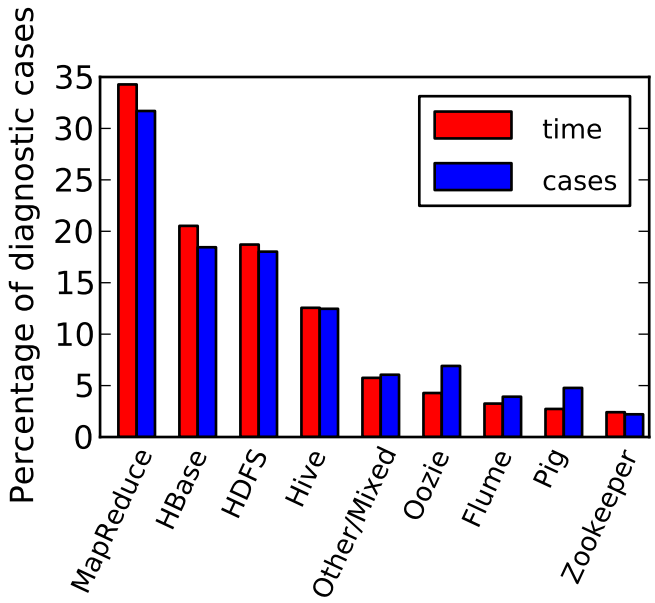

Figure 2.3. Breakdown of support tickets and time by application.

track or manage its size. But if a site runs many jobs with many tasks, it can grow large. This will cause slowdowns due to garbage collection overhead, and ultimately out-of-memory errors.

Memory is not the only aspect of the system prone to misconfiguration. Hadoop services, especially the HBase table store, will keep many files and sockets open. The number of available file handles can be exhausted if it is not increased well above the system default. Disk space, too, can be exhausted if not managed properly.

In addition to managing these resources, Hadoop and HBase rely on users to statically pick maximum sizes for various thread pools, including the sending and receiving sides of the MapReduce shuffle. If there are too many threads requesting data for each server thread, the requesters will experience frequent timeouts, leading to jobs aborting. This sort of issue is marked as "thread allocation" in Figure 2.5.

We noticed two other common sources of trouble. One was permissions and account management, accounting for more than 10% of misconfigurations. Mismatches between the user accounts for MapReduce jobs and those on the local filesystems was a particular source of trouble.

A last problem category we noticed is malformed or or misplaced configuration files. These can be hard to diagnose because if a configuration file is unusable or not found, Hadoop will fall back on the default values for the options in questions. These defaults may be sufficient for some use cases or workloads, but then result in failures under heavy load or when a user tries to use an advanced feature.

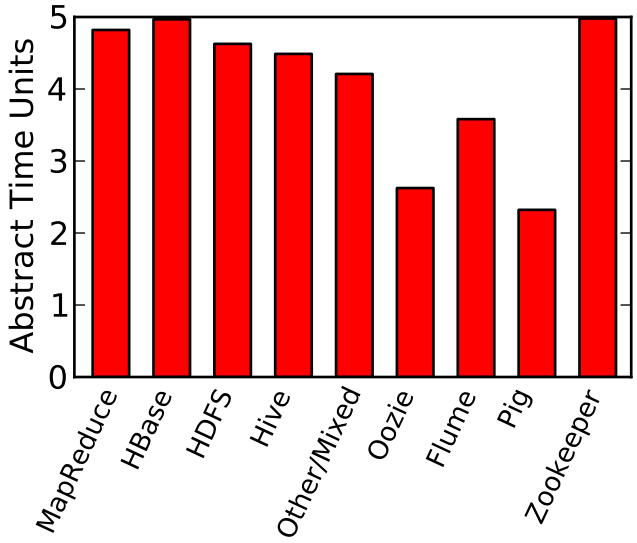

Figure 2.4. Average ticket time by application.

2.2.4 Limitations

The data presented above is based on Cloudera support tickets with a recorded diagnosis. This introduces two kinds of bias: Not all issues result in Cloudera support tickets and not all tickets have a recorded diagnosis. Problems that are clearly caused by hardware or operating system failures will be raised with the appropriate vendors. The easiest-to-solve or least important Hadoop problems may be solved by users without raising an issue with Cloudera support.

Even if an issue is raised with Cloudera, there may not be a definitive diagnosis recorded. Complex problems are often resolved via a phone call, screen-share session, or on-site visit. In these cases, supporters do not always record the final root cause. Sometimes, problems are worked-around until they go away. Upgrades and other reconfigurations can make a problem disappear even when the precise root cause was never established.

As noted, the data collection for this report was during the beta testing for CDH3. As a result, the proportion of issues caused by bugs may be higher than it would have been at other times.

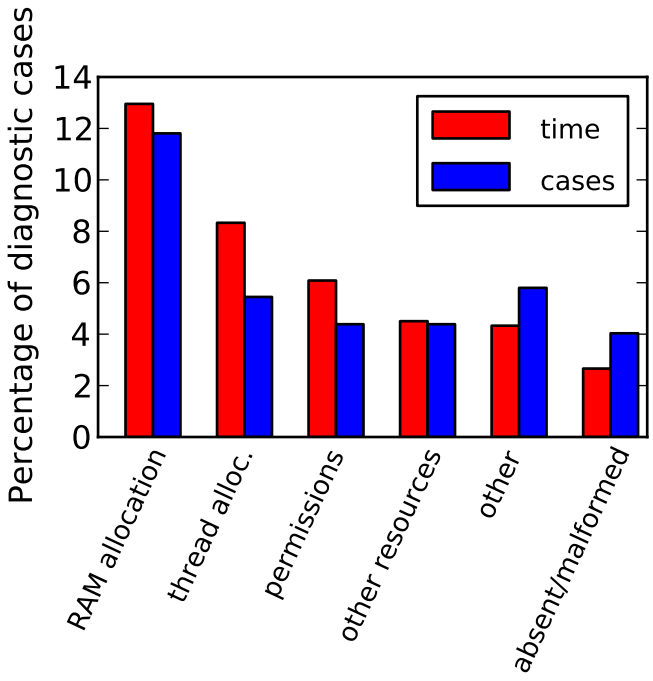

Figure 2.5. Breakdown of misconfigurations.

2.3 Observations

Perhaps the most surprising thing about the problems we observe in customer Hadoop clusters is how un-subtle the root causes often are. The next-most surprising thing is how many different ways a given root cause can manifest itself.

2.3.1 Serious Problems have Unsubtle Causes

As shown above, the two problems that account for the most support time are partial installs and memory misallocation. While these are hard to diagnose, they are in some sense unsubtle. There is a succinct explanation for what went wrong and how to fix it, rather than a long and indirect chain of causation. While it is hard to find out precisely which component has not been installed correctly or *why* memory is exhausted, detecting these problems is straightforward. They can often be reproduced consistently, although it may require long operation periods or heavy load.

Running out of disk space was responsible for eight incidents. This is surprisingly high; routine host health monitoring could have caught this in every instance, and given the cost of the subsequent outages, it is striking that it was not employed.

As noted above, memory mismanagement was the largest single cause of failures. There is

no fundamental reason why memory allocation parameters could not be analyzed for consistency. There are various humdrum engineering reasons, however. For example, processes cannot readily find out what their ulimit is. Some oversubscription of physical RAM may be sensible, if the majority of tasks use less than the maximum allowed memory or if swapping can be tolerated in a particular context.

2.3.2 Multiplicity of Symptoms

The TV character Richard House is famous for observing that "patients always lie." The same could be said of Hadoop clusters. While the problem, when pinned down, usually has a simple explanation, the first overt symptom reported by users is a poor predictor for what that root cause will turn out to be.

The same root cause can often have many divergent manifestations; different causes can have similar visible consequences. We found, for instance, three tickets that were resolved by tuning garbage-collector parameters. In one case, the error manifested as the Job Tracker stalling periodically and being fixed by a restart. In another case, it was occasional slow HBase responses. In the last ticket, the only symptom was excess memory usage by the NameNode, without any functional problems. Each of these symptoms could have had other causes.

This multiplicity of symptoms is unsurprising in this domain. We mentioned above that a large proportion of problems can be characterized as resource exhaustion. Resource exhaustion will manifest differently depending on which point in the code makes the next request after the resource was exhausted and how that condition is handled at that point. Likewise, a resource exhaustion problem can often be fixed in several different ways: if RAM is scarce, this can be addressed by revising user code, by reducing the degree of concurrency to leave more memory for each process, or by reconfiguring services.

This ambiguity can sometimes fool even experienced supporters. One customer site complained about a cluster-wide failure. An experienced supporter and solutions architect commented that "when the whole cluster goes bonkers, it's almost always a client submitted parameter." But in this case, the problem was not a client-submitted parameter. The true root cause was that user log directories were filling up, and exceeded the maximum number of files-per-directory. This resulted in a cascading failure of all the TaskTrackers.

Misconfigurations and bugs can be hard to distinguish. We have seven cases where the customer complains about the "too many fetch failures" message. In five of these cases, the root cause was a simple-to-fix misconfiguration. In the remaining cases, it was a bug in `jetty`, a library used by Hadoop.

2.3.3 Data Collection and Experiments Take Time

Because of the ambiguity of symptoms when systems fail, debugging often requires several rounds of interaction and exploration to find the relevant logs. This imposes a substantial burden.

Sometimes, users supply sufficient log and configuration data to debug the problem immediately, but at other times, it can take substantial time to find the key bits.

Figure 2.6 is a CDF showing when logs and attachments are added to a support ticket. The X-axis is time, scaled relative to the length of the ticket in question. As can be seen, a large fraction of logs are available near the start of the ticket, but both logs and attachments are still trickling in throughout the ticket. If getting logs and attachments was the only barrier to solving a ticket, we would expect a preponderance of arrivals near the finish time. Instead, there is an initial spurt as users describe the problem and supporters ask for logs, and then a roughly constant arrival rate. This shows that much of the work of dealing with a ticket happens after logs are made available.

Cross-layer debugging is a major challenge. The Hadoop ecosystem is a fairly deep stack of components, with query languages like Pig being implemented atop MapReduce, which in turn rely on storage systems like HBase and HDFS. A problem first noticed in Hive might have its true cause anywhere in the stack below that point. There is no simple way to correlate related log entries up and down the stack.

Structured cross-layer tracing tools like X-Trace and Dapper have been proposed to address this problem. There have even been efforts to integrate X-trace with Hadoop[4], but these efforts have foundered in the face of the engineering complexities and lack of community enthusiasm.

Adding to the difficulty is the fact that users are drowned in logs, most of which are irrelevant. On some clusters, nearly 20% of all log messages are warnings, nearly all of which are warning about a nonsensical configuration setting being ignored.

Anecdotally, a major delay in the support process is the need to test out a proposed solution. If testing a fix requires restarting a cluster, this can impose a delay of several days until a convenient opportunity for downtime. If problems are sporadic, this imposes substantial delays while the impact of a fix is assessed.

Permissions problems are comparatively hard to debug, taking longer than the average misconfiguration. It isn't easy to reason about or observe what credentials each piece of code has; services often run code on behalf of other users.

2.4 How Cloudera Copes

This section discussed the techniques used today at Cloudera to deal with enterprise software failures. These form the context for the techniques we discuss in the next three chapters. While we use Cloudera as a case study, we believe that similar techniques are used by other enterprise software supporters.

Cloudera's approach to automated troubleshooting emphasizes tools for supporters, not users. The core of the approach is a system called Clusterstats. Users are asked to install a client program

[4]See: `research.yahoo.com/files/andy_konwinski_x-tracing_hadoop.pdf` and `https://issues.apache.org/jira/browse/HDFS-232`

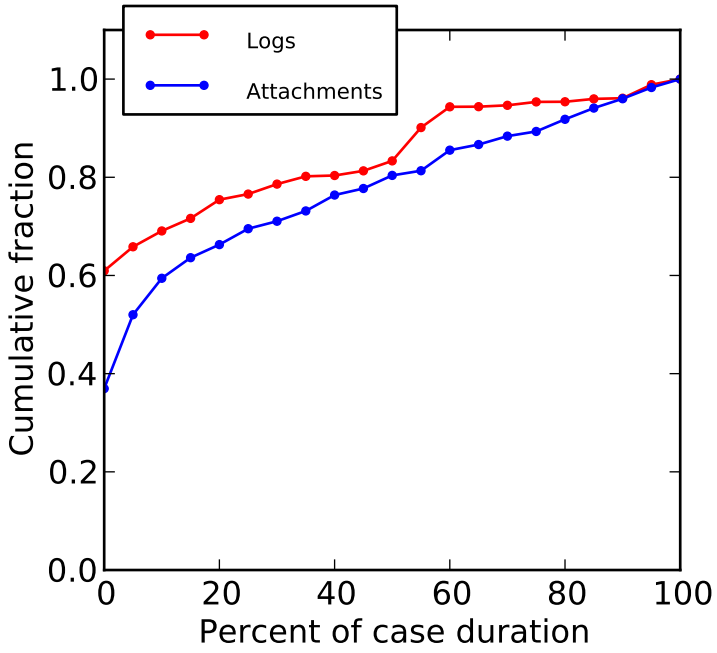

Figure 2.6. Arrival of logs and attachments, scaled to length of each ticket.

that collects logs and records many aspects of hardware and Hadoop configuration. The collected data is sent back to Cloudera, stored, and then made available to supporters via a web interface.

At present, Clusterstats is primarily focused on HDFS, MapReduce, and HBase. Those are the only subsystems from which logs or configuration are collected. (The hardware information, of course, will be applicable in diagnosing problems from other systems as well.) This is simply a reflection of the fact that those services cause the bulk of the configuration debugging time; nothing in the architecture prevents it from growing to encompass other services as well.

The Clusterstats back-end checks the collected data for a range of known problems, such as known-bad configuration settings or low disk space on the NameNode. These checks are done without any reference to the reported symptom. Any suspected problems are highlighted for supporters on the web interface.

All validation and checks for known problems happens on the server side. This allows support engineers to test out a proposed validation on the set of existing collections. It also allows faster innovation and experimentation, since supporters will tolerate more rough edges than users.

The usual practice, after writing a validator, is to run it on the existing data collections as a way of gauging the likely rate of notifications. Often, developers find an example of the problem on some cluster that was not previously known to be faulty.

These previously-unknown configuration issues are not false positives. To continue the medical analogy, the latent issues found by our validators resemble risky lifestyle choices or chronic

conditions that are not currently inflicting harm but may suddenly cause a serious problem. Since validators are aimed at supporters, not end users, they are free to flag problematic configuration, even if those issues are not currently causing user discomfort. Supporters will act on these results. They are encouraged to contact customers with a message like "you have a peculiar value for option X. Even if its not the cause of this problem, we recommend you change it to avoid problems down the line".

Most validators are checking for cases that may result in unpredictable consequences. Having multiple versions of a library or a configuration setting results in unpredictable behavior, depending which one is loaded first at run-time. What is much worse, a supporter who did not know about the multiple versions could easily miss it, and not realize that a valid option or library was being masked by a bad one.

2.5 Observations and conclusions

We now summarize some conclusions. These motivate our work on configuration management and debugging.

Problems rarely repeat in precisely the same way. There are general common patterns, but any one specific problem will account for a small fraction of support time. As a result, root causes and overt symptoms are loosely linked. Often, a bad configuration will lurk for a long time until some otherwise innocuous change to the system or workload causes difficulties. There is no need to wait for things to break before fixing the problem, however: supporters are interested to hear about potential problems that are not causing users current pain. Therefore, symptoms should not be the sole target for problem diagnosis tools: it also makes sense to look for potential problems not currently causing visible trouble. This is the approach we describe in Chapter 4.

Problem diagnosis is the bulk of the support workload. Administrative tasks and answering technical questions from users are only about a third of recorded support time. Hence, diagnosis is the aspect of support most in need of automation. Misconfiguration is the largest category of mistakes, and therefore deserving of particular aid. Chapter 4 follows up on this observation, by presenting a tool for automating misconfiguration diagnosis.

Log analysis is a bottleneck. Often, there is a log message, buried in thousands of lines of irrelevance, that clearly indicates the problem. Picking out the relevant bits in a long log listing is a skill that supporters slowly develop over time. It would be better for systems to clearly separate "actionable" messages pointing to a recommended user action from those that are purely informational or intended for debugging. Chapter 6 will address this problem.

Before we launch into the technical details of our solutions to these problems, however, we review previous academic work and discuss how well it copes with the needs of supporters for enterprise open-source software.

Chapter 3

Related Work

This dissertation is positioned at the intersection between several different communities. On the one hand, applying program analysis to practical problems is a major area of software engineering research. On the other, failure diagnosis is a topic pursued primarily in the systems community. Hence, there is a range of related work topics to cover.

We begin by comparing our failure-cause results from the previous chapter with the results of past studies. Following that, we address the remaining topics roughly in order from most formal to least. We begin by giving an overview of the state-of-the-art in static analysis. While this dissertation breaks no new ground in analysis techniques, the constraints and limitations of today's best algorithms are an important part of the story we tell about the challenges in this work and our contributions. Having laid this ground-work, we then discuss configuration-aware program analysis, which relies on these lower-level techniques. We then turn to debugging technologies, emphasizing techniques that are designed for the particular problems of configuration debugging. This order of topics parallels the structure of the remainder of this dissertation.

3.1 Failures

Several previous publications have discussed real-world failure causes. Jim Gray's "Why do computers stop and what can be done about it" is a prominent example, discussing failure data of Tandem computer deployments [30]. Huang et al. include data about failures seen in production by IBM [35]. Oppenheimer et al. looked at failure data from Internet services in the early 2000s [60]. Vishwanath and Nagappan present real-world failure data, although their focus is on hardware, while ours is on software [77].

Our results in the previous chapter differ from these previous studies. We found that resource and thread allocation were major issues, while they are barely mentioned in past studies. We suspect the reason is intrinsic to big-data systems. Users of these systems are trying to get the

most use possible out of their hardware, meaning they will in general try to operate close to the maximum CPU, memory, and disk utilization. In contrast, Tandem users were optimizing for uptime and were mostly not subjecting their system to continuous intensive load. The price of this more aggressive usage is new failure modes.

Yin et al. have studied configuration errors in five significant applications; they found that mistaken parameter values are 70-85% of all misconfigurations, and that invalid values are 40-50% of these [84]. Brown et al. suggest that typos are omnipresent and are a major problem [14]. These results are helpful for us, since our techniques are aimed at these errors. We note that these studies give a different result from that found in our study at Cloudera, presented previously. In the Cloudera context, invalid value errors are rare, since initial set-up is done on-site by an expert.

3.2 Applied Program Analysis

This dissertation is devoted to applying program analysis, not pioneering new techniques. A full discussion of all past related program analysis work would be out of scope. However, we do give an overview of past work that used similar techniques to ours and of the history of the techniques we use.

3.2.1 Static Analysis and Abstract Interpretation

Most of the analysis we present falls into the general family of abstract interpretation. An abstract interpretation of a program is a many-to-one mapping from program states to abstract states [19]. Typically, the set of abstract states will be finite, as it is in our case. Given a suitable abstract interpretation, static analysis is the process of determining which abstract states a program may or must occupy, without running the program.

Static analysis typically has a trade-off between complexity and precision.Various aspects of execution can either be modeled or else ignored.

- A **flow-sensitive** analysis takes into account the order in which different statements are executed.
- A **path-sensitive** analysis takes into account that not all statements in a given method will be executed. For example, statements on opposite sides of an if-then-else structure cannot both be reached in the same execution of the parent conditional statement.
- A **field-sensitive** analysis distinguishes between structures and their subfields.
- A **context-sensitive** analysis distinguishes different invocations of a different method call. (A *context* is effectively an abstraction over the set of function calls, distinguished perhaps by the calling instruction.)

3.2.2 Points-to analysis

An abstract interpretation of pointers and memory allocation is a *points-to* analysis [33]. (In this dissertation, "reference" and "pointer" are effectively interchangeable; C-based research will usually refer to pointers, and Java-based analyses will discuss references, but the algorithms are the same.) In a points-to analysis, the heap is modeled with one or more *abstract objects*, each of which represents some set of allocated objects. The abstract labels on references describe the set of objects that may be pointed to.

This analysis has several uses. Two references are *aliased* if they point to the same object. If the points-to sets of two references have non-empty intersection, then they may alias. Hence, points-to is effectively equivalent to a so-called *may-alias* analysis.

In object-oriented languages, the target of a method call can depend on the run-time type of the object on which the method is invoked. Hence, constructing an accurate call graph requires knowing the possible types of the objects a reference can point to. Consequently, points-to and call graph construction are often done together. In our work, they will always be combined.

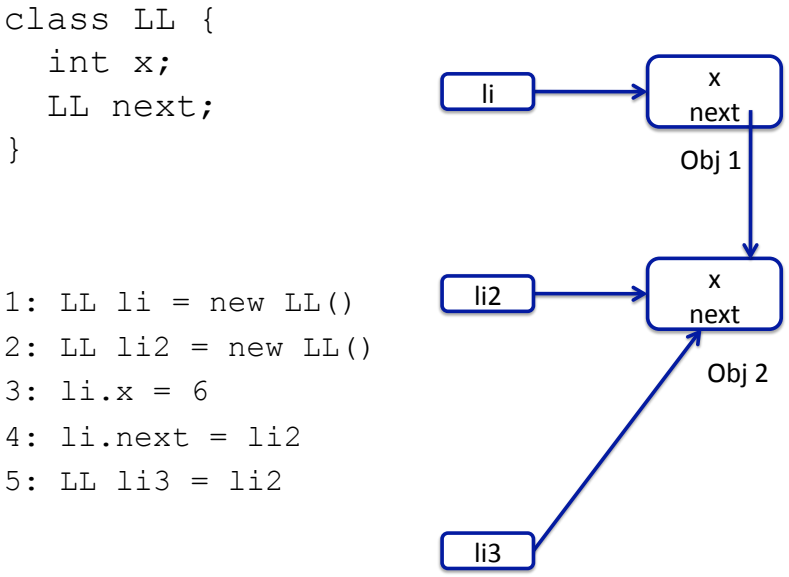

Figure 3.1. Illustration of points-to analysis. Shows a code fragment and associated points-to graph.

The work presented in this dissertation relies on efficient and precise points-to analysis. Much of the technical complexity in Chapter 5 is motivated by the need to cope with limitations of the underlying points-to algorithms. We therefore briefly review the state-of-the-art in points-to analysis algorithms.

The "received wisdom" in the points-to analysis community is that in strongly-typed languages like Java, field-sensitivity is cheap and valuable, that flow-sensitivity is expensive and less valuable, and that context sensitivity is the major variable of interest [33].

Context-sensitive approaches are typically distinguished based on how the contexts are distinguished. The two major approaches used in practice are *call-site sensitivity* and *object-sensitivity*. In the former approach, the abstract context of a method is the call site from which it was reached, and perhaps that parent method's caller. A context is therefore a string of method invocations [72].

Object-sensitive analysis is specific to object-oriented languages. It is the newer technique, and often the more effective [73, 56]. In object-sensitive analysis, a context is the abstract object on which a method is invoked, concatenated with the context in which that object was allocated. Smaragdakis et al. have observed recently [72] that many variants of object-sensitivity are possible. In particular, rather than a chain of allocation sites for abstract objects, some developers use the allocation sites of the caller.

Both object-sensitive and context-sensitive approaches are parameterized by the *context depth*, the length of the context string. So for example a `2-call-site-sensitive` analysis would label each abstract method context with both the caller and the caller's caller.

3.2.3 Sound and unsound analysis

Some program analyses are **sound**, meaning that the analysis will not deduce false claims about the program in question. For instance, a sound race detector will not find non-races. Other program analyses are **complete**, meaning that (within some domain) all true claims will hold. For instance, a complete race detector would find all races. It is rare for analyses to be both sound and complete. In general, nontrivial properties of programs are undecidable, and therefore an analysis must approximate the program in some way to be decidable.

Some analyses are neither sound nor complete. Abandoning both goals means that there will be no theoretical guarantees about the effectiveness of an analysis. However, being willing to compromise on both false positives and false negatives can result in a lower rate for the two, together. This can be more important in practice than a strong theoretical claim coupled with severe imprecision or incompleteness. Much of the work on bug-finding by Dawson Engler et al. has this flavor [25].

3.3 Program Analysis for Documentation and Configuration

Moving up the layers of the software stack, we now turn from describing static analysis techniques to analysis applications. In particular, we describe work on using static analysis to help explain program behavior or functionality to developers and users. This work parallels the configuration-documentation techniques we will describe in Chapter 4.

The general topic of automatic documentation of program behavior has been addressed previously. Rubio-González and Liblit show that static analysis can catch incorrectly documented error code return values in the Linux kernel [66]. Kremenek et al. show that static analysis can find resource allocation and deallocation sites in real-world systems programs, without the benefit of annotations [43]. Buse and Weimer show that the exceptions thrown by Java functions can be inferred more accurately than they are currently documented [16]. Static analysis approaches can also extract higher-level properties of program behavior, such as file or network packet formats [50]. Wang et al. use dynamic taint tracking to find security-related options. [79].

Our approach to configuration type inference relies on the fact that many options are used in similar ways and that programmer- and user-oriented descriptions of options are a close match for program structures. Prior work has made similar observations about object oriented design patterns. Reverse-engineering design patterns from program code has been an active area of research since at least 1996. This work sought to report pattern use as a form of design documentation [42, 23].

Like this prior work, we use program analysis to remedy deficiencies in documentation. Unlike this prior work, we are deriving a comparatively high-level and informal property. Return codes and exceptions are aspects of program behavior that can be directly expressed in the semantics of the associated programming language, as can many object-oriented design patterns. Configuration is a library-defined abstraction; configuration option types are an ad-hoc human concept.

Another branch of related work concerns configuration-aware program analysis. Reisner et al. have used symbolic execution to model configurable programs [63]. They show that configuration options often localized effects: most options only affect a small portion of program state. This result is in accord with our findings. We observed that option use falls into patterns; most of the configuration patterns we saw are likely to have localized effects.

There has also been prior work in analyzing compile-time configuration; for example, Krone and Snelting discuss finding flawed configuration models [45].

3.4 Rewriting Programs

Chapter 6 of this dissertation presents a technique for improving the quality of program logs by retrofitting a revised logging library. Others have also addressed the issue of log quality. Yuan et al. describe LogEnhancer, a system for improving log messages. In their approach, the program is analyzed statically, and patches are emitted that add the values of important variables to already-existing log messages [87]. This is complementary to our work; their analysis is expensive, static, and improve a complex property. Ours is simple and can be done on-demand.

In some cases, it is possible to do this sort of program enhancement incrementally. Software dynamic translation is the general term for techniques that rewrite a binary as it runs. They can be thought of as just-in-time compilation from native machine code to modified native machine code. This translation can be done for arbitrary programs. The DynamoRIO project demonstrated

software dynamic translation for x86 binaries, with negligible overhead, even for programs with complex interactions with the operating system or self-modifying code [15].

3.5 Logs and Debugging

There are two debugging techniques that most programmers learn very early in their professional careers: using print statements and using a debugger.

Console debugging has been used since early in the history of computing. The use of `printf` as a debugging tool comes well endorsed. Kernighan and Pike advocated debugging log statements in *The Practice of Programming*: "It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is. More important, debugging statements stay with the program; debugging sessions are transient." [40]

Over time, developers adopted more sophisticated libraries to help with message labeling and filtering. The `syslog` RFC requires that log messages have a severity indication "so that the operations staff could selectively filter the messages and be presented with the more important and time sensitive notifications quickly." [52]

There have been efforts, in recent years, to replace ad-hoc logging with more structured tracing. XTrace is a logging system that relies on propagating causal labels via RPC. This allows log viewers to pull out causally linked messages [27]. Google's Dapper system is similar, but makes more comprehensive use of sampling [70].

Even though Kernighan and Pike disapprove, specialized debugger programs are a common programmer tool. The core features of a debugger, typically, are letting programmers pause the execution of a program, inspect values in memory, and alter them. This is less feasible in a distributed system, where pausing the execution of the system may be impractical. Debuggers are also primarily useful for programmers, who have some background knowledge on what a program "should" be doing — they are not helpful for end users who generally do not understand program internals and do not wish to learn.

3.6 Automated Configuration Debugging

Chapter 5 of this dissertation describes using static analysis to automatically debug configuration problems. The manual debugging techniques discussed above, while an essential part of programming practice, are costly in terms of developer time. These techniques, are also a poor fit for configuration debugging. Debuggers and log statements are designed to benefit the developer of a program; they are less suitable for fixing errors that arise when a user tries to set up or run a program without a deep understanding of its design and implementation.

We are by no means the first to look at the problem of automatically detecting and diagnosing configuration problems in computer systems. There is a substantial body of prior work on

configuration debugging. We present these techniques in depth, since the bulk of this dissertation is intended as an advance on this prior work. We group these techniques into four broad areas: program-analysis approaches, signature-based approaches, inter-host comparisons, and replay techniques. Though our focus is on configuration debugging, we also discuss more general debugging techniques that are potentially applicable to configuration problems.

3.6.1 Using Program Analysis

We are aware of two instances of prior work using program analysis for configuration debugging. Sherlog takes as input a program and a log, and uses a SAT solver to infer the state of the program prior to failure [86]. Like our work, this is a static approach that does not require modifying the execution environment. Unlike our work, the analysis requires output from the faulty program. Users might need to wait as much as half an hour for an answer. In our approach, users can get an immediate answer from the pre-computed static analysis. Our failure-context-sensitive analysis takes no more than a minute or two.

ConfAid uses dynamic taint tracking (and short-distance speculative execution) to explain errors [10]. ConfAid tracks tokens from specified "configuration sources", and is able to pinpoint the tokens that most directly lead to an error. (This technique is essentially the same as that used by Wang et al. [79].) This technique is likely to be more precise than ours and encompasses a broader definition of configuration. There are two disadvantages in comparison to our work. ConfAid requires the user to modify the execution environment for the program being diagnosed. Our approach uses only the generated error message. Our approach can attribute errors caused by an inappropriate default value for an option, while ConfAid can only track options that are explicitly set.

3.6.2 Using Problem Signatures

Human troubleshooters often follow a routine, following an implicit decision tree in locating problems. If a web server is malfunctioning, an administrator might first check that the machine responds to pings, next that the server process is running, and so forth. The PDA tool (Problem Determination Advisor) [35] can automate this process, conserving scarce administrator attention for the harder problems. The catch is that PDA's decision-trees and probes need to be hand-built.

Problem signature approaches seek to automate this process. They categorize problems by extracting a signature of the program behavior associated with a particular problem. This requires that a library of problem signatures be created for each program. In 2006, Yuan et al. proposed using an application's sequence of system calls as a way to identify previously-encountered problems [85]. A similar system was proposed by Ding et al. [20] in 2008. In their approach, the set of system calls made by an application are summarized as a dependency graph, and this summary is used as a signature to identify a problem. There are two limitations to this technique. First, signature collection (needed both from faulty and correct runs of a system) is expensive, imposing

a 20% overhead. Second, signatures cannot be easily shared across sites, since they include the specific configuration on each machine.

Convincing customers to install data collection tools that do not affect the execution environment and that only run sporadically is a major challenge. Convincing them to make their cluster substantially slower in normal operation is usually out of the question, particularly since there is a good chance that a bug in the modified runtime will cause new failures even while it helps debug the existing ones.

Negative performance consequences are not always present in signature schemes. In 2011, Yuan et al. proposed a system of this type, specialized for Windows Registry-based configuration errors. They observe that patterns of registry reads and writes are highly predictive of configuration failures and that restoring relevant registry state can help recover from errors. As a result, they are able to reduce the performance overhead of signature approaches. Their implementation has low CPU overhead, and while it consumed 500-900 MB of RAM in experiments, the authors expect that an optimized implementation could use as little as 1-6 MB.

However, this is not a fully general approach. It fails to catch errors induced at system setup time, when training data is absent. Hence it would be unhelpful for the problem of interest to us, aiding users trying to set up a system. It requires training data, sometimes per-site training data. It also seems highly tailored to Windows Registry errors, where a common source of configuration error is one program overwriting another's configuration option.

3.6.3 Comparing Failing and Working Configurations

Instead of examining program behavior to diagnose errors, it is also possible to compare configurations themselves. Strider and the PeerPressure system built on top of it are designed to help identify problems caused by bad Registry entries on Windows machines. Both systems rely on having a large user base and being able to compare working and non-working machines. Both are specific to registry-based misconfigurations, rather than covering a wide range of possible errors.

Strider [80] tries to pair a working and non-working machine, and constructs the set of configuration differences. It then takes the intersection of this set with the set of Registry entries read while the user attempts to perform the failing action. The authors demonstrate that the root cause of the problem is likely to be found in the intersection of these sets.

PeerPressure goes a step further, using the relative frequency of various settings as a clue to their likelihood of causing the problems [78]. Since most machines, it is assumed, are healthy, an unusual configuration setting read by a faulty process is likely the culprit. Similar approaches have been shown to work well to identify configuration problems in grid deployments [61].

The above work is specific to configuration. Similar techniques are applicable to debugging more generally. The most impressive general version of this is statistical debugging, as developed by Zheng and Liblit [90, 49, 89]. In this approach, programs are instrumented with code to check the values of various predicates at various points in the execution: "variable `f` equals 0," for instance. The application counts the number of times each predicate was executed, and whether it

was true or false. A trace consists of a count, for each predicate, of true and false invocations. Traces from large numbers of runs and sites are collected, and machine learning techniques are then applied. The result is a statistical "fingerprint" of a bug, for instance "if variable f is zero, the program is very likely to crash". These can help developers quickly focus their attention on the cases that are likely to trigger failure.

Statistical debugging has requirements that are infeasible to satisfy in many settings, however. It requires compile-time changes to programs, to collect the relevant information. Both users with and without problems must run this modified version. The program must have a large installed base, to get sufficient data volumes. Users, both with and without problems, must upload usage data to a centralized site. Effectively, both users and developers must invest substantially in the process to reap the reward. While this is sometimes possible (as we discuss below), few open source efforts have taken the plunge to date.

The Windows Error Reporting (WER) service [29] takes an approach to "debugging in the large" similar to statistical debugging, but without the need for instrumentation or program modification. Like statistical debugging, the goal of WER is to detect subtle patterns in crashes, in order to focus developer attention on the common elements. But rather than sample predicates continuously, WER collects data at crash time. A certain amount of data is collected by default, including register and stack dumps, and the code immediately around the crash site. The WER service classifies error reports as they are received, and can be configured to request additional data about hard-to-pin-down failures, including other services on the system at the time.

WER is not the only system of its type. Crash-reporting software is now fairly common. If a particular installation of the Mac operating system or the Firefox web browser crashes or hits difficulties, a crash report is sent back to the developers. These reports are likewise summarized, aggregated, and automatically triaged.

Both statistical debugging and error reporting are applicable to configuration mistakes, although these are not their primarily focus. To adapt them for configuration, the primary requirement would be to insert predicates specifically covering the configuration space of a program: which options are set, what formats their values have, and so forth.

All these comparison-based approaches require large installed user bases, plus relaxed privacy policies. Some, but not many, Hadoop user sites are willing to share their configurations. The specific software and workload on any two sites may differ radically. New Hadoop versions are released every few months, often with different configuration options, different known bugs, and so forth. Hence, statistical methods are not promising in dealing with problems encountered with specialized Cloud software systems.

3.6.4 Replay Techniques

Given k possible diagnoses for a problem, one strategy is to apply the fix for each possibility until one succeeds. This is the essence of replay approaches. In practice, this is done with some sort of sandbox, allowing diagnoses to be tried without the risk of overwriting correct configuration or damaging the rest of the system. All of the replay approaches we discuss are applicable to a

broad range of possible error causes, not just misconfiguration. In principle, replay approaches can debug any problem that can be fixed by altering system software.

Chronus [82], AutoBash [75], and Triage [76] are three notable replay-based systems. Given a once-working system, Chronus uses virtualization and a customized filesystem to pinpoint the particular configuration change that caused the system to stop working. The AutoBash system uses kernel-level speculative execution to test the results of various configuration changes [75]. Given a set of predicates (programs that return "passing" or "failing"), AutoBash can try many potential configuration fixes in the background, without interfering with the rest of the system, until some set of fixes solves the problem. In subsequent work, the AutoBash developers showed that a program's pattern of system calls is often a sufficient "fingerprint" for identifying a particular bug [9]. The Triage system [76] takes a similar approach, speculatively replaying the events leading up to a failure with small variation, in order to pin down the root cause of failure. AutoBash and Triage do this with host-based virtualization or speculative execution.

Delta debugging is an algorithm for efficiently searching through a large combinatorial space of possible interrelated causes [88]. Delta debugging relies on having a working state, a broken state, and a set of potential changes. Importantly, delta debugging correctly handles cases where a combination of changes leads to the error. Delta debugging on its own is likely insufficient for configuration debugging, however. In many cases, a program will have mandatory options, that must be set at a given site, meaning that a correctly-working configuration may not be available. As mentioned earlier, current versions of Hadoop will actually crash with a null pointer exception if started with the default empty configuration. Delta Debugging pushes more of the work of troubleshooting onto the user site — potentially a serious bar to deployment.

The program analysis approaches discussed in this dissertation are complementary to replay. Knowing which options a program can read and how those values are used may help determine which potential fixes are most promising, thus reducing the time taken to diagnose an error, assuming an appropriate fix is available to the replay system. More precise information about program option use (including option types) may enable automatic generation of potential fixes, making replay more widely effective for problems not present in the diagnostic library. Conversely, replay removes much of the cost of imprecise analysis by automatically trying multiple possibilities.

All of this work is predicated on replacing human debugging time with machine resources. Replay makes sense trying out a new configuration is relatively low cost and that failures can be detected automatically. This is not true in our space. If a critical nightly job fails once a week on average, that is a major issue, but replay will take several weeks to test each possible diagnosis. Hence, it is worth investing substantial human time to pick out the most plausible diagnoses first. Nor can automate processes be trusted to stop, reconfigure, and restart a cluster. Downtime is expensive and must be coordinated with other human users and business processes in ways that are not easily expressed to an algorithm.

We sum up this summary of related work with Table 3.1, which depicts the tradeoffs between the approaches we have described here and our own debugging approach, static precomputation.

Approach	Specific to misconfiguration	Needs large installed base	Runtime modification required	
			While debugging	Other times
Taint Tracking	Yes	No	Severe	None
Statistical Debugging	No	Yes	Mild	Mild
Signatures	No	Yes	Severe	Severe
Replay	No	Yes	Severe	None
Static Precomputation	**Yes**	**No**	**None**	**None**

Table 3.1. Comparison of Different Automated Troubleshooting Techniques

3.6.5 Why Big Data is Different

The support challenge for enterprise software differs significantly from that for consumer software. If Windows or Firefox crashes, the diagnostic data will cover just the machine where the failure appeared. With a distributed system, information from seemingly-correct machines must also be collected. Windows and Firefox are installed on hundreds of millions of machines, often with very similar configurations. Any particular Hadoop version might be installed on only a few hundred clusters. (For more niche systems such as HBase, the numbers may be lower still.) These clusters vary wildly; the smallest deployments are single-node development environments and the largest have over 8,000 cores and over 10 PB of storage. Likewise, the mix of software in use varies across customers.

Another distinction is that vendors of consumer software generally have no obligation to fix any particular failure; if users choose not to upload crash reports, the problems they see might not be fixed. If a problem is rare, it might not be worked on until it has been seen several times. In contrast, enterprise vendors often have contractual support obligations; problems must be addressed even at privacy-sensitive customers or installations that are not connected to the public Internet. And they must be fixed even if they are unique to a particular site.

The constraints of the big data environment therefore push us towards static analysis: this results in no production overhead, no requirement at reconfiguration, and no requirement for a large installed base. The price of static analysis is reduced precision. This is acceptable in a context where supporters have a range of tools and expertise. Adding an additional tool to the debugging toolbox is valuable, even if the tool is only useful some of the time.

Chapter 4

Inferring Configuration Options

In Chapter 2, we presented measurements to show that misconfiguration was a major source of failures in deployed cloud systems. Chapter 3 described today's best techniques for coping with these errors. This chapter and the next one together describe our approach to the problem. We show how static analysis can help developers and administrators manage program configuration. This chapter describes analysis techniques intended to reduce the rate of misconfiguration failures; the next one will describe techniques to help diagnose misconfigurations after the fact.

4.1 Introduction

As we mentioned in Chapter 1, this dissertation is primarily concerned with named options in a key-value paradigm. The key-value style of configuration is convenient for programmers, because it makes it easy to add new options incrementally. Developers need not maintain a schema or a centralized list of supported options. For exactly these reasons, however, key-value configuration can cause problems for users. The program can do little or no explicit checking for configuration errors.

A common (and particularly subtle) error is for user-written configuration files to assign values to options that a program never reads. This can happen for several reasons. Sometimes, a typing error results in the user not setting the option they intended. Sometimes, options change between versions and a configuration file becomes stale. And sometimes, users are led astray by bad documentation.

As we show, documentation sometimes claims that an option has a particular effect, when in fact that option is never used at all. Conversely, developers sometimes add configuration options without documenting them. In these cases, the new configurability is of limited use. The open source systems software we study all have significant undocumented configurability, suggesting that this is a widespread problem.

We suspect that bad documentation is a side-effect of the open source development process. Documentation is often updated separately from code, and can fall out of step with an evolving program [68, 71]. In many organizations, documentation is done by technical writers, not programmers, and information can be lost at this boundary. Past studies have shown that bad documentation is a significant bar to adoption of open-source software, as well as a considerable headache for users [48]. Excess configurability and poor documentation have been recognized as a problem for several years [55] but solutions have been slow to emerge. Programmers often distrust documentation, preferring to read source code to understand program behavior [47, 71]. While this attitude may be appropriate for programmers who are actively working on a software system, it works less well for users and administrators of that system who do not wish to learn about the internals of it.

In this chapter, we make three contributions. First, we document the ways that the key-value configuration pattern is used in a range of open source projects. We analyzed seven open-source programs, totaling well over a million lines of code and representing the work of hundreds of developers. We observed that configuration options tend to be used in standardized ways. There are modest number of use patterns that together account for more than 90% of options. We found pervasive documentation errors. Every program in our sample had documentation for options that do not actually exist as well as significant numbers of undocumented options.

These pervasive errors motivate our second contribution. We describe and evaluate a static analysis that outputs a list of configuration options potentially used by a program, their default values, and the program points where each option is read. This analysis finds more than 95% of the options in the programs we inspected. This accuracy rate is higher than in the documentation of most of the associated programs, making the analysis practical for finding documentation errors.

The analysis can help users as well as developers. In operational experience at Yahoo!, typographic errors in option names are a major cause of problems with Hadoop and related programs[1]. A Hadoop user can easily set the value of a non-existent option like `default.fs.name` when they meant to refer to a similarly-named real option, `fs.default.name`, instead.

And imagine the plight of the poor user who tries to set `dfs.datanode.max.xcievers`, but accidentally types xceivers a mistake easy for machines to diagnose, but hard for humans to spot. This is a particularly menacing case, both because the option name is peculiarly spelled, and because accidentally misspelling it will not result in any immediate overt failure. Instead, it will result in the cluster using the default value, resulting in impaired performance and reliability that may be challenging to track down.

There is no central list of valid options that can be used as a dictionary for a "spell check for configuration"; references to configuration options are scattered throughout the code. As a result, this sort of error produces few overt symptoms and can be frustrating to track down. The developers could potentially maintain a list of valid options for use in configuration checking, but the many flaws we found in program documentation suggest that the task is difficult and error-prone. Automated analysis, by reducing the burden, can help. This is illustrated by Figure 4.1.

Last, we describe techniques for automatically documenting configuration options. The primary goal of this analysis is to document the domain of valid values for an option – effectively,

[1] Owen O'Malley, Yahoo!, Personal communication, January 2010.

to infer a type. We accomplish this by automatically matching the specific patterns by which developers use configuration. This approach finds types for most options in our sample, and has few false positives. This analysis builds on the previous one, consuming the set of options read by the program. It also builds on (and validates) the results of our empirical study: recognizing patterns in code makes sense because a small set of patterns covers a large fraction of options.

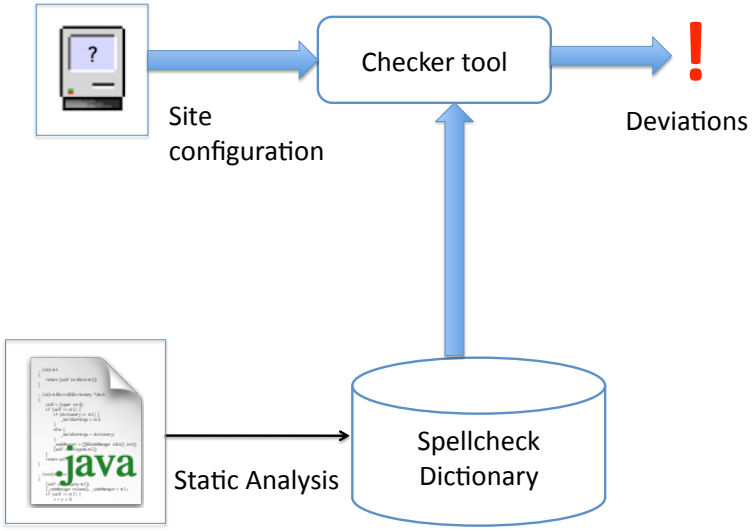

Figure 4.1. Illustration of overall approach to "configuration spellcheck". Thin arrows depict precomputation. Thick arrows represent action in response to user activity.

There are several applications for this analysis. It can be used to produce a first draft of program documentation for humans. It can also produce machine-readable specification for the permissible values of options. There has been work by the systems community on automatic performance tuning, for instance [24], and by the software engineering community on configuration-aware resolution of reflection in method calls [67]. Being able to automatically find tunable numeric parameters or class-name parameters would be helpful in these contexts.

Finding constraints on option values, even on a subset of options, would help catch additional user configuration errors. Our analysis can automatically determine not only that `hadoop.util.hash.type` is an option, but that the only possible values for it are "murmer" and "jenkins". Programs do not consistently report configuration errors [39]; often, they silently substitute a default value. Extracting constraints on option values enables static checking for these sorts of mistakes, potentially saving hours of user time if a bad configuration value causes a long batch job to go awry. This extends our "spell check for configuration" to validate values, not just names.

Application	LoC	KB compiled	Numeric	Identifier	Mode	Other	All options
Cassandra	36,823	1,961	17	14	3	2	36
Derby	1,136,718	7,903	16	17	23	13	69
FreePastry	175,085	6,073	154	8	39	10	211
Hadoop	167,653	5,440	89	70	31	16	206
HBase	104,781	3,149	38	20	3	3	64
JChord	35,761	1,209	5	26	14	12	57
Nachos	10,896	467	2	4	10	0	16
Total			321	159	123	56	659

Table 4.1. Numbers of options by application, with breakdown by type of option. KB compiled = size of compiled binary, in kilobytes, excluding third-party libraries.

We opt for static, rather than dynamic analysis. Many options are used only in particular program modules or as a result of particular inputs. Hence, dynamic testing has poor prospects for finding all uses of configuration options. Static analysis can attain high coverage much more easily. Our analysis uses standard points-to and call graph construction algorithms. Our code bases of interest make heavy use of reflection and remote procedure calls. This requires some tailoring of the underlying static analysis substrate. Those aspects of our analysis are discussed in Section 4.4.2.

We begin in the next section with our empirical study of program configuration mechanisms. In Section 4.3 we show how static analysis can find the set of options used by a particular program: our results are evaluated against the findings of our manual inspection. In Section 4.4, we describe and evaluate an analysis framework for determining the types of these configuration options. This second analysis builds on that presented in Section 4.3. Section 4.5 discusses the challenges of applying static analysis to complex code-bases and how we dealt with them. Section 4.6 discusses ways to generalize our work, avenues for future work, and some experiences using our work to improve the quality of the Hadoop code-base.

4.2 Quantifying program configuration options

To better understand program configurability, we looked at the configuration mechanisms in a range of existing software projects. For each program, we consulted the documentation and default configuration files to derive a list of options. We then manually classified each option. The projects in question were summarized in Section 2.1. We did not examine logger configuration or supplemental configuration files used by optional components .

Type	Category	Total
Time Interval	Numeric	118
Count	Numeric	115
Boolean	Mode	104
File	Identifier	60
Size	Numeric	53
Class Name	Identifier	38
Address	Identifier	28
Fraction	Numeric	28
Mode name	Mode	18
String	Other	17
Port number	Identifier	13
Internal ID	Other	9
Password	Other	8
URI	Identifier	7
User ID	Other	7
Network Interface	Identifier	5
Other	-	31
Total		659

Table 4.2. The most common configuration option types, with classification.

4.2.1 A Taxonomy of Configuration Options

Rather than impose a taxonomy *ex ante*, we built ours bottom-up, describing each option as we encountered it and then looking for patterns. We show statistics for each of the individual programs in our study in Table 4.1. We also present background statistics about each program. As can be seen, there is substantial variance across these programs: the absolute number of identifiers and the proportion of identifiers in each column varies widely. FreePastry, for instance, has a very large number of controllable timers that boosts both the total and numeric columns. Table 4.2 shows our list of option types and how many instances we found of each type of option, summing across all the programs in our study.

Several categories need explanation: a *count* is an integer parameter describing how many of some entity should exist, such as threads in a pool, iterations of a loop, and so forth. A *size* is a quantity of memory or storage, measured in bytes. A *mode name* is a string, drawn from a small set, that selects how a program should behave from a small set of options. An *internal ID* identifies some program-defined entity, such as distributed files in the case of Hadoop. "String" and "Number" are catch-all types for string or numeric options that are not used in some other well-defined way by the program. For instance, JChord can run a target program using dynamic instrumentation; the labels used to indicate each test run are uninterpreted strings.

Most options fell into a handful of types. The top three categories together account for slightly over half of all options seen. Numeric options are very common and are primarily used for a small

number of purposes: to control timers, resource pool sizes, and memory allocation. Non-numeric options are overwhelmingly external identifiers, often designating files, network addresses, or Java classes.

Some options include complex structured data. Examples include regular expressions, date format strings, or command line arguments for a subprocess. These complex options are rare in the programs we have inspected: we found a total of four options controlling process arguments and three options with nontrivial internal semantics: one regular expression and two `strftime`-style date format strings.

The list of option types can be further summarized. Most options fall into one of three broad categories. The largest category of options is tunable *numeric parameters*, controlling buffer sizes, time-out periods, and so forth. (This includes both integer and floating point options.) Another large group of options select a *mode of operation* from a small set. This includes Boolean options, as well as mode names. The last group of options consists of *external identifiers*: strings or numbers that refer to some entity outside the program, such as file names or network addresses. Java class names are effectively also external identifiers, since the runtime maps them to the names of files in the Java class path. A handful of option types remain outside this tripartite classification. Internal identifiers, opaque strings or numbers, and a few miscellaneous types such as passwords do not fit neatly into any of the three categories mentioned above. These are tabulated as "other" above.

While our ad-hoc approach worked reasonably well, there were a few difficulties. Some options can take a list of values. We treated "Type" and "list of Type" as equivalent. When a string is used as a suffix to a file path, we count it as a string, not a file name. Several Hadoop options are path names to files in a distributed filesystem. We consider these to be internal identifiers, not file names.

4.2.2 Configuration APIs

We also looked at the structure of the code each program used for reading and processing options. Of the seven programs in our sample, six had a narrow and well-defined API for configuration. In each, there was exactly one class that exposed a key-value interface to the rest of the program, through which configuration options could be read. These classes offer a set of methods for retrieving configuration values of particular types: `getBoolean`, `getInt`, and so forth. Each method takes the name of an option as parameter and optionally a default value to be used if the option is not set. The Java System Properties API, part of the Java Platform standard, offers this interface as well. The Unix system environment is also a key-value map. We therefore conclude that this style of interface represents a popular and widespread programming abstraction.

Derby was the one partial exception we saw to this pattern. In Derby, there are three levels of configuration, consulted in turn: global options specified in a file, per-database options, and options specified programmatically. Each option can be set in some subset of those tiers. A substantially more complex API is needed to manage configuration. There are configuration retrieval methods in several different classes, differing in which locations are searched for configuration. Each of these

configuration retrieval methods, however, followed the standard pattern, taking a string argument for the option name plus an optional default value.

The concrete syntax for configuration varied significantly. Hadoop and HBase use an XML-based format containing key-value pairs. Cassandra uses a more complex XML structure, with nested elements; the program reads elements from this file using XPath queries in an essentially key-value style. The remaining programs use a flat ASCII file with a list of `name=value` pairs.

Some of the programs in our study also accept command-line arguments. In many cases, these arguments duplicate the functionality of configuration file options or are accessed via a similar key-value interface. Hadoop largely eschews command line options; most of Hadoop's component programs only accept options that set configuration values.

4.3 Finding options

In this section, we answer two research questions about configuration options: how good is existing configuration documentation and how well does static analysis do compared to this human standard. This is motivated by our desire to have automated tools check documentation, or even produce the authoritative version.

Today, developers looking to extract a list of configuration options from source code would have to resort to searching through the text for calls to configuration read methods. This approach cannot readily find all option names. In the code we examined, we saw many examples of application methods that take an option name as parameter. As a result, there can be several function calls between the string literal for an option's name and the point where that name is passed to a system or library-defined configuration method. All of the programs in our study define several utility methods that take an option name and return a new object corresponding to that name. For example, in Hadoop there is a method that take an option name as parameter, read the value of that option, and reflectively creates and object of the class named by that value. Hence, finding which strings are used as option names requires inter-procedural analysis to track these string constants through method calls.

Simple lexical approaches would yield less precise information than our analysis. All of the software packages we examined consist of multiple executables sharing large portions of code. It can be helpful to know which component programs will use which options, or even whether an option is only read in dead code. Our technique derives this information readily, but lexical techniques cannot.

4.3.1 Approach

Our approach is outlined in Figure 4.2. We break the overall problem into two major pieces: First, finding the program points where options are read or written; second, finding the possible option names at each of these points. These two stages are somewhat independent; alternate al-

```
Construct points-to and call graphs.
Mark known configuration methods.
Mark option-name arguments to these methods.
while (not converged to fixpoint)
  for each method m:
    if an argument of m used as an option name
      Mark method as conf. read call.
      Mark argument as option name.
Find possible string params at call sites
Output option names and read points.
Output methods taking option names as arguments.
```

Figure 4.2. Pseudocode for analysis to find options

gorithms can be used in each without disrupting the overall structure of our approach. We output both the set of program points that read options and a regular expression for the options read at each of these points. This means that our analysis is independent of the syntactic details of configuration file format. Instead, we rely on the underlying APIs, which tend to be more similar across programs than syntax.

The usual API for configuration consists of a set of related calls, each of which has a string-typed argument corresponding to the option name. We assume that we have either annotations on these API methods, or, equivalently, a list of methods returning or setting configuration. For this study, the annotations required were compiled into the code of the analyzer. For each program, these annotations consisted of a few lines of the form "include all methods in class `Properties` whose name starts with get." These annotations also specify which parameter is the name of the option. (This is generally the first parameter of string type, in our experience.)

We expand this set of configuration-reading methods by finding all methods taking a string argument, where that argument is passed as an option name to an already-discovered configuration method. This accounts for the common programming pattern of having wrappers around other configuration calls to encapsulate type conversion. For example, FreePastry implements a method `getInetSocketAddress` that takes an option name as a parameter and returns a socket corresponding to the value of the option. Our analysis discovers that the method uses one of its arguments as an option name. We therefore infer that `getInetSocketAddress` is itself a configuration-reading method and that its return value corresponds to that option name. Finding these additional configuration read calls lets us find the earliest configuration read point in a call chain. Effectively, we are treating configuration reads context-sensitively, without the expense of a full-program context-sensitive string analysis. This improves the precision of subsequent analyses, including those discussed in the next sections.

Once we have the set of option read points, we find the string parameters potentially passed to each read call. Most configuration options are named by compile-time constant strings. Sometimes, however, option names are constructed dynamically. A common pattern is to construct configuration option names out of several components, of which just one is variable. For example,

in Hadoop, the implementation class used to access a filesystem of type *t* will be the value of option `fs.`*t*`.impl`.

We implemented a string analysis to capture option names constructed as a fixed sequence of variable and constant fields, using the any-string regular expression $.*$ to handle dynamic inputs. For the example mentioned above, our code produces the regular expression `fs.\.*\.impl`. This analysis captured nearly all the dynamic option name creation we saw in our programs. This approach was scalable, easy to implement, and integrated cleanly with out points-to framework. More sophisticated string analyses (such as JSA [18]) could be used without changing our overall approach to finding configuration read points.

Programs sometimes set the values of configuration options at runtime. This can also be analyzed statically. There is one significant difference, however. When an option is read, the value may be used by the program, and so downstream analyses benefit from knowing the context in which the option is read. In contrast, values do not propagate from a write and so knowing the calling context by which a particular option was written is much less useful. Hence, we can dispense with the recursive method-finding for option writes. Instead, we allow string constants to flow to the underlying write call.

4.3.2 Implementation

We have implemented the above analysis for Java bytecode. This means it is applicable to programs written in Java (whether or not source is available) as well as programs in other languages, such as Scala, that compile to Java bytecode. Bytecode is a convenient target for analysis, intermediate between source and binary levels. While bytecode is simpler to analyze than true machine language, it is closely enough related that we can reason about what would need to change in the machine language context. Conversely, a working bytecode analysis can trivially be applied to source code, by first compiling the code.

Our implementation relies on the JChord program analysis toolkit [57]. Our points-to and call graph construction is based on the standard algorithms in JChord. The analyses are context insensitive, flow insensitive, and field sensitive. We use the SSA-representation of the program, as advocated by Hasti and Horwitz [31]. We resolve reflection using the cast-based technique described in [51]. Given a list of program entry points, we use rapid type analysis [11] to find the set of potentially-reachable code.

4.3.3 Evaluation Methodology

We evaluated the performance of our technique by running our prototype on each of the seven software packages listed in Section 4.2. Our research goal was to measure the effectiveness of the technique, as compared with the existing human-written documentation.

We are looking for both undocumented and unused options. By "unused," we mean options that are never referenced anywhere in reachable code. By "undocumented" options, we mean detected

options, defined by the project in question, not mentioned in any user-readable documentation associated with the package, including its website.

There are two special cases in determining whether an option is undocumented. Configuration options are sometimes used by one part of a program to affect another part, rather than as a way for users to alter the program. In these cases, the lack of documentation for users is not a problem. Accounting for this, we do not report an option as undocumented if its value is set programatically. Similarly, options defined by incorporated libraries are not treated as program options. Java uses system properties both to collect user configuration and also to expose runtime information, such as the JVM version.

Our procedure was to run our analyzer on each of the software systems When there were mismatches between our output and the documentation, we manually checked the program's code, searching for substrings of the option names in question.

4.3.4 Results

Table 4.3 represent our best effort at reaching "ground truth" on program configurability in terms of both undocumented and unused options. As can be seen, errors are common, with many examples of both unused and undocumented options, across all the projects we studied.

Table 4.4 shows how well our analysis does at matching this manually-generated ground truth. Our analysis found just over 96% of documented options that actually exist. Our tool failed to find uses for 61 documented options. For a third of these, we were able to manually find uses of these options. The remaining two-thirds appear truly unused. In some cases, we found commented-out code referencing these options, suggesting that they used to exist and have since been removed.

Put another way: When our automated analysis and the program's documentation disagree about whether an option exists, the automated approach is more likely to be correct. This also is true on a per-program basis. Our analysis is more accurate than human-written documentation on five of the seven projects.

As another test, we modified Hadoop to record when an option is read. We then ran a basic workload, starting a cluster, running a word-count job, and evaluating the results. We looked for cases where an option was read at runtime but not found by analysis; we observed none. This demonstrates that our analysis has high coverage on this code-base.

Looking at the undocumented and unused options, we noticed a number of patterns by which these documentation errors arose. We begin by describing the undocumented options.

Many undocumented options appear to be new and specialized features, added for some specific purpose and not yet documented. Hadoop and HBase, which are production systems with many users, have disproportionately many of these specialized and undocumented features. JChord is a research system, and this also results in undocumented options. The pattern seems to be that researchers add additional options in their portions of the system without documenting them.[2]

[2]JChord does not have a formal release process; we are comparing in-development sources to in-development documentation.

Project	Unused	Opts.	Undocumented	Opts.
Cassandra	2	6%	3	8%
Derby	2	3%	26	38%
FreePastry	24	11%	12	6%
HBase	1	2%	21	33%
Hadoop	6	3%	34	17%
JChord	3	5%	29	51%
Nachos	3	19%	3	19%

Table 4.3. Manually confirmed documentation errors. Percentages are of all documented options for each project.

Application	Found Unused	True Unused	False positives
Cassandra	3	2	1
Derby	9	2	7
FreePastry	24	24	0
Hadoop	10	6	4
HBase	9	1	8
JChord	3	3	0
Nachos	3	3	0
Total:	61	41	20

Table 4.4. Accuracy in finding unused options

Undocumented options had a distribution of types similar to documented options. Numeric options were about half of all options, with the rest primarily modes or external identifiers. Some options are described in the program source code as "deprecated;" unexpectedly, these do not appear to be a major source of undocumented options. These options were often still referenced in documentation, even if only to describe them as deprecated.

In Hadoop and Derby, we noticed an additional pattern. Undocumented options were being set in test code and read in the application code. (We filter out options that are set in the main body of a program's code. The options we discuss here are set in unit tests, only.) These options appear to have been added solely for the benefit of test writers. For instance, Hadoop has several timers controlling activity that normally happens hourly or daily. By setting an undocumented option, unit tests can make the behavior in question occur much more quickly. Test-only options accounted for half the undocumented options in Hadoop and somewhat over half for Derby.

We observed the following pattern by which unused but documented options arise. Sometimes, an option makes sense in the context of a particular implementation of a program feature. Sometimes, the feature is rewritten in such a way as to make the option unnecessary or meaningless. but the documentation is not updated. This is a particularly common pattern for when unused options relate to specialized features, added in the past for exploratory purposes and since removed. We

noticed this particularly in FreePastry, which is a research testbed with many unused options. The unused options mostly correspond to features or modules that are no longer present in the code. In some cases, the code to read the option is still present but commented out.

We now turn from flaws in software development methodology to flaws in our analysis methods, discussing the reasons why our analysis does not find all option uses. By far the largest problem, accounting for 10 of the 20 false positives, was code that performed significant string manipulation on option names. HBase and Derby break the key-value model slightly, iterating over a subset of options and renaming them before use. Our analysis is not sufficiently sensitive to determine the set of names being iterated over. Path sensitivity would be required to model code of this sort accurately. Some errors are caused by more traditional limitations of static analysis. Four options in Derby are read in code invoked indirectly via native code in the system library, where no single-language analysis can easily follow.

Some systems, including Hadoop, do a form of macro substitution for option values. In Hadoop, if the value of a configuration option includes the name of another option, wrapped in braces, the value of that included option is substituted. Nothing in the Hadoop code indicates that the substituted-in option would be read. This caused one false positive for our analysis. This could be handled as a special case in practice: any option lexically included in this way in a configuration file should be marked as used.

4.3.5 Practical Experience

We used this analysis in an industrial context at Cloudera, the Hadoop services and support company we described in Chapter 2. We found several industrial applications, which we summarize briefly below.

Improving the platform We used the results of this analysis to find undocumented options in Hadoop version 0.23, before its release. Some of these have now have been described for users. We also noticed some undocumented options that should have been removed entirely from the code-base. The option had been renamed, and the previous names should have gone away entirely. But instead, the old option was used in some contexts. This is a bug, since it means that a user's expressed configuration will be ignored in some contexts. These bugs have been fixed. We found no documented-but-unused options in Hadoop 0.23. This would have been painful to audit manually, but was easy to check using our analysis.

Guiding Cloudera Enterprise development The listing of options was used by the Cloudera Service and Configuration Manager (SCM) developers to check that SCM correctly handles all the known options in Hadoop. By pinpointing where in the code an option is read and which daemons may read it, this analysis helps SCM avoid setting MapReduce options on HDFS-only nodes, and similar mistakes.

Debugging user configurations We also use these configuration option listings to help diagnose user problems. One of the ways Cloudera delivers support economically is by automating some failure diagnosis. When users upload configuration, the options set by the user are compared against the extracted (though hand-edited) list of options. We found many instances where users

set a configuration option that was defined in some version of Hadoop other than the version they were running. This is an easy mistake to make if a user reads documentation from a different version than they are running, or if they upgrade Hadoop without editing their configuration files.

As we noted, some of the options picked up by our analysis that are irrelevant for most documentation purposes. Happily, these did not pose significant problems in practice. Programs and libraries often have a shared prefix for their options (`java.*` for VM-defined options or `hbase.*` for HBase options), making it reasonably easy for humans to determine which options belong to which code base.

We developed one notable extension for this industrial use. Option read calls often have a default value. Since we know where options are read, this default can be readily extracted. This proved helpful to supporters. One important aspect of support is recommending configuration values to users. Without knowing the default, it is not so easy to know if a user should set a given option. Automatically extracting default values is also useful when documentation is being written.

Subsequent to our work appearing, members of the Java Pathfinder community developed a similar, though simpler, analysis [53]. Like us, they used static analysis to extract configuration option names. Unlike our work, they did not do interprocedural analysis to derive method names. This limits both the comprehensiveness and generality of the analysis; in particular, it means that a wider range of methods must be explicitly marked. This JPF experience is effectively a replication of our finding that options are amenable to static analysis.

4.4 Categorizing options

In Section 4.2, we noted that the large majority of the configuration options we encountered belong to one of a fairly small number of types. In the programs we inspected, these types often correspond to specific programming patterns. By finding the pattern, we can infer the type of the option. We start by discussing potential uses for the analysis, setting the standard against which its performance should be evaluated.

We are not attempting to replace human-written documentation. We have three goals: helping developers write documentation, helping developers spot mistakes, and producing machine-usable annotations on options to help user troubleshooting. In all these cases, false negatives are fairly innocuous: an incomplete but accurate analysis is helpful, but false positives can be confusing. Consequently, our analysis is tuned to return "don't know" rather than to make wrong guesses.

Option names, while often descriptive, are sometimes misleading. Hadoop includes an option `mapred.skip.map.auto .incr.proc.count`. Despite the name, this option is actually a Boolean. Printing inferred types alongside the names of undocumented options can help remind programmers what the option does.

Inferring types helps developers check that options are being used in the ways they expect. We have seen options that are read, stored, and logged, but put to no substantive use. The analysis we present here can help developers spot these cases. If the analysis finds an unexpected type for an option, that is suggestive of an underlying bug or incorrect documentation. One striking example

concerns the `ij.exceptionTrace` option in Derby. This option is documented as a Boolean, but our analysis was unable to determine a type. On inspecting the code, we discovered that the value was never used at all: the true states of the option were "set" and "unset". Setting any value, even "false" would enable the option; surely an unexpected behavior. We also spotted several options in FreePastry that were documented as being time intervals, but that were never used this way. They were read into the program and logged, but the code to use them was commented out.

This analysis also enables stronger automated checking of configuration files, effectively a "configuration spellcheck" for values as well as option names. This is only meaningful for some option types. For example, almost any string is potentially useable as a file name or password. Fortunately, our analysis works well on most types for which invalid values can be readily flagged.

The approach we take is to look for patterns in how programs use configuration values. The success of this approach helps validate our taxonomy of configuration options, since a type that corresponds to a well-defined programming pattern is likely to be a useful abstraction. Just as FindBugs [34] and similar tools exploit a library of recognizers for various types of bugs, we envision a separate recognizer for each type of configuration option. Because the number of common option types is limited, the implementation effort required is reasonable.

We have implemented recognizers for most the option types listed above in Table 4.2: Booleans, class names, files, fractions, network addresses, network interface names, mode names, and port numbers. (We refer to these as "recognized" types). Our analysis splits numeric options into "time", "port number", and "other," rather than attempting to distinguish "counts" from "sizes. In our sample, there were 528 options found by the option-finding analysis and belonging to recognized types. This is the set against which we evaluate our analysis.

4.4.1 Approach

We exploit three basic techniques in recognizing option types: looking at the return types of configuration reads, looking at which library methods configuration data is passed to, and looking at which values are compared with one another. We assume the presence of a call graph, a points-to analysis, and the results of the above analysis specifying which configuration options are read at each point.

The simplest of our three approaches is to inspect the return type of the configuration call. Many configuration are read using typed methods, such as `getBoolean`, `getFloat`, and so forth. If an option is exclusively read via `getBoolean`, then we conclude that option can only hold Boolean values.

Some programs read options as untyped strings, and then convert them to the correct type. Our second technique handles this case. Instead of looking at how values enter the program, we look at how they leave the program: which library calls they are passed to. For example, if a string is read from a configuration option and then passed to the library `parseBoolean` method, we can infer that its value is expected to be "true" or "false".

This technique is particularly geared to handling identifiers. There are a small number of

common system or library calls for opening files or resolving host names. We use approximately 30 rules to cover the option types in our classification. These rules are stored in a simple lookup table, mapping from called method and argument number to inferred option type. We refer to this table as the *argument type inference table*. We display this table below, as Table 4.5

```
String s = getConf("option name")
if(s.equals("foo))
  setMode(FOO)
else if(s.equals("bar"))
   setMode(BAR)
```

Figure 4.3. Example showing a common programming pattern for parsing mode options. This pattern can be recognized automatically.

Mode options, which have a small fixed set of valid values, are an important special case. Here, we are often able to not only determine that an option represents a mode choice, but can also determine the set of valid values. We have seen only two patterns by which programs use mode variables. Often, programmers parse these options by comparing the returned value against a sequence of string constants. Figure 4.3 illustrates this pattern. We are able to retrieve the set of valid option names by inspecting the strings a given configuration value is compared with. Alternatively, programmers can define an Enumeration class, and then use a standard library function to create objects of this class. Here, we can retrieve the set of valid values from the associated Enumeration class.

Similarly, we can often infer the parent type that a configurable class must have. When a configurable class name is used to create an object reflectively, and that object is cast to some type T, we infer that the permissible values for the options are subtypes of T.

To help distinguish time-valued options from other options, we employ our third and last technique. There are a handful of library calls for reading the value of the system clock. If an option's value and a known time value are used together in arithmetic, we assume that the option is likewise a time value. This works well in practice: we are able to find 90% of time options, with a 10% false positive rate. This approach is intrinsically imperfect: We have seen configuration options that are multipliers for time values. In these cases, the multiplier should not be marked as "time" – it is a dimensionless number, not a time period.

The whole option-finding analysis runs quickly enough to be used in the software release process. Analyzing Derby, the largest program in our set, took less than 30 minutes using a recent-model laptop with 4 GB of RAM and a dual-core 2.2 GHz processor. Most of the running time was used in the underlying points-to analysis, not in the type inference *per se*.

4.4.2 Implementation

As with the previous analysis, we used JChord to implement our option type determination. Unlike the previous analysis, finding option types requires a whole-program dataflow, tracking the

Called Class	Method	Arg number	Inferred Type
java.net.InetSocketAddress	<init>	1	Address
java.net.Socket	<init>	1	Address
java.net.InetAddress	getByName	0	Address
java.lang.Boolean	parseBoolean	0	Boolean
java.lang.Boolean	valueOf	0	Boolean
java.lang.Class	forName	0	ClassName
java.lang.ClassLoader	loadClass	1	ClassName
org.apache.hadoop.conf.Configuration	getClassByName	1	ClassName
java.text.SimpleDateFormat	<init>	1	DateFormat
java.io.File	<init>	1	File
java.io.FileReader	<init>	1	File
java.lang.Double	parseDouble	0	Fraction
java.lang.Float	parseFloat	0	Fraction
java.lang.Double	valueOf	0	Fraction
java.lang.Float	valueOf	0	Fraction
java.lang.Double	<init>	1	Fraction
java.lang.Float	<init>	1	Fraction
java.lang.Integer	valueOf	0	Integral
java.lang.Long	valueOf	0	Integral
java.lang.Integer	<init>	1	Integral
java.lang.Long	<init>	1	Integral
java.lang.Integer	parseInt	0	Integral
java.lang.Long	parseLong	0	Integral
java.net.NetworkInterface	getByName	0	NetworkInterface
java.net.Socket	<init>	2	Portno
java.net.InetSocketAddress	<init>	2	Portno
java.util.regex.Pattern	matches	0	Regex
java.util.regex.Pattern	compile	0	Regex
java.util.Random	<init>	1	RandomSeed
java.lang.Enum	valueOf	1	Special
java.lang.Thread	sleep	0	Time
java.lang.Thread	join	1	Time
java.nio.channels.Selector	select	1	Time
java.util.Timer	schedule	2	Time
java.util.Timer	schedule	3	Time
java.net.URI	create	0	URI
java.net.URI	<init>	1	URI

Table 4.5. Argument-type inference table

	Options	Fraction
Real documented opts. of recognized types	528	100%
Success	433	82%
All failures	95	18%
Out of scope	33	6%
Time/not-time miss	27	5%
Wrong guess	12	2%
No guess, other reason	23	4%

Table 4.6. Overall success rate for type inference and partial breakdown of errors. Percentages are of documented options of recognized types found automatically.

values returned from configuration read points. As in taint tracking, the results of arithmetic on a labeled data item inherits the label. Unlike standard taint tracking, we need not include all dataflow paths nor do we model control dependencies. Rather, we only track values until the first use that suffices to determine their type.

To cope with the complex, reflection-heavy programs in our study, we made several modifications to JChord's default algorithms. Most of the programs in our sample are networked services, making heavy use of remote procedure calls (RPCs). This means that much of the code in these programs is never invoked directly; rather, these methods are invoked via reflection. We handle this by specifying a list of extra entry points for each program. These lists had an average of three entries per program.

In Java, method dispatch depends on an object's dynamic type. Correctly modeling method calls on externally-supplied objects therefore required us to modify the underlying points-to analysis. We fall back on type-based alias analysis for these objects [21]. We add an abstract object for every type. When an entry point is invoked externally, we assume that reference arguments point to the appropriately-typed abstract objects. Reference-typed fields in abstract objects point, in turn, to other abstract objects of the appropriate types. In the programs we examined, RPCs are invoked on singleton "server" objects. As a result, this abstraction incurred no loss of precision.

In our experience, configuration options are seldom shared between the program in question and component libraries. To improve performance, we exclude library code from analysis. In most cases, once a value is passed into the library, it ceases to be tracked. In two cases, however, we explicitly model the behavior of library classes. Our points-to analysis is collection aware: if an object allocated at site h_1 is stored into a collection allocated at site h_2, then subsequent reads from that collection can return the object with site h_1. We also explicitly model the string-to-primitive conversion functions in the JVM, thus handling code that, for example, reads an option as a string, converts the string to an integer, and uses that integer as a time delay.

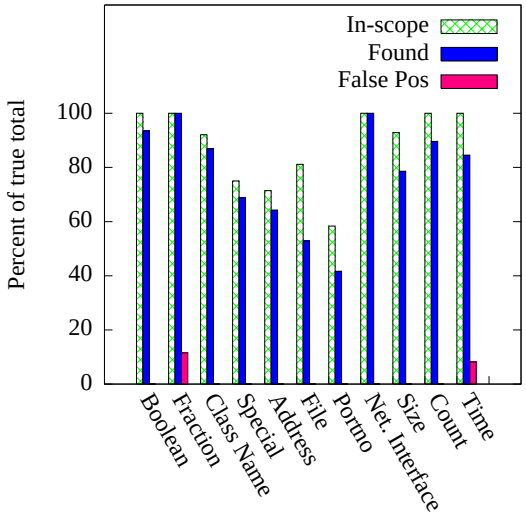

Figure 4.4. Accuracy in detecting option types. Only includes recognized types. In-scope means the option is used within the program being analyzed, not just in a library or external process.

4.4.3 Results

We compared the results of our automated analysis to the manual labels we described in Section 4.2. Our results are summarized in Table 4.6. We look only at options found by the option-finding analysis discussed in Section 4.3; we want to measure the effectiveness of type inference and option finding separately.

Overall, we succeed 80% of the time. There is substantial variation in success rate by type. Figure 4.4 displays this variance. False negatives are the gap between 100% and the "Found" bar. False positives (where the analysis finds a wrong type) are labelled explicitly.

The types are ordered based on the primary means we used to recognize them. The first two, Boolean and fractional, are recognized primarily based on the return type of the configuration read call; 24% of options of these types were recognized using called methods, our second technique. The next several columns are recognized entirely by this second technique. Time options, the last column, were found using our second and third techniques. Just over 80% of time options were detected by their use in arithmetic with known clock values; the remaining 20% were found based on their use as arguments to API calls such as `Sleep`.

Our analysis recognizes virtually all options whose values are Boolean, fractional or Java class names. It also does reasonably well for network addresses. Performance on files and port numbers is comparatively weak, for reasons discussed below. In distinguishing times from other numeric parameters, we succeed approximately 90% of the time; errors are roughly symmetric between false positives and false negatives. This imprecision accounted for 28% of misses.

We found few cases where documentation incorrectly described what an option did. We posit that developers add or remove options more frequently than they change the type of an existing

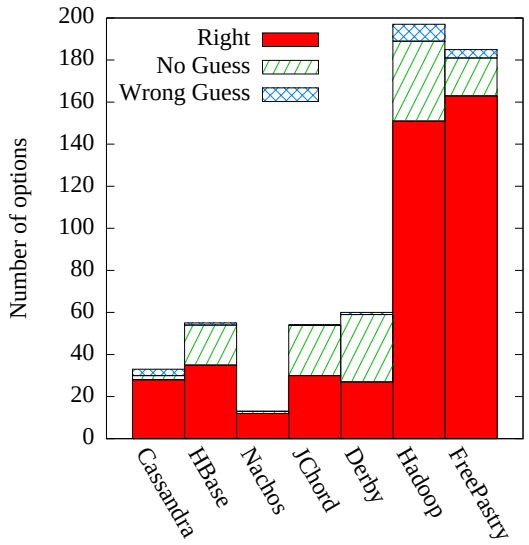

Figure 4.5. Success by application. Includes all options, not just those of recognized types.

option. Changing the meaning of an option would likely break existing configurations, a major enough change to trigger documentation changes. Mistakes in documenting an option's type do sometimes occur, however. The peculiar option in Derby mentioned above where "false" is treated as true is an example.

Despite its limitations, we believe this technique has practical uses. Ignoring the imprecision in detecting time options, incorrect type inferences happened on fewer than 3% of options. Our technique is accurate for several common types that have syntactic constraints. Our analysis is precise enough to accurately warn users when they put a string other than "true" or "false" for a Boolean option, or a similarly invalid value for numerical or class name options.

4.4.4 Sources of Error

This analysis has several sources of error. Sometimes, options are read and then passed to an external process before being used. This was the biggest single source of imprecision for our analysis. These externally-used or "out of scope" options accounted for 30% of all failures to infer a type, and almost half of misses on non-numeric options. This problem showed up primarily for identifiers, particularly file and host names. Since these refer to system abstractions, they can be readily passed to a subprocess or a library. The meaning of a particular class name or Boolean option is more often confined to a single program and therefore the uses for options of these types are generally in-scope.

String operations were another major source of imprecision. We do not track the contents of every string variable. If a configuration value is stored inside a string and then later parsed out

and used, we cannot report a type for the option. This is a significant issue with file names, port numbers, and network addresses: there are standard programming idioms (such as constructing a host:port pair) that defeat our analysis. This could potentially be fixed by a more sophisticated string analysis.

Some of the imprecision in finding time options is due to the presence of numeric utility functions. Since our analysis is context insensitive, we see time options "leak" through these utility functions. Context-sensitive analysis would help solve this problem. Otherwise, notably, points-to and dataflow imprecision did not appear to be a major problem.

4.5 Coping with Complex Code

In the previous sections, we avoided discussing certain technical details of our analysis algorithms, opting instead to emphasize the overall approach and experimental validation. However, the details are potentially of use to others who try to apply static analysis to complex code-bases. We therefore discuss them in this section.

Scale, alone, was not a major problem for our work. Even for the largest programs in our sample, run-times were less than an hour. Rather, the major problems we had to surmount were due to the presence of third-party libraries that were not of interest for analysis; due to reflection; due to the fact that we were analyzing frameworks and systems, not single programs. The combination of these issues was particularly challenging.

Above, we described Hadoop as containing a few hundred thousand lines of source code. However, at run-time, it is linked against many third party libraries. These account for ten times as much code as Hadoop itself (based on compiled size). Including this code would severely increase the run-time of our analysis. It would also degrade the accuracy: we are trying to help developers find and describe configuration options in their code, not in component libraries. As a result, we had strong incentives to treat library code and application code differently. This is a distinctive aspect of our approach — other analyses often do not impose any differentiation between library and application code. We suspect this is a tactic that can be re-used in other contexts.

Reflection is a mechanism by which programmers avoid exposing type information statically. Static analysis is therefore forced to make approximations. We did not invent any substantially new algorithms for handling reflection. However, existing approaches needed tweaking to handle our particular needs. As we mentioned above, our goal was to explore application code, not library code. This distinction is embedded deeply in our analysis: When we are finding possible types for reflectively-created objects, we only consider application types, not library types. We are unaware of a previous use of this heuristic.

Hadoop, or Pastry, or HBase, are not programs, strictly speaking. There is no `main` method at which they start. Rather, each of these projects has a code-base has several different entry points, more than a dozen for Hadoop and HBase. One approach would be to analyze each entry-point separately; effectively, this treats the software system as many separate programs that happen to share code. This approach is more precise, since it lets one distinguish the behavior of one

versus another portion of the complete system. We took this approach when analyzing Hadoop for commercial purposes at Cloudera.

It is also possible to do the opposite: to treat these separate processes as executing together with the same heap. There are advantages to this too: one can model interprocess communication very simply, by replacing remote procedure calls at analysis time with simple procedure calls. One avoids redundant computation re-analyzing the common portions of the code base that are invoked regardless of entry point. These advantages were significant enough that we primarily took this approach.

4.6 Discussion

In the previous sections, we described how to statically analyze programs to find the set of option names in use and the types of these options. We now discuss how general the problem and approach are. We also supply some insights from discussions with industry developers about why so many options are undocumented.

More careful programming with attention to configuration could reduce the mismatch between programs and documentation, and could catch more user configuration errors. Current versions of Derby and in-development versions of Hadoop attempt to confine option names to a small number of interfaces and classes. Our analysis could help enforce this property during development, since it is efficient enough to run every night alongside the regression test suite.

The most fundamental aspect of our approach is the observation that for named configuration options, the program, the programmer, and the user all use the same labels to refer to the same value, and these labels can be extracted automatically. Likewise, for many common data types, the semantics of the standard library are a close match for human understanding: our ad-hoc notion of a "class name" is effectively the same as the system library's notion.

4.6.1 Extensions and Future Work

We focused on key-value style configuration, but there are at least two other common styles for configuration management where this same observation applies. Many programs have graphical configuration management interfaces. Our techniques are potentially applicable to these programs; often, the graphical interface masks an underlying key-value model and not all options are exposed via the graphical interface.

Another common model for configuration is structured XML, where the program walks the DOM tree to retrieve values. An advantage of this style for programmers is that schema validation can catch a wide variety of user errors. The downside is that programming with this approach can be more cumbersome. Retrieving an option by name or an XPath query can be done a single line; walking the DOM tree cannot.

The techniques presented in this paper could be generalized to this alternate style. The limitation in generalizing our analysis is matching program points to the DOM nodes they retrieve. Other work has shown that static analysis can model the output (including output XML) from a program fragment [22, 41]; similar techniques may be applicable here.

We focused primarily on large highly configurable systems, with dozens or hundreds of options. Many programs, however, are small and have just a few named configuration options, commonly environment variables. The APIs to retrieve environment variables fall into the key-value pattern, which we have shown is easy to analyze. Hence, our technique would be useful to extract and model environment dependencies in smaller programs.

Our prototype implementation of our analysis was confined to Java. However, the key-value configuration model we focused on is also used in other contexts, notably the OS-defined configuration mechanisms in Unix and Windows (environment variables and registry keys, respectively).

Our analysis has a certain degree of inaccuracy: some options are not found at all, and others have types that are not found correctly. For finding options, we showed above that much of the inaccuracy in our approach was due to unsophisticated string analysis and path-insensitivity. This could potentially be fixed by a more sophisticated analysis, particularly if it were demand-driven and only needed to check small portions of the code.

For type inference, the prospects are substantially worse. Many types are distinguished only in ways that cannot be detected by a program analysis. For instance, the difference between a username and a password is that one is kept secret. For options whose types are potentially determinable statically, a third of all option values in our study are passed to other processes or libraries before being used, meaning that their types cannot be inferred by static analysis. For the remaining cases, integrating a string analysis into the dataflow step of our analysis is probably the most profitable enhancement. This might cover half of the non-numeric options. For numeric options, we suspect context-sensitive dataflow is the key optimization.

4.6.2 Why so many undocumented options?

Applying our work in practice at Cloudera gave us the opportunity to discuss our findings with members of the Hadoop development community. Their comments help explain the large numbers of undocumented options in Hadoop. We suspect that similar processes are at work in other software development projects.

Developers often use options as a sort of configurable named constant. Most complex programs, including Hadoop, are full of parameters that are theoretically tunable but where developers have no insight as to what the right value is. For example, the operating system offers many options controlling the details of TCP connection management. In most cases, there is no reason for a program to prefer one value over another. Offering this control to users would add more complexity and confusion than benefit; hence, the option is not documented. But it is conceivable that the configurability might be useful in some circumstance. Rather than compile in a default, developers use a configuration option. This way, in the event of a production failure, engineers reading the code can find and tune the options, without recompiling and redeploying. As one developer put it,

“anybody who hasn’t read that code won’t really understand what the option does, and therefore isn’t qualified to set it.”

Chapter 5

Explaining Configuration Errors

The previous chapter described how we can automatically extract a list of program configuration options and constraints on their values. This is potentially valuable for catching erroneous configuration before an error. This chapter addresses the problem of diagnosing errors, once they happen.

We address use this problem using broadly similar program analysis techniques to those presented before. However, failure diagnosis is a substantially harder problem and therefore substantially more sophisticated techniques are required. When inferring a type for an option, any analysis imprecision that causes two options of the same type to be aliased will be unnoticeable. In contrast, for failure diagnosis, this will result in visible inaccuracy. For type inference, the analysis can concentrate on the small portion of the program required to infer a type. With failure diagnosis, in contrast, the analysis must incorporate a much larger portion of the program since errors can manifest some distance away from where the option is read.

5.1 Motivation

Debugging configuration problems is a serious challenge. Consider Hadoop, for example. As we have noted, it is a large well-established project, with substantial developer resources. Despite this maturity, configuration error checking is still weak. If a user attempts to launch a recent widely-used version (0.20.2) of HDFS, the Hadoop file system, using the default out-of-the-box configuration, it will crash with a null pointer exception, affording users little guidance about what to fix. We will use this error as a running example.

This chapter describes a technique that can help users troubleshoot this sort of startup error. When a user faces an unhelpful error message, such as Hadoop's "NullPointerException at line 134 of NetUtils.java", our approach points them to a configuration option that, if changed, will make the error go away. Our goals are to give users an answer as quickly as possible, without users

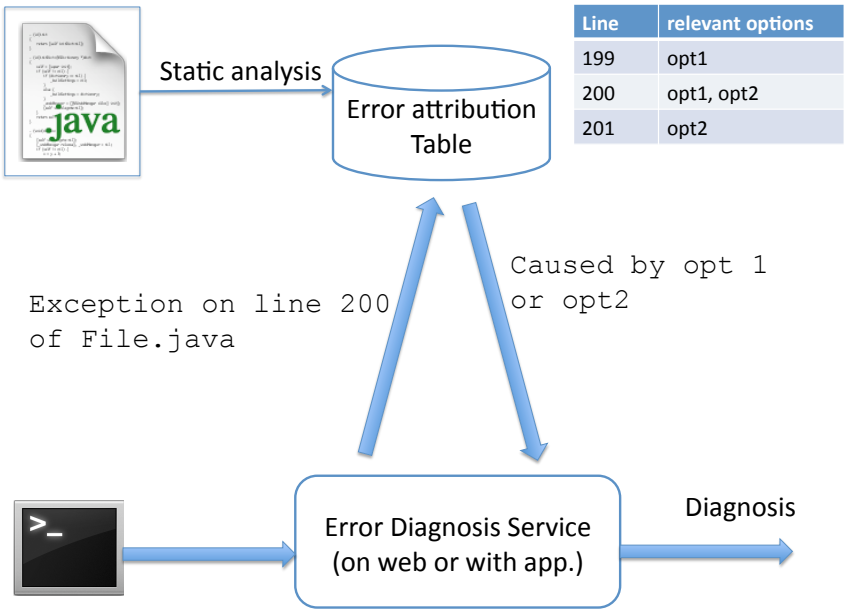

Figure 5.1. Illustration of overall structure of static precomputation approach to diagnosis. Upper portion of figure (thin arrows) is precomputed; lower part (thick arrows) happens in response to error.

installing any new debugging or analysis tools, without them having to understand the program's source code and without exposing sensitive site-specific configuration.

Most prior work on configuration debugging has relied on large user communities or on modifying the program's execution environment. Both of these are deployment challenges. In contrast, our approach lets developers shield users from the complexity of diagnosis.

We do not attempt to diagnose all possible misconfigurations. We target the case where the user has introduced a typo or nonsensical value for an option, rather than the case where some numeric parameter needs workload-dependent tuning. As we mentioned in Section 3.1, failures due to typos or invalid configuration values are very common. Hence, a technique that points to a specific "wrong" option value can potentially resolve a large fraction of real-world misconfiguration errors.

5.1.1 Our Contributions

The core of our approach is to analyze the program in question, producing a table mapping each line in the program's source code to the set of relevant configuration dependencies at that point. We envision this being done by the developers at release time. When a user encounters an error, they will input their error message, perhaps to a web service, and the response will be a list

of configuration options that are potentially to blame. Our approach requires no reconfiguration, new tools, or program modifications on the part of the user, unlike replay-based approaches or delta debugging. It requires no alterations to the JVM or standard library. This distinguishes our work from competing techniques such as dynamic taint tracking.

Previous work has shown how to map log messages back to the origin line in the source code [86, 83] and so we do not address this part of the process. We concentrate, instead, on the challenge of building the appropriate table mapping program points to possible misconfigurations.

In managed environments such as the Java runtime, un-handled errors often result in a stack trace. We show that these stack traces can significantly improve the precision of analysis. Our technique, which we call *failure-context-sensitive analysis* (FCS), re-analyzes the call chain corresponding to the stack trace, pruning out irrelevant paths. By reusing the results from a prior static analysis, the run time for FCS can be kept low. Effectively, FCS is a static analysis parameterized by a single concrete input, where much of the analysis can be shared across inputs.

For Hadoop, each FCS analysis takes approximately a minute with our current implementation. The results for each error can of course be cached, reducing the time to answer for subsequent user queries with the same stack trace.

As with the previous chapter, our implementation uses JChord and targets Java bytecode programs. We use the option-finding analysis presented there as a component for this work. We found, however, that substantially more complex points-to analysis was required to get good results. As a result, the fraction of code shared between the two approaches is relatively low.

5.1.2 Methodology and Organization

Our analysis is targeted to large complex software systems, such as Hadoop. In these systems, data will flow in and out of the system via the network and the filesystem. There may be native-language code. These data flows are difficult to capture dynamically, and even harder to model statically. As a result, we accept that our analysis will be imprecise and will miss some configuration dependencies. There will be both false positives and false negatives. We believe this is acceptable, so long as the analysis performs well in the common case, giving a correct diagnosis and not too many wrong guesses.

While our focus is on static analysis, we evaluate the benefits from several kinds of run-time instrumentation. This lets us gauge the sources of imprecision in our static approach. We show that tracking which options are read by the program can substantially improve analysis precision. This information can be recorded by the program and incorporated into the analysis cheaply. Only normal logging is required, not any sort of dynamic tracing or taint tracking.

The rest of this chapter is organized as follows. We begin by describing our model for configuration options and give an overview of the analysis techniques we propose. In Section 5.3, we present the details of our analyses. Our evaluation is in Section 5.4. Section 5.5 discusses limitations and sources of experimental error.

```
NetUtils.java:

SocketAddress getNameNodeAddress() {
60:   return createSAddr(
        getDefaultUri().getAuthority());
}
...
URI getDefaultUri() {
100:   return Conf.get("fs.default.name",
      "file:///"));
}
...
133: SocketAddress createSAddr(String t) {
134:     int colonIndex = t.indexOf(':');
135:     ....
```

Figure 5.2. Simplified Hadoop code that produces null pointer exception with default configuration

```
...
NetUtils.java 60 depends on fs.default.name
NetUtils.java 134 depends on fs.default.name
...
```

Figure 5.3. Analysis output for code in Figure 5.2. Note that reading a variable, as on line 60, is not a use of the variable.

5.2 Model and Overview

As in the previous chapter, we model configurations as a set of key-value pairs, where the keys are strings and the values have arbitrary type.

5.2.1 An Example

This section gives an example of how our analysis can diagnose a configuration error. We continue our running example: Hadoop 0.20, when started with default configuration, prints "NullPointerException at line 134 of NetUtils.java". Figure 5.2 is a simplified version of the code in question. The problem arises as follows. The `getNameNodeAddress` method attempts to construct an address from the authority (server) portion of the filesystem URI. The default filesystem URI (controlled by option `fs.default.name`) is `file:///`. This URI has no authority portion and so `URI.getAuthority()` returns null. This null then propagates to `createSAddr()`, which attempts to dereference it and then crashes. In the real implementa-

tion, this code is scattered across three different classes. Tracking down the problem in the source code would be a substantial task, particularly for new users.

Our approach builds a table mapping each line in the program to the options most directly associated with it. In this case, there will be an entry saying that `fs.default.name` affects line 134 of NetUtils.java. This table can then be presented to users via a web service (or a general-purpose search engine), letting them discover that `fs.default.name` is a relevant option at the point where the exception was raised. Our analysis does not tell users what value for the option will resolve the problem, but it can avoid wasted time tinkering with irrelevant options.

Before describing our analysis in detail, we give a sketch of how our analysis ties together the option and the relevant source code line. This is intended to give the overall flavor of the approach. The analysis marks the call to `Conf.get` as an option read. Its return value is therefore assigned the label `fs.default.name`. Dataflow analysis tracks the flow of this label into `createSAddr`. Line 134 of NetUtils.java uses this value, and so the analysis outputs that the line in question depends on `fs.default.name`. Figure 5.3 depicts this. Note tha *reading* a configuration option, as happens on line 100, is not a *use* of the option.

5.2.2 Overview of Approach

The next section will discuss our analysis algorithms in detail. Here, we give an overview of the approach. We define a label for each configuration option. We then use dataflow analysis to determine which values and program points are associated with each label. This analysis happens at the bytecode level. At the end, we map these results back to line numbers. Since the analysis is being done at development time, we assume that debugging information is available to supply the line numbering. These error-attribution maps are small enough to easily fit in memory, even for large programs, as discussed in Section 5.4.3.

The steps in our analysis are listed below. The first step is a standard context-sensitive points-to analysis. The second step is described in the previous chapter. The next three steps are responsible for mapping program points to configuration dependencies, and are the contribution of this chapter. Below is an outline of the approach; we also depict the approach schematically in Figure 5.4.

1. Points-to analysis and call-graph construction.
2. Find configuration read points and associated names.
3. Dataflow analysis of configuration labels.
4. Optional: Filter data-flow using failure-context inferred from stack trace.
5. Method-local control-flow analysis using results of
   either whole-program or failure-context-sensitive dataflow analysis.

The basic analysis can all be computed statically and shared across all configuration errors to be diagnosed. Only the (optional) failure-context-sensitive analysis needs to be repeated for each distinct error message.

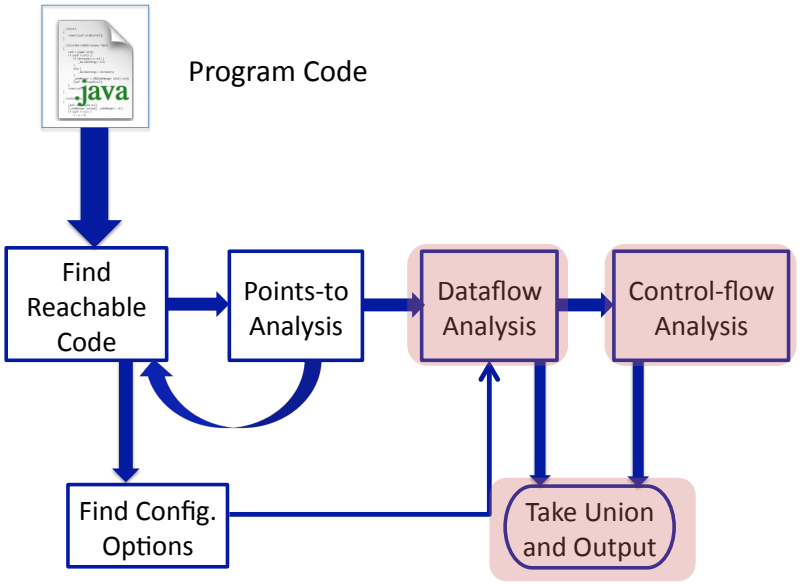

Figure 5.4. Schematic of algorithm for static analysis to pre-compute configuration dependencies. Highlighted parts are discussed in this chapter.

Our approach is similar to taint tracking in that we are concerned with tracking the flow of labels through a program. Labels are introduced via configuration reads, and propagate via assignment and via library calls. Unlike taint tracking, we are not trying to find all possible dependencies, but only the most relevant ones for troubleshooting. To avoid the well-known problem of taint explosion [69], we apply a number of heuristics, discussed below. Our analysis can also also be thought of as an application of thin slicing [73]. We are effectively computing a forward (thin) slice from each configuration option read, and outputting the set of slices that each program point belongs to.

5.3 Implementation

This section describes our analysis in more detail.

We first describe the overall strategy for our static analysis, followed by the details of the approach, followed by our failure-context-sensitive technique and then, last, the dynamic analyses we used to evaluate sources of imprecision.

```
public static void main(String[] args) {
   String world = System.getenv("world");
   System.out.println("Hello, " + world);
}
```

Figure 5.5. A small configurable program

5.3.1 Static Analysis: overview

There is a large set of possible analysis algorithms along the lines described above. Two design choices were especially significant for us: Taint labels can apply to objects or else to references. Library code can be analyzed directly or replaced with a simple model. The two design choices are coupled. Table 5.1 summarizes the relevant design tradeoffs.

We chose the first approach in the table. This is unusual in program analysis but follows the example set by RacerX [25]. As we discuss below, not analyzing the library was a major gain in both analysis accuracy and speed. Applying labels to references, not objects, works better in the case where much code is un-analyzed. This decision also lets us treat primitive types (such as integers) identically to reference types (such as Strings). The price is that this analysis is unsound. However, it works well in practice and we therefore adopt it. We discuss the reasons for this success in the next subsection.

Approaches 2 and 4, which involve analyzing the library, we ruled out experimentally. Consider the code in Figure 5.5. This is virtually the smallest possible configurable program. Using Approach 1, this ran in 30 seconds. Analyzing the library ballooned the runtime by a factor of six. Most of the run-time is used up in the points-to phase of our analysis, which has to be conducted for either Approach 2 or Approach 4.

Results on larger programs were even more discouraging. The run-time on Cassandra went from 4 minutes to 210 – a factor of 50. For the components of Hadoop, the analysis ran for a week on an Amazon EC2 large instance and then crashed. We tried again, reducing the context-sensitivity from $k = 2$ to $k = 1$. This allowed the analysis to finish in a few hours (2.5 hours, for the Datanode, which is the smallest portion.) The average number of false positives went up from 4 to 20.

The bulk of this run-time was spent in building the points-to model for the program, including libraries. Unfortunately, this work cannot be cached and amortized across different analysis targets. The points-to model of the library will differ, depending on which portions of the library a program uses. As a result, this immense overhead must be paid each time.

The fundamental problem posed by library code is aliasing between calling contexts. Library code is often called from many unrelated locations in a program. If analysis is context-insensitive, values will flow in from one context and out through another, resulting in spurious dependencies. Adding higher levels of context-sensitivity would reduce the imprecision, at the cost of still-higher run-time. This cost was expensive enough to justify the modeling work we used to avoid it.

Approach 3 is unworkable theoretically. Using a model for the library implies that there will

	Library	Labels on	Pros	Cons
1	Modeled	References	Good performance, both in runtime and accuracy	Unsound. Some special cases in models.
2	Analyzed	References	No need to define library model	Performance is unacceptable
3	Modeled	Objects		Unworkable because many objects have no allocation site
4	Analyzed	Objects	Handles aliasing directly.	Native code and primitive values as special case, potentially poor performance

Table 5.1. Overview of high-level design tradeoffs

not be an allocation site for references returned by library methods. As a result, Option 1 was the route we took.

5.3.2 Static Analysis: details

Points-to We used the k-object-sensitive points-to analysis [56] built into JChord. The analysis is field-sensitive, path insensitive, and flow insensitive. It uses the static single assignment form of the program to gain some of the benefit of flow sensitivity (as suggested by Hasti and Horwitz [31]). We handle reflection using the technique presented in [51].

Many of the programs we analyzed make significant use of remote proceedure calls (RPCs). Labelled values should flow from the initialization code, invoked from main, to RPC methods, invoked remotely. This requires that the same abstract object be used in each context. To incorporate remotely-invoked methods into our points-to analysis, we automatically generate Java stubs that call each remotely accessible method of a server object. We then hand-wrote a few more lines of Java to ensure that these remote methods were invoked on the server object created by `main`.

This is a departure from the approach in the previous chapter, where we simply marked every public method of each remotely-callable class as "reachable." That approach proved hard to generalize to the context-sensitive case. It isn't enough for the method to be reachable; it must be reachable in some particular context(s). Picking out the appropriate contexts within our analysis proved troublesome. The stub code approach avoided these problems.

Finding configuration We find configuration options using the approach described in the previous chapter. To review, briefly:

Each of the applications we examined used a handful of "configuration" classes (often with that name) responsible for reading a configuration file and exposing a key-value interface to the rest of the program. We manually construct a list of "configuration methods", including both API methods like `getenv` and their application-defined equivalents. For the programs in our study, this required a handful of per-application definitions. This list also records which argument corresponds to the

option name. Configuration is typically spread across several domains: an environment variable named `someOption` may have no connection to a JVM property of the same name. To remove this ambiguity, we associate each configuration method with the name of the relevant domain.

At each call site that reaches one of these methods, a string analysis determines the name of the option being read, or "Unknown" if the analysis cannot find it. We used a custom-written string analysis that integrated easily with our points-to analysis. In prior work we showed that this technique finds options with over 95% accuracy on large complex programs, including the ones analyzed in this paper, making a more sophisticated string analysis unnecessary for us. More sophisticated string analyses (such as JSA [18]) could be used without changing our overall approach to finding configuration read points.

Dataflow We found that our results were not extremely sensitive to the details of the analysis. We briefly summarize the implementation choices we made, but due to space limitations, we do not give detailed comparisons.

The biggest benefit came from using object-sensitive analysis. This decreased false positives by 40% compared to context-insensitive analysis.

We mark a method's return value as depending on an argument if there is a control-dependency between the return statement and the argument. This rule is not strictly necessary; adding it increased coverage slightly (two additional true dependencies caught) while decreasing precision by approximately 10%.

If two variables may be aliased locally, we propagate labels from one to the other. Likewise, if a local variable aliases a static or instance field, and that variable becomes tainted via a library call, we taint the field. In the interests of precision and run-time performance, we do not model arbitrary interprocedural aliasing. Most important Java library-defined types (including files and strings) are immutable. This means that even if the object is aliased, labels cannot flow between contexts: all the labels for an immutable object must appear in the context where the object is created.

Library modeling We adopt an approximate and pessimistic model for library code. We were driven to this model by the need to accommodate native code. We then found that it was practical to use for all library code, reducing the size of the code that needs to be analyzed and gaining the effect of an extra level of context-sensitivity for library calls.

We apply this model to the Java run-time libraries as well as to libraries used by the applications in our study. As a special case, we mark the Path types in Ant and Hadoop as library code. These types are returned by configuration-read methods and we need to treat them as opaque to maintain our invariant that labels apply only to primitive types and to references to library types.

We assume that if a parameter to an API call depends on a configuration option, then the object on which it is invoked and the return value both also depend on that option. As a special case, we do not mark input or output stream objects as depending on the data being written or read. We distinguish a handful of methods such as `equals`, where the arguments influence the return value but do not alter the object on which they are invoked.

Collection classes (such as arrays, hash tables, and so on) are mutable, and so our naive library

model was inappropriate here. Our response was to use a simplified model implementation in Java. (Simply analyzing the library directly would not have captured the implicit dependency between the argument to `get` and its return value.)

Control-dependence We follow our dataflow analysis with a static and method-local control-flow analysis. Our primary concern was to capture the common programming idiom of checking a value and then emitting an error message if something is amiss. The analysis marks a program point as depending on an option if that point is on one side of a branch, where the branch condition has a data dependence on the option. Previous work on thin slicing has demonstrated that this sort of method-local analysis is likely to capture most relevant control dependencies while including few extraneous ones [73]. We have explored alternative slicing approaches that included additional implicit flows; these caused only small changes to our results.

The final output at each program point is the union of control and data dependencies at that point.

5.3.3 Failure-context-sensitive analysis

In managed languages like Java or C#, a crash often produces a stack trace, not just a simple error message. The approach we described above, however, only uses the top line of the stack trace, the point where an exception was raised. In this section, we describe how to exploit the full stack trace to get additional precision. We refer to our approach as *failure-context-sensitive* (FCS) analysis.

Values can flow between calling contexts, via the heap. FCS therefore requires having a whole-program analysis in the background. Our static analysis gives us a set of labels for each abstract heap location. It also gives us a summary, for each method, of how the method's return values depend on the arguments.

The FCS analysis runs after our context-sensitive dataflow analysis and before the control-flow analysis. We use FCS to filter the results of the whole-program dataflow to the particular context containing the crash.

We model a stack trace as a sequence of method calls $m_0 m_1 m_2 \ldots m_N$, where m_0 is the first-called method (farthest down the call stack) and m_N is the method containing the point where the program crashes. We extract this sequence from the text of the stack trace. Separately, our whole-program points-to analysis gives us a set of tuples (c_1, m_1, c_2, m_2), such that M_1 in context c_1 can potentially call M_2 in context c_2. We call this the context-sensitive call graph.

Failure-context-sensitive analysis joins together the stack trace with this set of calling contexts to produce the set of contexts in which the failure could have occurred. We begin by defining the *failure path*. If m_0 is the first method in the stack trace, then let C_0 be set of contexts in which m_0 is potentially called. Inductively define C_k as the set of contexts such that $(c_{k-1}, m_{k-1}, c_k, m_k)$. At the end, C_n is the set of contexts in which m_n, the failing method, can be reached via the call chain defined in the stack trace. Call this the set of failure contexts.

Once we have the set of failure contexts, we use them to filter the set of data dependences picked up by the analysis. We only output the configuration options that can cause a data or control dependence in a failure context, whereas our basic static analysis includes dependencies in any possible context.

A key point about FCS is that the runtime does not depend directly on the size of the program. The run-time is proportional to the depth of the stack trace and to the number of contexts in which each method can be reached. The stack trace can be pruned down to a fixed size to produce more predictable run-times.

5.3.4 Dynamic Approaches

The static analysis above has several sources of imprecision. To gauge their importance, we replaced portions of our static analysis with instrumentation-based dynamic analysis.

The value of a configuration option that is never read cannot affect the program. Our first dynamic approach, *instrumented configuration reads*, records which options are read and where. This is then consumed by the static analysis discussed above. Effectively, this is static analysis using only the options actually read by the program.

Our second dynamic approach, *instrumented configuration flow*, goes farther. In addition to dynamically monitoring option reads, we track the flow of labelled objects through the program dynamically. When a configuration value is returned by a method, that value and the associated option name are recorded in a lookup table. This table is kept in the memory of the instrumented process. At each new invocation, the options associated with each parameter are written to disk. Unlike our static analysis, this approach tracks objects, not values. We augment the dynamic tracking with a method-local static analysis to track primitive types and the flow of values that were null at run-time. We reuse our static control-dependence analysis.

Instrumented configuration flow was intended purely to measure sources of imprecision. In contrast, instrumented configuration reads could be used in practice to improve the quality of diagnosis results. In the programs we have seen, configuration data is accessed exclusively via a handful of program classes. The value of each option could be logged the first time it is read. The volume of information would be small and proportional to the number of configuration options. This information could be extracted from log files during troubleshooting and used to filter the static analysis results. This logging would also compensate for imprecision in finding option names by making the concrete run-time names available.

5.4 Evaluation

In this section, we evaluate the static analysis approaches we presented above. We seek to answer several research questions: How complete and correct are the results of the analysis? What are the sources of imprecision? How much gain is there from failure-context-sensitive analysis?

Table 5.2. Results using ConfErr for Fault Injection: coverage is high and false positives are low.

Target	Errors Injected	Avg False Positives	False Neg.	Success rate
Hadoop HDFS	5	1.0	1	80 %
Hadoop MapRed	6	3.5	1	83 %
JChord	7	2.1	1	85 %

We performed two different sets of experiments. First, we performed a fault injection study using two programs, Hadoop and JChord. This experiment measures how well our static analysis works, how much gain there is from failure-context-sensitive analysis, and how much different sorts of dynamically-collected information improve precision. Our second set of experiments controls for selection bias in the injected faults and the programs we investigated. We measured the statistical properties of our analysis on a range of additional programs and program points. We demonstrate that the programs and program points evaluated in the first set are not anomalous, evidence that our technique is more broadly applicable.

5.4.1 Catching Injected Configuration Errors

Our fault injection experiments focused on two software systems, Hadoop and JChord. The Hadoop source code consists of several hundred thousand lines of Java, and over 20 library dependencies. Its functionality is split across several separate but communicating programs. It makes heavy use of reflection, making it a suitably "hard target" for program analysis. JChord is a program analysis tool and deadlock detector. It has five developers, approximately 30,000 lines of Java source code and a dozen library dependencies. It too makes heavy use of reflection. Both programs support several dozen options.

We used the ConfErr tool [39] to insert typographic errors into otherwise-working configurations for each of JChord, Hadoop MapReduce, and Hadoop's HDFS filesystem. Some typographic errors are masked silently by the program. Each of the remaining erroneous configurations led to either an error message or a stack trace. If there was an error message, we check for dependencies at the line where the message was printed. For stack traces, we use the first point on the trace with a dependence. (This rule covers cases where an exception is raised inside unanalyzed library code.) We analyze the results with our basic static analysis.

Our results are displayed in Table 5.2. ConfErr found 18 errors across the three systems under test. Of these, our tool diagnosed all but three, a 17% false negative rate. (The false negative rate is the fraction of errors for which our algorithm did not find the true injected root cause.) Once each for Hadoop MapReduce and HDFS, an injected string broke the XML format of the configuration file. This caused an error at configuration read time. Our tool was unable to diagnose these errors, because they were not tied to any particular option. The un-diagnosed JChord error was caused because the bad value was used to set up the command line for a child process; our analysis

Table 5.3. Detailed results for Hadoop. Shows count of false positives. "N" = technique failed. "X" = inapplicable (no stack trace). Average false positives for each technique is key figure of merit.

Run ID	Program	Error	Static	Dynamically-measured		FCS		Last
				Reads	Flow		+ Dyn. Reads	Load
1	TaskTracker	rpc socket factory class not a class	0	0	0	X 0	X 0	Y
2	TaskTracker	master hostname not set	7	0	0	6	0	Y
* 3	TaskTracker	child work dir not writable	N 0	N 0	N 0	N 0	N 0	N
* 4	NameNode	storage dir not writable	3	1	0	3	1	N
5	NameNode	fs.default.name not this machine	0	0	0	0	0	N
* 6	NameNode	fs.default.name not HDFS	5	0	0	0	0	Y
7	NameNode	Data port unavailable	1	1	0	1	1	N
8	NameNode	Info. port in use	2	2	0	2	2	N
9	NameNode	invalid topology mapping class	0	0	0	0	0	Y
10	NameNode	topology mapping script not valid	4	4	1	4	3	N
11	NameNode	FS object quota exceeded	0	0	0	0	0	N
12	NameNode	malformed XML	N 0	N 0	N 0	X 0	X 0	N
13	JobTracker	jobtracker info port in use	2	2	0	2	2	N
14	JobTracker	storage dir not writable	8	3	1	8	2	N
* 15	JobTracker	master hostname not set	1	0	0	0	0	Y
16	JobTracker	log dir not set	0	0	0	X 0	X 0	Y
* 17	JobTracker	carriage return at end of address	1	0	0	0	0	Y
18	JobTracker	malformed XML	N 0	N 0	N 0	X 0	X 0	N
19	DataNode	missing DN port number	8	3	1	7	2	Y
20	DataNode	use of deprecated option	0	0	0	X 0	X 0	Y
* 21	DataNode	master host not specified	8	0	0	0	0	Y
22	DataNode	storage dir not writable	0	0	0	X 0	X 0	Y
Success %			86.4	86.4	86.4	86.4	86.4	50
Average False Pos			**2.3**	**0.7**	**0.1**	**1.5**	**0.6**	

does not capture dependencies between command line arguments and configuration options. We compute the false positive rate by counting the number of configuration dependencies other than the one with the injected error. This averaged between 1 and 3.5 for the systems in question. (The ideal is 0, meaning exactly one diagnosis for every error.) Our technique thus succeeds in drawing attention to a handful of possible problem diagnoses.

Next, we examine precision more closely by comparing five different analysis techniques: static analysis, the two dynamic approaches discussed above, failure-context-sensitive static analysis, and failure-context-sensitive analysis using only options read at runtime. For errors without stack traces, FCS is equivalent to static analysis. We mark these cases with an X in our data tables and reuse the result of the static analysis. All our techniques had the same false negative rates, so average false positives is the relevant figure of merit.

We felt that the errors found by automated fault injection were not necessarily representative of the full range of user mistakes. Errors such as an unavailable port number for a server or insufficient

Table 5.4. Detailed results for JChord

Run ID	Program	Error	Static	Dynamically-measured		FCS		Last
				Reads	Flow		+ Dyn. Reads	Load
1	JChord	no main class	1	1	0	1	1	N
2	JChord	no main method	0	0	0	X 0	X 0	N
3	JChord	no such analysis	3	0	0	X 3	X 0	N
4	JChord	invalid context-sensitive analysis name	1	1	2	1	1	N
5	JChord	printing nonexistent relation	0	0	0	0	0	N
6	JChord	disassembling nonexistent class	0	0	0	X 0	X 0	N
7	JChord	invalid scope kind	4	2	0	X 4	X 2	N
8	JChord	invalid reflection kind	4	2	2	X 4	X 2	N
9	JChord	wrong classpath	N 2	N 2	N 1	X 2	X 2	N
Success %			88.9	88.9	88.9	88.9	88.9	0
Average False Pos			**1.7**	**0.9**	**0.6**	**1.7**	**0.9**	

disk space show up in operation but not in automated testing. For Hadoop, we found a number of mailing list messages with errors, configuration, and a confirmed diagnosis. We incorporated those into our test inputs to provide a wider range of test cases. They are marked with asterisks in Table 5.3. For JChord, we reused the errors found by automated testing. Our results are presented in Tables 5.3 for Hadoop and 5.4 for JChord. We display these results graphically in Figure 5.6.

For both Hadoop and JChord, the biggest precision gain came from dynamically recording which options are read (the *instrumented reads* approach). Dynamically tracking values (*instrumented flow*) adds additional benefit. The gap between instrumented reads and instrumented flow is a measure of how much imprecision comes from our dataflow and points-to analyses. As can be seen, there is a gap, but a comparatively smaller one.

For Hadoop, in the cases where static analysis finds many possible options, failure-context-sensitivity improves precision substantially, reducing the average number of guesses by approximately a third. Failure-context-sensitivity plus restricting to options read at runtime reduced false positives by nearly a factor of four, as compared to pure static analysis. For JChord, FCS was ineffective; JChord errors largely lacked stack traces, and so the technique does not apply.

A few aspects of our measurements require explanation. Hadoop Test 3 is a real user-reported error that manifests in a subprocess spawned by the Hadoop TaskTracker process. As mentioned, our analysis does not track this type of dependency. Instrumentation-based approaches can sometimes find more options than were found statically. Our static analysis combines related option names. At run-time, `fs.hdfs.impl` and `fs.file.impl` are distinct, but our static analysis treats the two as one option, `fs..*.impl`. Tracking objects can also introduce spurious option dependencies not found statically; we observed this in our fourth JChord test case.

We also tried a simple heuristic, *last-option-read*. This heuristic looks at the last configuration option that a program read before failing, but does not examine the program structure in any way. We had expected this to work because many wrongly-set options will result in errors on first use, if they will result in errors at any point. Experimentally, we see that this heuristic works half the

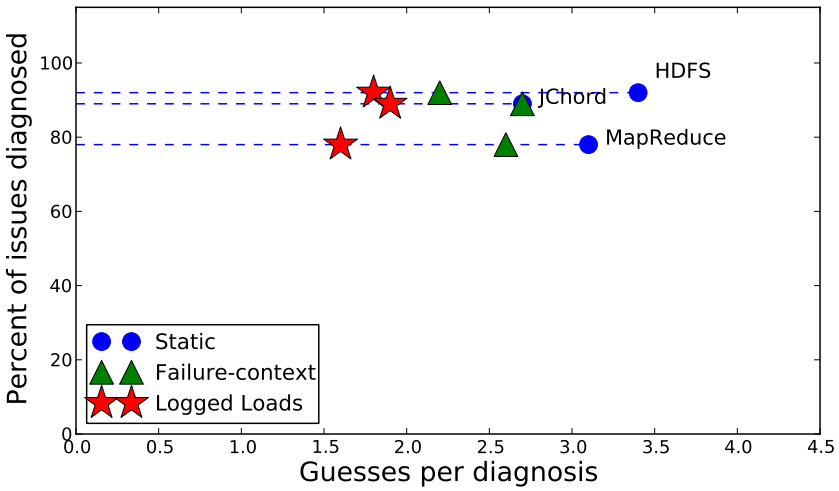

Figure 5.6. Precision-recall plot for error explanation

time for Hadoop: Hadoop often reads options immediately before using them; if the first use of an option value triggers an exception, then that option was likely the most recent one read. The heuristic does not work at all for JChord. JChord reads most of its options at program startup, so there is no close connection between reads and uses. We conclude that this heuristic relies entirely on particular styles of programming for its effectiveness.

5.4.2 Measurements From Other Programs

In the previous section, we only looked at configuration errors in two software systems, Hadoop and JChord. To estimate how well our approach would work on other errors and other programs, we compare aggregate statistics from these two programs to those from several others. (We profiled these projects in Section 2.1.)

To estimate precision, we look at the average number of exceptions at method call points. (Non-method call statements cannot print error messages and can only raise exceptions in a few circumstances.) We display our results in Table 5.5. We separately display the average number of dependencies at points with at least one.

Most program points have no configuration dependency. Of points with a dependency, the average number at any point is less than eight for all the programs in our sample, and less than six for all but two programs. As can be seen, Hadoop and JChord (the programs examined in detail above) are not radically different from the other programs we examine here. This suggests that our failure injection results are generalizable to other programs. Moreover, the average number of dependencies per program point found here is similar to the average number of false positives measured in the failure injection experiments. This further validates our methodology, showing

that the program points at which errors arose in the experiments above are comparable to the average for each program as a whole. (Note the average number of false positives is lower than the average number of dependencies: one option is actually responsible for a given error in our tests.)

In some programs, log messages explicitly mention configuration options. Programmers may print the value of an option alongside its name or produce error messages with option names. These messages are effectively labelled data, and give us an additional avenue for estimating the false negative rate for our analysis: find messages that mention options, and check whether the analysis finds a dependency between that message and the named option. The results of this analysis are also displayed in Table 5.5. All but a handful of these "explicit dependencies" are detected by our technique. We saw two causes for false negatives. First, our analysis does not track control flow via exceptions. Second, some messages mention an option that does not actually influence the message, such as "option X is deprecated, use Y instead." This latter case is a limitation of our evaluation technique, not of the underlying program analysis.

Table 5.5. Measurements showing analysis coverage. Table shows number of method call sites observed by analysis, total options for program, options per call site, and options per call site at sites with at least one.

Program	Analysis time (min:sec)	options	method calls	calls with dependence	avg deps per call	avg deps/call if > 0	opt mentions in logs	detected mentions
Ant	28:08	73	34071	6579	1.4	7.2	14	11
Cassandra	3:02	50	3693	667	0.7	4.1	0	0
FreePastry	23:50	119	24193	6221	1.3	4.9	0	0
HBase	7:15	140	15122	3815	1.3	5.2	3	3
Hadoop DataNode	3:35	72	7258	1766	0.7	2.9	12	11
Hadoop JobTracker	4:04	126	8939	2281	1	3.9	9	7
Hadoop NameNode	4:20	112	11739	3161	0.7	2.7	5	5
Hadoop TaskTracker	4:58	113	8112	3170	2.3	5.8	12	12
JChord	2:20	90	6873	2234	0.8	2.4	63	61

5.4.3 Performance Aspects

We next consider the running time and output size of our analysis. Time costs for all the programs we studied are displayed in Table 5.5. Measurements were conducted on a modern dual-core laptop with 4 GB of RAM. The largest program and longest-running analysis was Ant, which took just under half an hour. Most others were under 10 minutes. This is acceptable as part of a nightly build process.

Our prototype outputs flat, uncompressed, text, with one line for each (line,option) dependency pair. Each entry is approximately 150 bytes of uncompressed ASCII. Even with this space-inefficient format, the largest table we have seen, for Ant, is 3MB. This can be kept in memory and searched very quickly, taking less than a second. A more space-efficient output format could reduce space and time cost substantially.

Unlike our base static analysis, failure-context-sensitive analysis has a measurable run-time

per error. For Hadoop and JChord, the average FCS run time was approximately 10 seconds. For JChord, FCS took approximately 5[1]

5.5 Discussion

This section discusses the limitations of our work. We start with experimental flaws and broaden our scope to limitations of the technique and the underlying model.

There are programming practices that would reduce the effectiveness of our analysis technique. In particular, our techniques would produce useless results on programs that catch all exceptions and emit only a general error message. Our log-based experiments (though not our technique) implicitly assume that messages that mention options are representative of program points that users care about.

Our technique focuses exclusively on configuration errors where the value of an option is wrong and this causes a program to fail in a deterministic way with an error message. This category represents many but not all, real misconfigurations [84, 14]. The injected errors we used to evaluate our analysis are intended as representative examples.

We focus exclusively on named configuration options with a key-value semantic. This model is common, but not universal. It has two properties that we rely on. The first is that named options are associated nearly one-to-one with points where the program reads the option, and this mapping can be found automatically. The second is that options are not arbitrary data. In the previous chapter, we showed that a large majority of options fall into one of three categories: numerical parameters (such as timers and buffer sizes), named modes of operation (such as Boolean switches that enable and disable features), and names of system-level entities (such as network addresses and file names). Values of these types are used in narrower ways than arbitrarily chosen program inputs would be, including those inputs that might be viewed as configuration under a broader definition. For example, our approach would not help pinpoint which line of a malformed program caused a compiler to abort.

The work we presented here describes the back-end algorithms for a configuration diagnosis system. We have not discussed what the front end might look like. A key step, in deploying this work in practice, will be for the diagnosis service to map error messages to program points. This is straightforward if the error message includes a stack trace. It is not quite so simple otherwise.

One difficulty, which has not been mentioned in past work, is that log messages can be ambiguous. There may be multiple logging statements that produce similar log messages. As a result, the error message may not actually identify a unique program point. One approach would be to take the union of all possible points where the message could have been printed. Another approach is to revise the program to remove the ambiguity. The next chapter of this dissertation describes an automated solution for accomplishing this.

[1] Because of limitations of the JChord implementation, parts of the static analysis need to be repeated for each run, but this is not inherent to our algorithm; these time costs are not included in the measurements here.

Chapter 6

Improving Program Logs

The previous two chapters described a suite of static analysis algorithms for managing and debugging program configuration. This chapter offers a different application of program analysis to operational challenges. Here, we describe how program transformation can improve the quality of console logs from legacy codebases. This transformation has two benefits. It makes logging easier to configure, offering finer-grained control than was previously available. It also makes it easier for automated tools to parse logs, addressing a difficulty we mentioned in Chapter 5. As in the previous chapters, our focus here is on practical, scalable techniques that work well with realistic systems.

6.1 Introduction

Many programs produce semi-structured textual logs ("console logs") for debugging and administration purposes. As we noted, developers routinely add logging statements to their program as a debugging technique [83]. Logging statements often remain in programs after they are deployed in production. The emitted logs are then used to debug operational problems, particularly those that are challenging to reproduce or localize.

However, conventional logging has limitations that make it less useful than it could be. Finding actionable insight from systems logs is challenging. Finding consistently useful ways to analyze logs is an open research problem [58].

As we discussed in Section 3.5, there have been extensive efforts to replace ad-hoc textual logging with more structured techniques. However, such approaches are comparatively challenging to adopt. Logging is a cross-cutting concern, and this poses a challenge for some contemporary development techniques. In particular, it is a problem for open-source projects where changes are more easily adopted if small [38].

In contrast, conventional logs are simple to understand, simple to use, and can be modified incrementally. Sometimes, logs are the only debugging tool present in a system. Production sites routinely have extensive infrastructure in place for collecting and managing these logging volumes. Existing systems include Facebook's Scribe project [4], Chukwa [12] at Yahoo!, and of course, the venerable and ubiquitous `syslog` [52]. Replacing logging outright has a large fixed cost; improving logging in ways that leverage these assets is far easier to deploy.

6.1.1 Problems with Logging Today

We believe that many of the problems with logging today are due to users having insufficiently precise control over the behavior of the logging libraries. (When we say "users", we mean the users of the resultant logs. These may be administrators, developers engaged in debugging, or perhaps others, such as authors of log analysis tools). Whereas today logging is typically controlled at the granularity of files, we believe logging libraries should offer fine-grained control. It should be possible for users to enable or disable individual logging statements and to label statements with customized audience or severity indicators. Offering this control requires that every logging statement have a distinct identity, so users can specify which statement is being configured.

The remainder of this section will explain which problems with logging today we think would be solved by fine-grained control and statement identifiers.

Message grouping Statistical analysis of system logs has become a significant research area. Analysis often requires grouping messages based on the log statement that printed them. Many different machine learning techniques have been proposed for this task [7, 74, 37, 54, 28, 26]. Xu et al. [83] used program analysis to generate parsers for matching particular statements. The substantial body of work on this topic suggests that there is a perceived need for such grouping but that no simple, robust, and precise technique has yet emerged . Existing techniques all have some degree of imprecision. Many are computationally intensive.

Message Volume The volume of logging from a system can become excessive. Chapter 2 described a study of approximately 300 Hadoop trouble tickets at Cloudera. In that study, we found four tickets caused by excess logging. In each case, a system got stuck in a failure state, repeatedly printing the same warning message, without leaving the failure state. This resulted in a single machine generating tens of gigabytes of logs per day. Even in an era of large hard disks, this rate can cause operational problems, particularly at sites with lax log monitoring or limited storage. Dealing with these issues required many hours of supporter time. We have been informed anecdotally that similar problems appear in other software systems.

The remedy offered by logging libraries is to increase the threshold severity at which log messages are recorded. Different sets of messages can have different thresholds. However these sets themselves are fixed at compile time: a standard practice is for severity to be controllable at the granularity of source files.

This solution is unsatisfactory. Excess logging indicates a real operational event. Administrators should see that the event is happening, but have this expressed in a few messages, not gigabytes per day. Increasing the logging threshold is all-or-nothing. Further, file-granularity requires turning off other unrelated messages to disable the excessive ones. The price for silencing excessive noise is losing unrelated useful signal in the logs.

Priorities are Fixed Too Soon Oliner and Stearly studied `syslog` output from several supercomputer deployments [59]. They observed that the severity field in a message does not reliably predict the message's importance to administrators. A routine event might be marked FATAL, while a genuinely ominous indicator might merely be "INFO" [58].

This is partly a consequence of the design of the logging API. The severity of a message is encoded in the logging function called to generate it — `info()`, `warn()`, and so forth. Administrators can change the severity threshold, thus receiving more or fewer log messages from that logger – but cannot change the developer-specified relative priority of two messages without recompiling. Without statement numbering, there is no straightforward way for users to indicate which statement's priority they seek to change.

A related drawback of today's log severity abstraction is that programmers must pick the severity of a message without knowing the audience. There is no way to express that a message is safe for users to ignore but important to a programmer engaged in debugging.

Why source identifiers are insufficient It is tempting to use the source code file and line number of each log statement to label the resultant messages. This is fine-grained, however, it is too brittle. Line numbers can become invalid or ambiguous if the code is changed. If logging were configured by reference to line numbers in the code, then adding a one-line comment to a source file could break existing configuration files for both the logger itself and any downstream analyses. To simplify configuration of the logger and to ease analysis of the resultant logs, statement identifiers should remain stable from version to version, even if the log statements in question are modified slightly. Without consistency, users would need to re-configure their systems after every patch.

6.1.2 Contributions

The problems described above would be simpler to solve if loggers offered fine-grained control over which messages are printed and how they are labelled. This chapter shows how to achieve this control, even for legacy software.

In our approach, statement numbers are inserted into each log message. Suppose some message, perhaps "File Not Found", is printing excessively. With our revised logging system, the message will be printed with a number, perhaps "[123] File Not Found". To disable that message, administrators can send a UDP packet to the process or edit a configuration file, supplying that number.

We make three contributions:

1. In the next section, we point out a problem with current approaches that try to retrospectively match log messages to statements. We show that these approaches are intrinsically less powerful than those that involve program modification or a new API. While the gap may be small in most cases, it is unnecessary: the approach we present is not subject to this bound.

2. In Section 6.3, we describe an API and architecture for logging that offers fine-grained control while maintaining consistent statement numbering across different program versions. We achieve consistency by separating the user-facing statement numbers from a canonical internal representation. An explicit set of tables maps between these representations. A strength of this approach is that cross-version matching is independent of the rest of the system. Arbitrarily sophisticated algorithms can be used if need be. However, we have found that a simple greedy algorithm works well in practice.

3. We present an implementation of this architecture and demonstrate that we can retrofit our revised logging to legacy Java programs without disrupting the user experience or imposing significant performance consequences. Our implementation is called *relogger*, the **R**etroactive **E**nhancement logger. Given a compiled Java program, `relogger` imposes a global numbering on all log statements using load-time program weaving. This weaving is equivalent to what AspectJ or another aspect-oriented system would do, however our implementation is standalone for reasons we will explain. Figure 6.1 illustrates our approach.

   The implementation is presented in Section 6.4; Section 6.5 evaluates the implementation.

While our focus is on Java (and our prototype implementation targets the JVM), similar problems and approaches apply to other platforms. Our implementation is available online from `https://github.com/asrabkin/Relogger`.

6.2 Quantifying Ambiguity

Many log analysis applications, as a building block, map log messages back to their originating statement. This sub-analysis is used both to group messages and also to use logs to reason about program execution. To further motivate our approach to enhancing program logging, this section points out a fundamental limitation of all such analyses that do not modify the program in question.

Sometimes, two distinct log statements can produce textually identical output. In particular, this can happen if a log statement is inside a code clone. Lexical ambiguity imposes an upper bound on how well any retrospective analysis can match log messages to statements. This ambiguity is problematic for techniques such as SherLog [86] that use logs to infer a program's precise execution path. Our approach, explicitly identifying the source of every log message, breaks the ambiguity.

To assess this bound, we performed a small experiment on the Hadoop filesystem. We used the static analyzer from previous chapters to extract the string contents of log statements. We then ran a simple workload – starting a cluster, running a word-count job, and turning off the cluster. This is a basic workload, using only standard well-debugged parts of the system. Table 6.1 summarizes

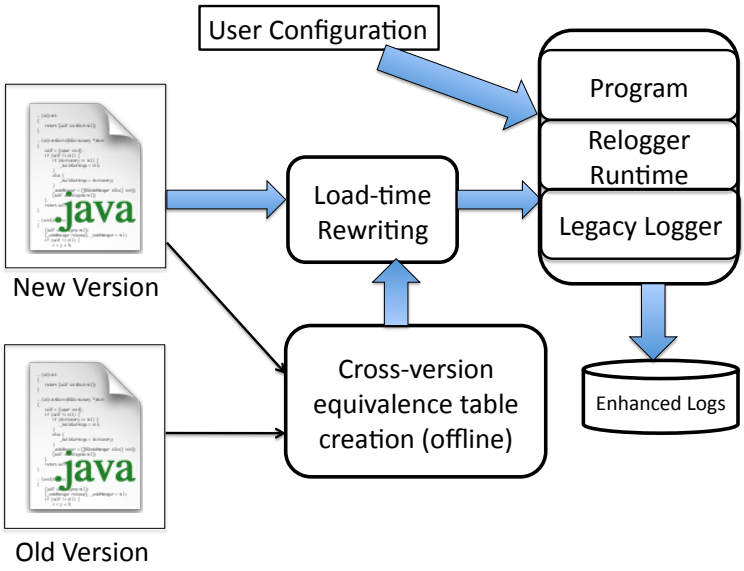

Figure 6.1. Illustration of overall approach to improving logs. Thick blue arrows represent steps that take place at runtime; thin black arrows are precomputation.

our results. Statically, 3% of statements are indistinct, meaning that two different points in the code could give rise to identical log messages. In our concrete workload, 4.5% of log messages came from these indistinct statements, and were therefore ambiguous.

Put another way, even a perfect retrospective analysis will fail to locate the source of 4.5% of messages in this workload, while an approach with precise identifiers would succeed. This limitation is not limited to automated analysis; a human programmer would face the same ambiguity.

This upper bound is not quite tight – one might imagine an analysis that infers the origin of a log statement using some sort of path-sensitive reasoning based on which other messages are printed. However, the point remains that it is easier to match statements with messages in advance, rather than retrospectively.

Statements (Static)	1127
Indistinct Statements	37 (3.3%)
Messages (Dynamic)	175
Ambiguous Messages	8 (4.5%)

Table 6.1. Frequency of lexical ambiguity in the Hadoop filesystem. A measurable fraction of messages cannot be unambiguously tied back to a specific logging statement.

Note that this is an upper bound. Real message-grouping algorithms may perform worse than optimal. Nor is Hadoop likely to be the worst-case for ambiguity. Precise static string analysis can fail on large or complex programs. Machine-learning approaches work badly on small input sizes. Hence, completely resolving this ambiguity is an advantage of log-rewriting over post-hoc approaches to message grouping.

6.3 Design

This section presents the enhanced logging interface we advocate. We begin with the user-visible portions of the design, and then present the naming system we have devised for logging statements.

6.3.1 Interface

As we mentioned in the introduction, our purpose is to offer fine-grained control of logging. This control has several aspects:

Enabling It should be possible to enable or disable a single log statement in the code, at run-time. In contrast, today's logging libraries often require the user to control logging at the granularity of files (or of "logger" objects, which are commonly one-to-one with files).

Custom Labeling Today, logging statements are commonly associated with a priority or severity level (such as "WARN", "NOTICE", or "DEBUG". The severity level is used both to determine which messages are printed and also to label the resultant messages. Having unique identifiers per statement helps decouple these aspects. Having the ability to refer to a specific statement lets users specify the labels or tags that should go along with the messages from that statement.

Today, systems sometimes have log messages for deprecation warnings, notifying users that they are using a feature that will eventually be removed or that might have hidden drawbacks. This is not a warning of any particular or immediate risk, hence, it might be a better interface design to give it a label like "SUGGESTION."

Grouping Having a unique ID per logging statement does not require users to configure statements one at a time. Statements could be grouped into named sets, either at runtime or before. (In particular, it is straightforward for the logging library to populate statement sets automatically based on the hierarchy of the source code.) These sets can be enabled or disabled together. Unlike traditional logger interfaces, these sets need not form a strict hierarchy: statements can be associated with any number of sets, in any structure.

Mapping statements to code The logging library should help users map messages back to a statement in the code responsible for them. Authors of log analysis tools today rely on complex and resouce-intensive methods to do this, such as whole-program static analysis. The process can also be tedious and time-consuming for humans engaged in debugging.

However, the logging library can greatly ease this process. Per-statement identifiers mean that a lengthy file name and line number does not need to be printed with each message. Instead, a short identifier can be put in each message, and a table emitted separately with one entry for each identifier, associating it with a source location.

6.3.2 Naming

Here, we present the specific approach we took in identifying log statements and the design considerations that led us to it. This approach to naming log statements, and the demonstration that it is practical to implement and supports the desired feature set is the core of our technical contribution.

Our high-level goal was to have concise, durable, and unique identifiers for each statement. We can break this down into several sub-goals:

Uniqueness Two distinct log statements should have distinct names. Absent this, we would lose fine-grained control.

Uniqueness across versions A log statement in one version should not share a name with a different log statement in another version. This requirement prevents a stale logger configuration from silently causing trouble.

Durability across executions A given statement should have the same name in all executions of the same program. Without this, configuration would be a constant chore.

Durability across versions If a program and its log statements are modified slightly and a statement in the new program is similar to a statement in the new, the two log statements should have the same name. This lets valid configuration remain valid across a program upgrade.

Conciseness The displayed name for the user should be compact, to ease recognition, reduce cognitive load, and reduce storage and I/O consumption.

Automatic Assignment Names must be assigned without imposing a burden on the developer for every statement.

We do not offer a precise definition of the "closeness" between log statements in different versions. We want results that programmers will find reasonable. We suspect different programmers will want different approximations here, and we therefore sought to separate cross-version matching from the core of our architecture.

Satisfying many subsets of these requirements is straightforward; satisfying all is challenge. Absent the uniqueness and durability requirements, we could use the source location of each statement. Absent the durability requirements, we could number statements sequentially based on a traversal of the program code. Absent the requirement for conciseness, for instance, we could simply encode a summary of a log statement's structure and use techniques similar to those for code clones [36]. (We discuss possible alternatives in more detail in Section 6.6.1.

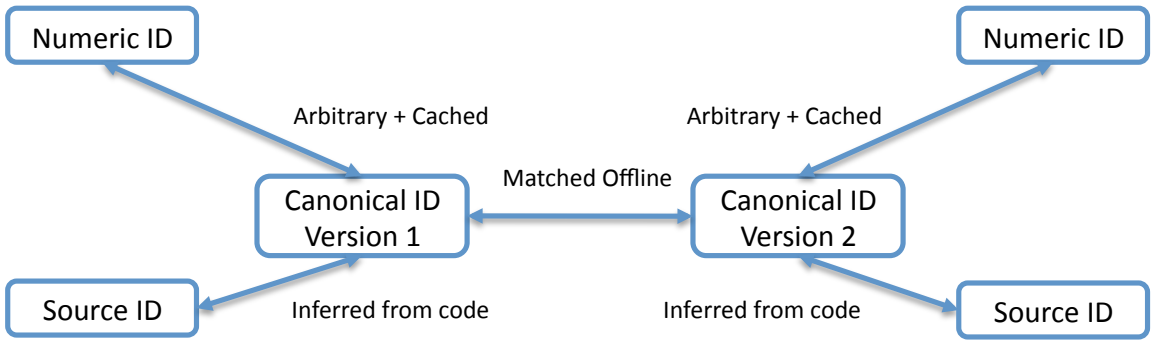

Figure 6.2. The approach we took to naming log statements assigns several different forms of names to each statement. Arrows represent relationships between names managed by the system.

Our solution was to split up the notion of naming into two pieces: *canonical names* and *convenient names*. The mapping between the two is computed explicitly and incorporates cross-version information. This ensures that all our desired properties are upheld. The overall approach to naming is summarized in Figure 6.2.

A *canonical name* designates a particular statement in a particular version of a source file. For example, in our Java implementation we use the hash of the class file containing the statement and the statement's position within the file.

These names are *rigid*: they refer unambiguously to a given statement in a given version of the program. In any program execution, a log statement will always have the same canonical name. If the source file containing a log statement is changed in any way, the canonical name will not refer to any statement [44].

This approach is convenient internally, since the logging library, at program start time, can compute the name for any given statement. However, names like `28e327a19aca95833-902b099f4d87349_1` are unfriendly for humans. These names are also *too* rigid: they will change, completely, if the underlying source file changes in any way. They also fail our conciseness criterion.

We therefore add two kinds of more-convenient names. We number every statement in the program, in some arbitrary but fixed sequence. We term these *statement numbers*. They are meant for user consumption. We opted to use integers because they are easy to work with internally, easy to parse, and compact in the output logs. They are also easy to `grep` for at debugging time. Our architecture could easily support using user-specified convenient names, including arbitrary textual strings, to identify statements.

The mapping between canonical names and statement numbers is arbitrary. We envision that developers or users will compute a mapping ahead of time, statically. This mapping can then be read by the logger at run-time.

Users can also use a source file and line number (a *source location*) to designate a log statement, although the user is responsible for making sure that they are appropriate for the version being run. Source locations are mapped to canonical names and thence to statement numbers at load-time, using the program's debugging symbol table. (Accessing this information about a caller is expensive; it can be accessed efficiently while a class being loaded.)

This hybrid naming approach separates the problem of designating statements, in one program version, from that of identifying statements in different versions as equivalent. Users can rely exclusively on convenient names; these do the work of designating statements. Behind the scenes, the logging library maps these to the appropriate canonical names at run-time.

The logging library also maintains a mapping (the *equivalence table*) between pairs of canonical names. This table is computed off-line and records when a given statement is "effectively equivalent" to a different statement in a different version. Using this table, the logging library can ensure that statement numbers remain the same when a program is modified. We expect application developers would generate these tables before a release and manually check or edit the results. They would then be bundled with the release, allowing forward compatibility of user configuration. Alternatively, they could be generated by a user during an upgrade.

We discuss the details of how our prototype does this matching in Section 6.4.4. From an architectural point of view, though the key point is that the user-visible interface and run-time are separate from the decision of how to do the matching. Arbitrary algorithms for matching can be used, without a run-time performance cost or needing to update deployed systems.

6.4 Implementation

This section describes how the interface described above can be implemented. We begin by discussing general strategy, then describe our concrete prototype implementation for Java.

6.4.1 Strategy

Programs already have loggers (which we refer to as *legacy loggers*). Building a new logging library would duplicate existing functionality. Instead, we advocate enhancing legacy loggers, by building the additional functionality we describe as a thin layer. This leverages the legacy logger for output and formatting, while the enhancement layer handles configuration and labeling.

Rewriting logging calls to use a new library is a straightforward program transformation. Depending on the language in question, it might be accomplished through macro substitution, library interposition, or some other technique. Supporting our naming scheme for logging statements,

however, requires something more: it requires a way of unambiguously identifying versions of a given piece of code.

As we mentioned, in Java we use the hash of the compiled class file along with a sequential ID within the file. In C/C++, we might use the file name, line number, and compilation time.

6.4.2 Class Transformation

We turn now to the details of our concrete implementation, `relogger`.

We use bytecode weaving, very similar to AspectJ, to redirect logging calls into the `relogger` run-time. However, AspectJ is not suitable for our purpose. While AspectJ allows the required call redirection, it does not give easy access to the raw bytes of a class file, which are required to compute canonical IDs. Hence, we built our own simple class transformer.

Our transformer replaces each call to the legacy logger with a call into the `relogger` run-time. We insert the appropriate identifiers on the caller side. Figure 6.4 offers an example of this transformation.

This rewriting could potentially be done statically, either before or after compiling the program. Instead, we opted to bundle it as a library that rewrites classes incrementally as they load. We use the Java Instrumentation API; this allows users to specify, on the command line, a series of transformations to be applied to each class as it loads into the JVM. This was done for deployment reasons: we wanted users to be able to adopt `relogger` without having to recompile software and without having to maintain a separate modified version of the program whose logs they wish to enhance.

As each class loads, we check if it has been previously seen. If so, the cached canonical-to-statement number mapping is used. If not, we check if there is an equivalence table entry for the statement. If so, then the new statement is equivalent to an old statement, and the ID and configuration for that statement are used. Failing that, (a wholly new class) then sequential numeric IDs are assigned to each logging statement, starting from a number higher than any used number. A copy of the canonical-to-statement mapping is stored on disk, read on startup, polled thereafter, and updated whenever a new mapping is introduced.

Approximately half of the run-time overhead of `relogger` is due to concatenating the statement ID with the log message. This can be optimized out in many cases. If the log message being printed is a compile-time constant, we can merge the statement ID with the message, at program-rewrite time. We keep both the original and altered copy of the message text, so our rewrite will not interfere with other uses of the string constant.

Programs sometimes need to introspect on their own logging. For example, the code fragment below is a common programming pattern:

```
if (LOG.debug_enabled()) {
   LOG.debug("Internal structures: " +
        format_data());
```

```
Example.java:

Logger = LogFactory.get(Example.class);

void some_method() {
 Logger.info("text");
//outputs "INFO Example: text"
}
```

Figure 6.3. Existing code, showing typical use of a logging library in Java.

```
Log = LogFactory.get(Example.class);

void some_method() {
 Relogger.Log(42, Logger, "info", "text");
//outputs "INFO Example: [123] text"
}
```

Figure 6.4. Same call, as modified by `relogger`.

```
}
```

This pattern avoids incurring the cost of computing and formatting in cases where log a statement is disabled. This requires a way, from within the program, to find the identifier of a log statement. To support this, we offer a method, `idOfNextStatement`, which returns the ID of the lexically-next logging statement.

Sometimes, a programmer wishes to introspect or alter a log statement other than the lexically-next one. If `idOfNextStatement` were a normal method call, it would require that point in the code to be reached for the ID to become available. Program rewriting again rescues us. At translation time, we look for calls to `idOfNextStatement`. We use dataflow analysis to find where that ID is assigned to a variable outside the local context. Our rewriter moves that assignment into the class initializer so it executes before the statement is reached. Effectively, the `idOfNextStatement` has been relocated implicitly. Figure 6.6 illustrates this rewriting. We would have preferred to use annotations, making explicit that the statement ID is inserted before the program executes. Sadly, Java annotations can only apply to declarations, not to method calls. We are therefore using a method call as a sort of pseudo-annotation.

```
Example.java:
static int optionalLogID;
void some_method() {
  optionalLogID = idOfNextStatement()
  Log.info("text");
}
```

Figure 6.5. What the programmer writes to retrieve a statement number from within a program

```
static int optionalLogID = 123;
void some_method() {
  Relogger.Log(123, Log, "info", "text")
}
```

Figure 6.6. What the machine executes: the statement number has been made accessible from outside the method in question, before that method is reached

Old Version	New Version	Matching Canonical IDs	Identical Content + Position	Approximate	Only in Old Vers.	Only in New Vers.
hadoop-0.18.2	hadoop-0.19.2	0	47	398	38	567
hadoop-0.19.2	hadoop-0.20.2	136	79	759	38	153
hadoop-0.20.2	hadoop-0.21.0	16	5	896	210	614
hadoop-0.21.0	hadoop-0.23.0	108	25	966	432	359

Table 6.2. Statistics about the evolution of Hadoop logging. Most log statements are consistent between versions. Most changes are the addition of new statements.

6.4.3 Runtime

Our rewriting gives control to the `relogger` run-time whenever the program invokes a logging method. The runtime listens for user commands, either via UDP or via a designated file on disk. Commands can enable, disable, or tag statements. By default, `relogger` lets the legacy logger decide whether to print a message. As a performance optimization, `relogger` will cache the legacy logger's decision. Calls to reconfigure the legacy logger are intercepted and the cache is cleared in response. To help users understand which statement corresponds to which number, `relogger` by default records the first message from each log statement, along with the associated statement number and source location.

We have support for three legacy loggers at present, showing that our approach is adaptable to existing code-bases. Supporting each legacy logger took only a few dozen lines of code. No difficulties emerge for applications that use multiple logging libraries, as can happen when an application is built out of separately-developed components.

Retrofitting our new API to existing code requires the co-operation of two pieces: the class

transformer and the runtime library. The transformer is responsible for rewriting legacy code to call the runtime; the runtime handles logging requests and user reconfigurations.

Sequentially numbering log statements had substantial implementation benefits. We can use bit vectors and integer arrays to store per-statement configuration. This, in turn, improves performance. (As we show, performance is higher than the existing legacy loggers in some cases.) In particular, the initial decision of whether a statement should be omitted is a single bit-set lookup. This bit-set is copy-on-write, meaning that, when printing a message, lookups are fast and require no synchronization.

6.4.4 Translation

Our implementation reads several tables at startup: a mapping from canonical to convenient name, and an optional set of equivalence tables, relating the canonical names of two different program versions. We have offline tools for generating each of these tables.

The canonical-to-convenient mapping is simple to produce. A Java program takes as input a list of jar-files or classes, and reads each class. Log statement IDs are assigned sequentially. This process runs very quickly – milliseconds per class.

Equivalence tables are more complex. In our prototype, we generate equivalence tables off-line using a separate tool. We extract the constant text from all log statements, using an available static analysis system [62]. We then use a greedy algorithm based on Python's built-in text similarity metric. For each message in version A, we pick the best match in version B, doing approximate matching on the statement's constant text along with its location in the source code. If the best match is close enough in absolute terms, we treat the two statements as matched. We then remove the paired statements from further consideration. The exact match threshold is tunable, we found that 80% similarity worked well.

		Without Relogger		With Relogger		Relogger no-fuse	
Logger	Enabled	Mean	Std dev	Mean	Std dev	Mean	Std dev
Apache Commons	No	5.31	0.02	2.66	0.02	2.66	0.01
Java.util.log	No	1.90	0.01	2.67	0.03	2.66	0.02
Log4J	No	3.83	0.02	2.66	0.02	2.66	0.02
Apache Commons	Yes	544.84	2.03	652.24	2.84	805.66	4.14
Java.util.log	Yes	29982.00	200.14	10230.00	107.14	10853.00	151.79
Log4J	Yes	543.84	2.20	721.52	3.96	876.80	4.20

Table 6.3. Microbenchmark performance, showing that Relogger imposes no overhead in the non-printed case, and modest overhead in the message-printed case. No-fuse turns off the optimization that merges statement numbers with string constants. All times are in nanoseconds.

6.5 Evaluation

We evaluate our implementation in two ways. First, we demonstrate that our offline matching is effective in finding corresponding messages between versions, using Hadoop, the popular open source MapReduce and distributed file system, as a case study. Next, we use microbenchmarks to measure the overhead imposed by our more flexible logging library.

6.5.1 Durability of Numbering

As noted, we cope with software evolution by doing off-line matching between different versions. To assess the effectiveness and necessity of this matching, we tried our matching process on five successive versions of Hadoop, releases 0.18, 0.19, 0.20, 0.21, and 0.23. (Version 0.22 was never released.) Hadoop is a large, complex software system, making it a reasonable case study.

The results are shown in Table 6.2. We break down matches where the class file hash was unchanged, those where the exact line number and text of a message was unchanged, and those where either content or position changed. As can be seen, most matches are approximate, not exact. This shows that line numbers really do change between versions, and so software evolution must be handled somehow.

We manually inspected the results, checking how close the non-exact matches were, and whether there were plausible candidates for messages that should have matched, but didn't. We did not see any spurious matches or omitted matches. The approximate matches involved only slight changes; often only the line number or a minor wording change. We did not see any pairs of messages that narrowly fell outside our match criterion. We conclude that, at least for Hadoop, log messages change minimally or else are essentially rewritten from version to version, without an ambiguous middle ground in which cross-version identification is questionable. As a result, simple matching algorithms are adequate.

6.5.2 Performance

We use microbenchmarks to show that `relogger` does not impose significant performance costs. Logging overhead, with or without our library, is a negligible fraction of program runtime and could be swamped by execution time variance. Hence, micro-benchmarks are the best evaluation methodology.

Each benchmark involved running the code in question a million times, as warmup, and then again many times to measure performance. We thus are measuring the expected steady-state performance for JIT-compiled code. (This follows Oracle/Sun's best practices for Java microbenchmarks [64].)

Table 6.3 reports performance for both enabled and disabled messages. For the "enabled" messages, the logger was configured to format the message text, but to discard it instead of writing

to file. This means that IO overhead and noise is not included in the measurements. Output is handled exclusively by the legacy logger, so performance should be identical with and without relogger.

As can be seen, `relogger` has lower overhead than some legacy loggers when a statement is disabled — numbering statements allows highly optimized data structures to decide whether to log a message. In contrast, `relogger` adds overhead to the "enabled" path: adding statement IDs and tags to the message has perceptible cost. For Log4J and Apache Commons log, the overhead is approximately an extra 150-250 nanoseconds per message. For a program printing a message per second, this is on the order of parts per million— much too small to be noticed operationally. The optimization that inserts the statement ID statically into the appropriate string constant saves 150-200 ns. Effectively, the overhead `relogger` would otherwise have is cut in half.

The standard `java.util.logging` library is strikingly slower than other loggers when printing messages. This is because it prints the source code location of each log statement by default. This performance cost further motivates the use of `relogger`, which can inline such information.

We also benchmarked the overhead of the class-rewriting process, measuring how long it takes to rewrite the 1132 classes making up Hadoop 0.20.2. Hadoop is a large complex application containing several hundred thousand lines of code. Without rewriting, the total load time was 1.6 seconds, or 1.4 ms per class. With rewriting, it was substantially higher – 8 seconds total, or 7 ms per class. While this is a significant relative increase, the absolute numbers are still negligible: 6.5 seconds of overhead amortized over the lifetime of a long-lived service is unlikely to be operationally significant. We do not expect the performance of the rewriting to change markedly depending on the program in question. Our analysis and transformation is purely method-local and is lightweight.

6.6 Discussion

We now discuss the applicability of our approach to other platforms and possible extensions to our work.

6.6.1 Possible Alternatives

There were several several design choices we considered but ultimately opted against.

Manual labeling Instead of using load-time rewriting, developers could audit their code and enforce this property ahead of time. (This approach was suggested to us by a reader of an early draft of this chapter.) The burden might be reduced by using tools to search for ambiguity and then following some standard for identifying messages. This would achieve the same benefits as our approach, without the difficulty of rewriting. However, it would add a substantial engineering

burden. Enforcing uniqueness would require constant vigilance to avoid code clones and care to make sure identifiers always remained unambiguous.

Perhaps worst, this approach requires ongoing active cooperation from the developers. A particular site, or perhaps a research group, would face difficulties maintaining a fork of the source tree with unambiguous logs. A program incorporating a library with its own log statements would face the same challenge. Our approach obviates this need, since it can be done for legacy code and has low ongoing costs as the software evolves.

Assigning IDs incrementally Our implementation enhances program logging incrementally, as classes load. We had initially hoped to also assign statement IDs incrementally and cache the results. This approach works adequately for a single process. However, it is unsuitable for distributed systems. We would like each copy of the program to have identically-numbered log statements. However, the order in which statements are seen will vary from program instance to instance. Assigning numbers consistently would require blocking execution to synchronize instances – which we judged too heavy a price. Since one of the most important uses for logging is to monitor and debug distributed systems, we abandoned incremental name assignment.

Consistent Hashing for IDs Instead of numbering IDs sequentially, we might try to derive some consistent but durable identifier based on the location in the code and the message text. That is, we might have a deterministic function from source location and message text to ID, in such a way that small changes leave the ID unchanged.

This would be effectively merging the notions of canonical and convenient identifiers. It would avoid imposing an arbitrary naming, and therefore circumvent the concurrency problem mentioned above. There are three problems here, however.

The first is that the mapping function has strictly less information available than our offline matching. Offline, we can compare two complete programs and use global structure. If IDs are assigned statement-by-statement, this global structure is unavailable.

The second problem is that having a fixed algorithm for assigning statement numbers makes upgrades challenging. All the policy about detecting moved statements is baked into the numbering. Change the policy, and all existing numbers change. This means that user policy cannot remain fixed during an upgrade— precisely the condition we sought to avoid.

Last, an ID big enough to be durable and unique will be excessively large for users. We can calculate the necessary name size by treating the statement-to-name mapping as a random hash function. In this model [6], the number of elements required to have a probability p of a collision is approximately $\sqrt{2H \cdot \ln \frac{1}{1-p}}$, where H is the size of the range of the function. To have a probability of collision less than 1% for a program with a thousand log statements, this would require names to have 25 bits, roughly equivalent to an 7-digit number or a 6-element hexadecimal number. We thought this was an undue burden on users and therefore rejected the approach.

Using only canonical IDs Much of the complexity in our scheme comes from trying to support short convenient IDs as an aid to human readers. Our approach could be simplified if this goal is dropped. Instead of displaying convenient IDs, the logs would include only a canonical ID. It would still be possible to keep identifiers consistent in the presence of code changes, using equivalence tables.

Using source locations for identification Still another possible alternative would be to throw out canonical names and use only source identifiers (file name and line number). This would also remove substantial complexity. However, it would also remove substantial benefit. With only source identifiers, there is no way to disambiguate which version of the code is meant. When a user configuration refers to "the statement at Line X of file Y", does that mean of the old version or the new?

Alternative matching approaches In our prototype implementation, we match log statements between versions based purely on the statement text and source code location. An alternative approach would be to generate these entries from a program's version-control history: two statements are likely related if a patch removes one and adds the other. The engineering complexity of this approach did not seem warranted in a prototype implementation, however.

6.6.2 Non-Java languages

The core idea of numbering statements to improve control applies to other languages, which may not offer fine-grained control in their logging libraries. The basic structure of our approach – a shim layer between the legacy logger and the application, with imposed statement numbers – should be broadly applicable. Likewise, doing cross-version matching out-of-band will be possible regardless of implementation.

In contrast, the use of load-time rewriting is Java-specific. For other platforms, it might make sense to do the modification statically, on either binaries or source code. Alternatively, for legacy C or C++ code, the logging library could determine the caller by inspecting the call stack. Calls to legacy loggers can be intercepted into a revised logger either by a static change to the program or by using library interposition.

6.6.3 Future work

Our work addressed only logging done via dedicated libraries. However, programmers sometimes use standard IO mechanisms – *e.g.* `printf` – for debugging. In principle, these statements could be handled by our approach in exactly the same way as calls to logging library. However, not all console output consists of log messages, and any change to the text output from non-logging IO might break the program's functional specification. We therefore assume any message modification in this context would need to be off by default.

The work presented here describes introducing a new set of mechanisms for controlling program logging. We have not addressed the question of policy. Once it is possible to rapidly toggle whether a specific statement is enabled, it becomes possible to use sophisticated algorithms for choosing what to log. We might, for instance, rate-limit or sample specific statements. We might choose to sample all log messages from a particular time window, just to record what is being discarded the rest of the time. A range of other approaches is surely also possible.

Chapter 7

Conclusions

This dissertation has presented several applications of program analysis to system management and debugging problems. In particular, we have presented static approaches to configuration management and debugging, along with dynamic techniques for improving the quality of program logs and the degree of control over logging given to users.

In the next three sections, we review the major technical contributions of our work. Following that, we outline some possible avenues for future work. Last, we offer some thoughts about applying program analysis to complex codebases.

7.1 Managing Configuration

Chapter 4 showed that static analysis can help discover configuration options, document these options, and catch potential configuration bugs. To summarize in more detail, we found the following:

Open-source programs are rife with stale documentation for configuration. All of the programs we examined had documentation for options that were not actually present. This appeared to be the result of error, not design. The programs we looked at also contained undocumented options. Some undocumented options appear to be intended only for unit test writers. Others might be useful to users, particularly users with atypical needs. As a result, the lack of documentation is a problem worth correcting.

Configuration option names are available statically. The ground truth can be established automatically and statically. Our static analysis was able to find the overwhelming majority (more than 95%) of options in the programs we looked at. This is more complete than existing manually-produced documentation.

String analysis and external dependencies pose the biggest challenges to static analysis

of configurability. Points-to and call graph imprecision was not the biggest challenge for our analysis. The chief limitations we found were two-fold. Values are sometimes stored inside strings, frustrating simple data-flow analysis. Some options are passed to external programs before being used. This frustrates single-program analysis.

Option behavior can often be described and documented automatically. For some types of options, such as class names and Booleans, analysis can find a high proportion of options, without any observed no false positives. For options drawn from small sets of strings, analysis can often also give the set of legal values. Beyond documentation, this offers the possibility of catching a range of user configuration mistakes, via a "spell check for configuration."

7.2 Debugging Configuration

Chapter 5 addressed the related problem of diagnosing configuration errors after the fact. The key claims of our argument are as follows:

Computing and storing configuration dependencies is tractable. Our results also help diagnose errors after they occur via static analysis. In theory, each program point could depend on an arbitrary subset of the program's configuration options, resulting in excessive computation and storage costs. In practice, we have observed that most program points depend on only a small number of options. As a result, storing the dependencies at each point is feasible, with size roughly proportional to the program's code.

Dataflow analysis can explain typo-induced errors. The high accuracy of our analysis implies that, at least for the programs we studied, there are usually data dependencies between configuration values and the points in the code where a bad value can cause an error.

For diagnosing configuration errors, it is safe to model library code, rather than analyzing it. Our approach uses an approximate model for library code, instead of analyzing the library and tracking the flow of options through it. This gave us substantial performance benefits without causing any false negatives in our tests.

Failure-context-sensitive analysis helps exploit stack traces for configuration debugging. In our testing for Hadoop, failure-context-sensitive analysis reduced the imprecision of static analysis by a third while taking approximately a minute per stack trace.

Programs should log the configuration options they read. If a program does not read all its options, then knowing which options were actually read substantially improves analysis precision. Recording this information only requires developers to insert a handful of log statements, rather than make extensive changes to their programs. Since even large systems (like Hadoop) often have only a few hundred options, the logging overhead is small.

7.3 Controlling Logs

Chapters 5 and 4 addressed named options with static analysis. Chapter 6 addresses logging and logger configuration, using primarily dynamic techniques. We include this work to show the wide range of programming-language techniques that are applicable to configuration and debugging problems in open-source systems.

Unique statement numbers address problems with existing logging. Giving developers and administrators an unambiguous way to refer to individual log statements enables precise control over which messages print and how they are labelled. This is demonstrated by our working prototype. Our implementation shows that this need not have significant performance impact.

Careful design of the log statement naming enables robustness across program changes. By separating the names used by users from those used internally in the system, a logging library can detect and compensate for a program change, without breaking user configuration or changing the visible statement identifiers in the logs.

Load-type rewriting works well for Java. This approach compensates for the incremental and modular nature of Java code loading. Alternate approaches might be preferable on other platforms. For example, static rewriting of C or C++ might be more appropriate.

Similar approaches would work on other platforms The core idea of numbering log statements to improve control over logging applies to languages beyond Java. The basic structure of our approach – a shim layer between the legacy logger and the application, with imposed statement numbers and cross-version matching done off line – should also be broadly applicable.

7.4 Future Directions

7.4.1 Generalizing the Problem

The analysis presented in Chapter 5 outputs a list of options that might explain a failure. A potential topic for future work would be ranking the results. Combining our debugging tool with the "configuration spellcheck" approach discussed in Chapter 4 may be a way forward. We conjecture that if a configuration option is associated with an error message, and also appears to be of the wrong type, then that option is much more likely to be the culprit. Another way to combine the work of Chapters 4 and 5 would be to use option types to explain *why* an option is wrong.

We do not address errors that manifest themselves as performance problems, resource exhaustion, or silent failures. Addressing these errors will require a revision to our problem statement, not just algorithmic refinements. In these cases, there is no unique option or small set of options responsible for the problem. If a program runs out of memory, any configuration option that controls memory allocation could in principle be tuned to make the error go away — potentially a very large set. The options that control the preponderance of memory allocations will be workload-dependent. Therefore, dynamic analysis may be better suited to this problem.

7.4.2 Generalizing to Other Languages

Our prototype and experiments were restricted to Java. The techniques we outline might not work as well for languages such as C where the weaker type system can make points-to analysis more challenging. Our results in Chapter 4 used among the simplest points-to algorithms, suggesting that we can find options and infer their types even in harder-to-analyze languages.

Even the 2-object-sensitive analysis used in 5 is simpler than many static analysis approaches that have been explored in the literature. This suggests that optimal points-to accuracy is not required for configuration debugging. Moreover, nothing in our technique requires a sound points-to. Our analysis should degrade gracefully in the presence of approximate heuristic call-graph construction. This would allow usable, even if imperfect, results, for C programs.

As with all static analysis approaches, our techniques would not work to diagnose problems in "programs" consisting of multiple components in several languages. This includes the important special case of problems that are caused by a program's start script.

7.4.3 Generalizing to Other Problems

One way to think about static analysis is that it is the process of constructing a static model, intelligible to humans, of what happens inside a running program. In this work, we had the advantage that configuration options and log messages are composed of text strings that are present both in the program and at run-time on the user's screen. As a result, we can use these labels to tie together a program analysis with a user-visible challenge. Chapter 6 turns this insight on its head: the work presented there describes how to automatically modify a program to include suitable strings that match source code log statements to user-visible messages.

There are other problems where this insight about the importance of constant strings may be useful. We could potentially use similar insights to help debug web applications, matching the page structure elements or web form inputs to back-end events and code.

7.5 Handling Complexity in Applied Program Analysis

A distinctive aspect of this dissertation has been the application of static analysis to large, complex code-bases. Here, we consider some lessons of that experience. These are intended to guide future researchers who seek to employ static analysis.

A static analysis is typically designed to model the semantics of some particular programming language. But the semantics of the language are only a partial description of how programs execute. The operating system and the hardware offer control-flow and data-flow paths not defined by the programming language in question. An accurate analysis must take these into account. Programs often have callbacks or signal handlers invoked asynchronously by the operating system or runtime. They often rely on interprocess or networked communication. Data can be written into

the filesystem and then flow back into the application. To make analysis effective on on systems software, it must be augmented to deal with these aspects of the execution environment.

We have explored two different strategies for this. At first, when we hit some hard-to-analyze structure, such as collections or remote procedure calls, we added a special case to our analysis to account for them. Later, we switched an approach we call *analyzable models*. In this approach, we create a Java stub implementation of the code fragment to be analyzed. Our analyzer is then configured to analyze the stub instead instead of the concrete implementation.

To give a concrete example: in the first approach, we would create a datalog relation inside the analyzer for a program's multiple entrypoints. In the latter approach, we would create a Java stub that invokes each application entry point. This stub is the analyzable model of the application.

We found that analyzable models were usually simpler to create and reason about. Designing an analyzable model requires reasoning about a specific (and small) piece of Java code, rather than about how an analyzer would respond to any possible Java program. Another advantage of analyzable models is that they integrate cleanly with any analysis. We began using them heavily when we switched to object-sensitive analysis.

These models had their own challenges, however. Often, a model of a program-specific feature, such as multiple entry-points, would have a static dependence on the code-base being modeled. This was a challenges, since it meant that changing the target program required altering the model before it would even compile. In contrast, adding additional analysis rules imposes only a run-time dependence. This could potentially be addressed by using reflection within the model. Another disadvantage of analyzable models is that they allow higher-order rules, for instance of the form "for all classes with a certain property, do the following."

Regardless of which strategy is employed to cope, we found that these sorts of tweaks to well-understood analysis techniques were essential to get good results on complex code-bases. As a result, program analysis cannot simply be used as a black-box technique: different sorts of programs use different programming idioms; getting good results often means making sure that these idioms are handled accurately.

7.6 Last thoughts

Regardless of what strategy is employed to model the complexities of modern systems software, this dissertation has shown that such complexities can be managed. We were able to extract information about program configuration, in ways that were industrially useful. We showed that configuration problems can often be debugged without any run-time work. And we showed that program logs can be improved retroactively. We hope this encourages others to apply program analysis techniques to systems problems.

Bibliography

[1] Apache Ant. `http://ant.apache.org/`.

[2] Apache Derby. `http://db.apache.org/derby/`.

[3] HBase. `http://hbase.apache.org/`.

[4] Scribe. http://sourceforge.net/ projects/ scribeserver/, 2008.

[5] T. j. watson libraries for analysis (wala). Online. `http://wala.sf.net/`, August 2011.

[6] Wikipedia: Birthday attack. Online: `http://en.wikipedia.org/wiki/Birthday_attack`, April 9 2012.

[7] Michal Aharon, Gilad Barash, Ira Cohen, and Eli Mordechai. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, 2009.

[8] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report 2009-28, UC Berkeley, 2009.

[9] Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *2008 USENIX Annual Technical Conference*, 2008.

[10] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[11] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1996.

[12] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa, a large-scale monitoring system. In *First Workshop on Cloud Computing and its Applications (CCA '08)*, Chicago, IL, 2008.

[13] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices*, 44(10):243–262, 2009.

[14] Aaron B. Brown and David A. Patterson. To err is human. In *Proceedings of the First Workshop on evaluating and architecting system dependability (EASY'01)*, 2001.

[15] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[16] Raymond P.L. Buse and Westley R. Weimer. Automatic documentation inference for exceptions. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, New York, NY, USA, 2008.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[18] A.S. Christensen, A. Møller, and M.I. Schwartzbach. Precise Analysis of String Expressions. In *SAS '03: Proceedings of the 10th International Symposium on Static Analysis*, 2003.

[19] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28:324–328, June 1996.

[20] Xiaoning Ding, Hai Huang, Yaoping Ruan, Anees Shaikh, and Xiaodong Zhang. Automatic software fault diagnosis by exploiting application signatures. In *Proceedings of the 22nd USENIX conference on Large installation system administration conference (LISA'08)*, 2008.

[21] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. *SIGPLAN Notices*, 33(5), 1998.

[22] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, 2009.

[23] J. Dong, Y. Zhao, and T. Peng. Architecture and design pattern discovery techniques-a review. In *International Conference on Software Engineering Research and Practice (SERP)*, 2007.

[24] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. *VLDB*, 2009.

[25] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *19th Symposium on Operating Systems Principles (SOSP)*, 2003.

[26] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *Proceedings of the 35th Annual Symposium on Principles of Programming Languages*, January 2008.

[27] R. Fonseca, G. Porter, R.H. Katz, S. Shenker, and I. Stoica. XTrace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, MA, April 2007.

[28] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *International Conference on Data Mining (ICDM)*, 2009.

[29] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: ten years of implementation and experience. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[30] J. Gray. Why do computers stop and what can be done about it. In *Symposium on reliability in distributed software and database systems*, pages 3–12. IEEE Computer Society Press, 1986.

[31] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998.

[32] Dan Hettena and Rick Cox. A guide to nachos 5.0j. `http://inst.eecs.berkeley.edu/~cs162/sp07/Nachos/walk/walk.html`.

[33] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, pages 54–61. ACM, 2001.

[34] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):106, 2004.

[35] Hai Huang, Raymond Jennings, III, Yaoping Ruan, Ramendra Sahoo, Sambit Sahu, and Anees Shaikh. PDA: a tool for automated problem determination. In *Proceedings of the 21st USENIX conference on Large Installation System Administration Conference (LISA'07)*, 2007.

[36] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*, 2007.

[37] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 249–267, 2008.

[38] Niels Jørgensen. Putting it all in the trunk: incremental software development in the freebsd open source project. *Information Systems Journal*, 11(4):321–336, 2001.

[39] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings of the 2008 International Conference on Dependable Systems and Networks (DSN)*, 2008.

[40] Brian Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Inc., 1999.

[41] C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Symposium on Static Analysis*, 2006.

[42] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering*, 1996.

[43] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: inferring the specification within. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, 2006.

[44] Saul Kripke. *Naming and Necessity*. Harvard University Press, Cambridge, MA, 1980.

[45] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *ICSE*, 1994.

[46] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A Structured Storage System on a P2P Network. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[47] T.C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6):35–39, 2003.

[48] M. Levesque. Fundamental issues with open source software development. *First Monday:*, Special Issue #2: Open Source, October 2005.

[49] Ben Liblit. *Cooperative Bug Isolation*. PhD thesis, UC Berkeley, 2004.

[50] Junghee Lim, Thomas Reps, and Ben Liblit. Extracting output formats from executables. In *Working Conference on Reverse Engineering*, 2006.

[51] B. Livshits, J. Whaley, and M. Lam. Reflection analysis for Java. In *Third Asian Symposium on Programming Languages and Systems*, 2005.

[52] C. Lonvick. RFC 3164: The BSD syslog Protocol. http://www.ietf.org/rfc/rfc3164.txt, August 2001.

[53] W. Luks, O. Tkachuk, and D. Bushnell. Automatic extraction of jpf options and documentation. In *JPF Workshop*, 2011.

[54] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[55] M. Michlmayr, F. Hunt, and D. Probert. Quality practices and problems in free software projects. In *First International Conference on Open Source Systems*, 2005.

[56] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14:1–41, January 2005.

[57] Mayur Naik. JChord. `http://jchord.googlecode.com`.

[58] A. Oliner, A. Ganapathi, and W. Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2):55–61, 2012.

[59] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks (DSN)*, 2007.

[60] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[61] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff. Mining for misconfigured machines in grid systems. In *International Conference on Knowledge Discovery and Data Mining*, 2006.

[62] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, 2011.

[63] E. Reisner, C. Song, K.K. Ma, J.S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*, 2010.

[64] John Rose. MicroBenchmarks. `http://wikis.sun.com/display/HotSpotInternals/MicroBenchmarks`.

[65] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[66] C. Rubio-González and B. Liblit. Expect the unexpected: error code mismatches between documentation and the real world. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2010.

[67] Jason Sawin and Atanas Rountev. Improving static resolution of dynamic class loading in java using dynamically gathered environment information. In *Automated Software Engineering*, 2009.

[68] Daniel Schreck, Valentin Dallmeier, and Thomas Zimmermann. How documentation evolves over time. In *IWPSE '07: Ninth international workshop on Principles of software evolution*, 2007.

[69] E.J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[70] B.H. Sigelman, L.A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, 2010.

[71] Janice Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*, 1999.

[72] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *ACM SIGPLAN Notices*, volume 46, pages 17–30. ACM, 2011.

[73] M. Sridharan, S.J. Fink, and R. Bodik. Thin slicing. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.

[74] J. Stearley. Towards informatic analysis of syslogs. *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 2004.

[75] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: improving configuration management with operating system causality analysis. In *21st Symposium on Operating Systems Principles (SOSP)*, 2007.

[76] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *21st Symposium on Operating Systems Principles (SOSP)*, 2007.

[77] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *SOCC*, 2010.

[78] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.

[79] R. Wang, X.F. Wang, K. Zhang, and Z. Li. Towards automatic reverse engineering of software security configurations. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS)*, 2008.

[80] Y.M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H.J. Wang, C. Yuan, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *LISA*, 2003.

[81] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *18th Symposium on Operating Systems Principles (SOSP)*, 2001.

[82] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.

[83] Wei Xu, Ling Huang, Michael Jordan, David Patterson, and Armando Fox. Detecting Large-Scale System Problems by Mining Console Logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[84] Zuoning Yin, Jing Zheng, Xiao Ma, Yuanyuan Zhou, Shankar Pasupath, and Lakshmi Bairavasundaram. An empirical study on configuration errors in commercial and open source systems. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[85] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. Automated known problem diagnosis with event traces. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2006.

[86] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of ASPLOS 2010*, 2010.

[87] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *Proceedings of ASPLOS 2011*, 2011.

[88] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *ESEC/FSE*, 1999.

[89] Alice X. Zheng. *Statistical Software Debugging*. PhD thesis, UC Berkeley, 2005.

[90] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML*, 2006.

Appendix A

Documentation for software artifacts

Some of the research in this dissertation has been embodied in usable software artifacts. In particular, we have developed usable implementations of the configuration option extractor, discussed in Chapter 4, and Relogger, discussed in Chapter 6. This appendix includes the programmer documentation for these artifacts.

A.1 Conf-alyzer

Conf-alyzer is a tool for analyzing Hadoop's configuration usage. It is available from `git@github.com/Confalyzer.git` It is built on top of JChord (`http://code.google.com/p/jchord/`).

A.1.1 Usage

Invoke conf-alyzer by running:

`python analyze_config.py <hadoop or hbase jar> [ onlyhdfs | onlymapred| onlyhbase ]`

You should ignore the printed warning about `chord.work.dir`. Expect it to take 20-40 minutes depending which version of Hadoop you are analyzing.

The analyzer assumes that the Jar is alongside the usual hadoop lib directories with the dependencies. So you should be pointing it, e.g., at `hadoop-xyz/hadoop-core-xyz.jar`.

By default, the tool will check all entrypoints for mapred+hdfs. Specifying `only[mapred|hdfs|hbase]` will look at the entry points for those systems.

Outputs a list of options in hadoop-configuration.html, tagged with:

- Name
- Default value, both in -defaults.xml and in the code.

- Where in the code the option is read.
- Which daemons potentially use the value.
- The description in -defaults.xml.

Right now, this is extracted via static analysis, alone.

A.1.2 Configuration

The main configuration option that needs to be monitored as Hadoop evolves is the list of entrypoints. JChord doesn't know which main() methods matter and which classes can be invoked remotely, so you have to specify those explicitly. The files hbase-entrypoints and entrypoints-20-all have lists of the main methods and RPC-exposed interfaces that I know about. If you are getting inadequate code coverage, see if there's some entrypoint that's missing.

Much of the core of the analysis is written in datalog. By default, a Java datalog solver is used. If you want to make analysis faster and you know what you're doing, you can switch to using buddy, a native datalog solver. Add the line "chord.use.buddy=true" to chord.properties, and drop a suitable 'buddy' library into the lib directory.

A.1.3 Debugging

Log output will be in `chord_output/log.txt` Other important files in chord_output are methods.txt, the list of reached methods and MM.txt, the call graph.

A.1.4 Planned refinements and known limitations

- The option name regexes have .* when they should have something like `\W*` .
- Dynamic analysis to pinpoint excessive reads and to get more precise information about usage.
- Should print inferred type.

A.1.5 More Information

The static analysis uses the JChord analysis tool by Mayur Naik et al. See `http://code.google.com/p/jchord/` for JChord documentation. JChord is under BSD license.

The option extraction algorithm is based on the one presented at ICSE 2011 in the paper "Static Extraction of Program Configuration Options" by Ariel Rabkin and Randy Katz

A.2 Relogger

Relogger, the retroactive enhancement logger, lets you add new features to the logging subsystem for existing legacy Java programs. If your program uses standard Java loggers like Log4j or Apache Commons logging, Relogger gives you an enhanced logger **retroactively**, without the need for any source code changes.

Support for additional loggers can be added upon request, even support for System.out,err.println() ”logging”.

A.2.1 Features

Relogger offers several features not present in conventional loggers.

- By the magic of program rewriting, every log statement is numbered uniquely. The number of the corresponding statement is inserted into every log message. This eases analysis.
- It’s possible to enable or disable any particular statement, by number, at run-time.
- You can get every message printed once, providing an example of all the trace/debug messages.
- The overhead of not printing a message is **lower** with relogger than with the underlying logger, so the performance cost will be low.

A.2.2 Future Features

These can be added if called for.

- Using the print-just-once feature for sampling and rate estimation.
- Tagging messages with arbitrary strings, like ”USER” or ”ADMINISTRATOR.”
- Changing the printed priority; if you think an ERROR message isn’t, you can change that without recompiling, or even relaunching, your program.

A.2.3 Building Relogger

Just say ’ant’. You should get a numberedlogs.jar

A.2.4 Using Relogger

To use relogger: Add asm-3.3.1-all.jar to your application classpath. This is the bytecode manipulation library used by Relogger. Add -javaagent:numberedlogs.jar to the java VM arguments list for your application. You need to add it before the main-class argument – this is a VM argument, not an application argument.

You should replace numberedlogs.jar with the path to wherever you put the relogger library.

You can also pass several arguments. The syntax looks like this:

```
-javaagent:numberedlogs.jar=portno=2344,file=/tmp/relogger
```

Formally:

```
-javaagent:<path to agent>=<arglist>

 <arglist> = <option=value>
           = <option=value>,<arglist>
```

Available options:

port/portno The UDP port on which to listen for instructions. Defaults to 2345

file The location to read and store statement-numbering information. Defaults to 'relogger/mapping.out'

alwaysonce Whether to print every message at least once. Any value for this option evaluates to true, except 'false'

A.2.5 Command Syntax

Relogger takes several commands. You can send send these by UDP to the command port (defaults to 2345).

In addition to UDP, it's possible to configure Relogger by creating a file named "commands" in the same directory as the mapping file (by default, 'relogger'). This file is checked for changes every two seconds. It should contain a list of commands, using the same syntax as for UDP.

```
up <statement>Enables a given statement-set
on <statement>

down <statement>Disables a given statement-set
off
```

`once <statement>` prints the given statement-set at least once the next time it appears. If the statement is enabled, it will be printed more than once.

Many of these commands take a statement-set as argument. This is a string that designates one or more log statements.

```
<statement_set> =  [numeric statement ID]
            =  [canonical ID of the form 0x...._lineno]
            =  [class name:lineno]
            =  [tag name]
```

A.2.6 Tags

Tags are an in-progress feature. A tag is a string that labels a set of messages. This comes with two new commands:

`addtag <statement> <tag>` Add the given tag to the given statement-set
`rmtag <statement> <tag>` Remove the given tag from the given statement-set
`deltag <statement> <tag>` Equivalent to above