AUTOMATICALLY IDENTIFYING CONFIGURATION FILES

by

Zhen Huang

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Automatically Identifying Configuration Files

Zhen Huang

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2009

Systems can become misconfigured for a variety of reasons such as operator errors or buggy patches. When a misconfiguration is discovered, usually the first order of business is to restore availability, often by undoing the misconfiguration. To simplify this task, we propose Ocasta to automatically determine which files contain configuration state. Ocasta uses a novel *similarity* metric to measures how similar a file's versions are to each other, and a set of filters to eliminate non-persistent files from consideration. These two mechanisms enable Ocasta to identify all 72 configuration files out of 2363 versioned files from 6 common applications in two user traces, while mistaking only 33 non-configuration files as configuration files. Ocasta allows a versioning file system to eliminate roughly 66% of non-configuration file versions from its logs, thus reducing the number of file versions that a user must manually examine to recover from a misconfiguration.

Acknowledgements

First and foremost, I would like to express my gratitude to Professor David Lie for his patient supervision, relentless enthusiasm, financial support, and large amount of time spent on discussions for this research.

I am thankful to my fellow graduate students: Lionel Litty, Tom Hart, Lee Chew, and Stan Kvasov. I would also like to thank Professor Ashvin Goel and the members of the Security Reading Group (SRG) for their valuable feedback.

I am very grateful to my family for their much needed moral support.

Finally, I would like to thank University of Toronto and the department of Electrical and Computer Engineering for their financial support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Configuration management is the bane of system administration. An application can become misconfigured due to a multitude of causes – human error, incorrect patches or buggy software, just to name a few. In addition, misconfigurations are not always immediately obvious and may only affect an application when used in a certain way. As a result, the cause and symptom of a misconfiguration can be separated by weeks or even months.

When a misconfiguration is discovered, often the system administrator's immediate priority is to restore availability to the system. While the ideal solution is to fix the misconfiguration, configuration debugging is a laboriously slow process. Therefore, a more short-term and realistic goal is to roll the broken application back to the state just before the misconfiguration occurred. Finding this state may also provide clues on how to find a more permanent fix for the misconfiguration. Many systems facilitate configuration rollback by providing some capability for recording and rolling back the state of the file system. For example, Windows Vista comes with Shadow Copy disks [32] and Mac OS includes Time Machine [21], both of which take block level snapshots of an entire disk. Such systematic solutions are frequently combined with ad-hoc "best practices", such as keeping configuration files under revision control.

However, these solutions are not quite satisfactory for recovering from misconfigurations. Versioning an entire disk or file system consumes a lot of disk space. As a result, such systems take snapshots, so they may miss file versions, and even then, usually can only maintain several weeks' worth of versions. In addition, when a misconfiguration occurs, the user is confronted with a legion of files and versions within which she must find the cause of the problem. Thus, current versioning systems will only work if the cause of the misconfiguration is recent, and still require the user to try rolling back many different files to find the right one.

Manually maintaining versions in revision control or manual backup reduces the number of files versioned, saving both space and effort. However, it requires the user to know the exact location and number of configuration files for each application. Unfortunately, configuration complexity has driven more and more application developers to provide a tool or graphical interface for configuration, rather than allowing the user to edit configuration state directly. This effectively abstracts and hides the storage of the configuration settings from the user. Unlike traditional Linux/UNIX or Windows applications that store configuration in `/etc` or in the Windows registry, these applications have being storing configuration state in a variety of files that do not have descriptive names. Versioning file systems, which record versions at the file system level instead of the block level, can selectively version files on a file system, but they still place the burden on the user to find the configuration files [43].

Since we cannot assume that a user knows where the configuration files for her applications are stored, we propose a system, called *Ocasta*, that will automatically find the locations of an application's configuration files for the user. Ocasta treats applications as black-boxes, requiring no a priori knowledge of the application, and heuristically ranks a file's likelihood to contain configuration state by observing how it is accessed and how its content changes over time. Ocasta performs a statistical analysis of how a file's contents change over time to determine whether the file stores configuration state or not. The intuition behind this approach is that configuration state changes very slowly, so that files that exhibit a high amount of *similarity* as they are modified are more likely to contain configuration state. Thus, by measuring the similarity of a file's versions, Ocasta can assign a score that denotes the likelihood that a file stores configuration state. This information is then used to guide a versioning system into selecting which files to keep and which to discard when it is approaching its space quota. Having a more accurate picture of which files to version not only reduces the storage requirements of the versioning system, but also reduces the effort the user must expend to roll the application back to a working state and unravel the cause of the misconfiguration.

1.1 Contributions

This research makes the following three contributions to the area of configuration management:

1. To the best of our knowledge, it is the first research that systematically analyzes the locations, formats, sizes, and change patterns of configuration files

2. A novel statistic-based similarity metric to identify configuration files for arbitrary applications

3. A working system that automatically identifies configuration files using the similarity metric and aids a user in recovering configuration problems

1.2 Thesis Structure

The structure of this thesis is as follows. In Chapter 2, we provide the relevant background for the reader. Chapter 3 characterizes the problem Ocasta faces with identifying configuration files. Chapter 4 presents our trace-based analysis of configuration files. Chapter 5 describes Ocasta's solution for identifying configuration files and which versions hold significant configuration changes. Finally, we present evaluation results in Chapter 6 and conclude in Chapter 7.

Chapter 2

Related Work

This Chapter presents the most important research related to this thesis. Firstly, we highlight the difference of our work with other work in configuration management. Secondly, we explore work in system recovery and versioning file systems. Finally, we present work that used Rabin fingerprint.

2.1 Configuration Management

There has been a lot of interest in configuration management. Recent work [22, 36–38] shows that configuration problem is a significant cause of system faults. Broadly speaking, research in this field can be classified into three categories: providing aids to the users to fix configuration problems, validating user actions that change configurations, automating some error-prone tasks of configuration management.

Much of this work [1, 4, 31, 47, 51–54] uses computers to aid the process of configuration problem troubleshooting. Chronus [53] searches for the time when system configuration transitioned from a working state to a non-working configuration state. During the diagnosis process, Chronus loads and runs historical system snapshots in a virtual machine monitor, and tests whether a historical system snapshot works correctly by executing user-provided software probes. To keep a history of the system snapshots, Chronus logs every change to the disk in a time-traveling disk. In addition, it uses binary search to speed up the search of the state transition point.

Ocasta is inspired by the idea of searching for historical working state of Chronus. However, Ocasta differs from Chronus in two ways. First, a major problem of the Chronus method is that it can not differentiate configuration-related disk block changes with user data-related disk block changes. Thus it can inadvertently cause user data loss with its indiscriminating disk block roll back. Chronus solved the problem by rolling back only configuration-related file changes. Second, binary search can not guarantee to find the latest working state if there are multiple working state to non-working state transitions. Ocasta ensures that the latest working state is found by using linear search.

Autobash [47] and its successor [4] help a user to locate existing solutions to configuration problems. They maintain a database of user actions to known configuration problems and uses a trial-and-fail approach to test candidate solutions. To ensure that an incorrect solution does not leave persistent changes to the system, they apply each candidate solution in a speculative environment. Any persistent changes made by a solution can be undone if the solution is not appropriate for the configuration problem that the user is troubleshooting. Like Chronus, they execute user-specified software predicates to automatically test whether the configuration problem is solved or not. In addition, they leverage causality-tracking and analysis to decrease the time for finding and testing solutions. Most recently, Su et al. [48] further enhances Autobash by using heuristics algorithms to automatically find user actions that are used to test system states as software predicates.

While Autobash allows a user to learn from other users that experiences the same or similar configuration problems, it can not solve the problems that do not have existing solutions. Ocasta is able to fix these kind of problems since it can find out the historical state when the user's system was working.

PeerPressue [51] and its predecessor, Strider [52] uses statistical analysis to find the root causes of misconfigurations. They maintain a database that stores the system state of a large body of computers. To diagnose a configuration problem, they first trace the faulty application and use causality-tracking to find suspect Windows registry entries that may cause the configuration problem. Then they retrieve the same set of entries from the system state of other computers from the database. Finally, all these entries are used in Bayesian statistical estimations to calculate the probability for each entry to be the root cause.

Like other statistical approaches, the success of PeerPressure in finding the correct root causes of misconfigurations for an application requires the use of a large set of system states from computers that runs the same application. This requirement may be feasible for enterprise users that have a large number of installations of computers that run the same set of applications. But it is difficult for an individual user to collect this kind of information from other people's computers. A software vendor is also unlikely to do so due to privacy and other issues. Ocasta avoids this kind of problems since it does not need information from other users' computers.

Some research [36, 37] provide frameworks to validate configuration changes before they are put in effect on production systems. Nagaraja et al. [36] explored the nature of operator mistakes with operator experiments on system maintenance and troubleshooting tasks. They implemented a prototype validation framework with two alternative techniques: trace-based validation and replica-based validation. Oliveria et al. [37] conducted a survey on a group of database administrators to study database configuration problems. They introduced a new model-based validation technique in its validation framework, which is based on [36]. Both studies indicate that operator mistakes are significant source of system faults. The experiments with these validation frameworks illustrate that they can effectively detect and prevent some operator mistakes, but they miss a considerable number of operator mistakes due to the complexity of the interactions between the operator and the system.

Others [2, 3, 9, 55] have proposed to use computers to accomplish some of the most tedious and error-prone tasks of configuration management. These systems allow the user to specify the rules that the system will follow to automatically generate configuration files. They usually provide the user with some high-level language directives or templates that define how the configuration files should be generated under certain circumstances. Although they relieve some of the burden of configuration management from the users, they still rely on the user to supply error-free rules, which can become complicated since there are numerous different kinds of circumstances in which a configuration file needs to be changed.

2.2 System Recovery

Computer systems can fail for many different reasons [19]. It has been reported that software bugs and operator mistakes are the most significant sources of system failures [5, 20, 22, 30, 36, 37]. Despite the fact that many mechanisms have been proposed to detect software bugs or to identify operator mistakes, many bugs or operator mistakes still bypass the detections and cause system failures, largely due to the sophisticated nature of these issues. Consequently, we believe a more pragmatic solution is to enable computer systems to automatically recover from failures. We present here some of the primary work in this field.

Previous work has used reboot or restart techniques, including whole program restart [19, 49], microreboot [10], and software rejuvenation [26, 50] to bring the system to a clean state after system failures. Whole software restart can cause substantial software down time, although it is the most effective way to recover from some system failures. Microreboot reduces restart time by recursively restarting the smallest recoverable component, a software module for example, to the entire software. In many cases, a software module restart can restore the availability of the system and it incurs much less time than a entire software restart. However, being able to microreboot requires the software to be built as a collection of ”recoverable” components. While legacy applications can not enjoy this technique, many complicated applications will be difficult or nearly impossible to be designed this way.

Checkpoint and recovery [6, 24, 27, 28, 39, 41, 44] is another important failure recovery technique. It reduces system down time by periodically taking a checkpoint of the system state and restarting the system from a prior checkpoint. A checkpoint consists of at least the memory state of one or more applications [16]. Many times it also includes a partial or full file system snapshot. The checkpoints can be stored to disk, non-volatile memory, or remote systems.

Rx [41] periodically uses light-weight checkpoint techniques that store a snapshot of an application in main memory and keep a copy of each accessed files of the application. When a software failure is detected in the application, Rx rolls back the application to a recent checkpoint and re-executes the application in a modified environment. This mechanism is able to recover the application from many memory management related problems such as memory corruption and double free, certain non-deterministic software bugs such as race conditions, or user requests based problems that are triggered from some user requests or inputs.

ASSURE [44] identifies an application's rescue points, which are locations in the existing application code for handling failures, and inserts checkpoint and rollback code into the system with runtime binary injection. It first discovers candidate rescue points in an application offline by exploring the call-graphs of the application's executions when failure handling code is being triggered. An online test is then used to determine which candidate rescue point is most effective. The selected rescue point is inserted into the application in the production environment at runtime. When a fault occurs in the application, ASSURE restores execution to an appropriate rescue point and uses the applications own failure handling code to recovery from the fault.

Furthermore, work by others [11, 29] has shown that application-specific knowledge is necessary to recover from many kinds of application faults. Brown et al. build an e-mail store system [8] that is capable of rolling back and replaying application-specific user operations. Each user operation is converted into verbs, which is an encapsulation of all the operations needed to re-execute or compensate a user operation, and is recorded into a timeline log on disk by an undo manager. During the troubleshooting process, the user can edit the timeline log to remove, replace, or add verbs. The undo manager then re-executes the appropriate part of the timeline. This mechanism reflects the Rewind, Repair, and Replay model [7] that allows a user to repair a problem without losing work that was performed after the time at which the problem occurred.

2.3 Versioning File Systems

Versioning file systems keep a history of changes made to the file system to provide the ability to recover from file system related faults. Research in the field of versioning file systems can be classified into comprehensive versioning file systems, open-close versioning file systems, and snapshots versioning file systems,.

Comprehensive versioning file systems [13, 46] create a new version for every write to the file system. This approach provides the finest granularity of versioning. Wayback [13] is a user-level versioning file system built on the FUSE framework [15]. It logs the data to be overwritten or truncated into an undo log before any overwriting or truncating is performed. Our versioning file system uses an undo log similar to that of Wayback, but it is not implemented as a user-level file system. Rather it is implemented as a kernel module that intercepts file system calls made by user applications to create versions. The main drawback of a comprehensive versioning file system is the space overhead required since a great number of versions are created.

Open-close versioning file system [43] create a new version for all the writes occurs in a session of an open and a close of a file. The Elephant file system [43] is a versioning file system that provides users with a range of version retension policies. One interesting retention policy, called Keep Landmarks, retains only landmark versions in a file's history. A landmark version can either be specified by the user or automatically identified with heuristics, which is based on the assumption that versions generated within a short time period will be indistinguishable to the user if the versions were generated a long time ago.

Recently, Muniswamy-Reddy et al. proposed a versioning file system [33] that leverages the causal relationships among files. It aims to gain the benefits of both open-close versioning file systems and comprehensive versioning file systems by creating a new version only when a new causal relationship is observed between a process and a file. This limits the number of versions to be created while preserving sufficient causality information for the recovery.

Snapshots versioning file systems [25, 40] create versions for the entire file system. Each version is a snapshot or an image of the file system. Due to the overhead of taking and saving file system snaphot, these systems usually create versions periodically to avoid disrupting normal user operations.

2.4 Rabin fingerprints

Rabin fingerprinting schema [42] generates a fingerprint for a string by using randomly chosen irreducible polynomials. It guarantees the probability of a collision when two different strings have the same fingerprint can be made to be very small by choosing a sufficient number of bits to represent the fingerprint. It is widely used by string-matching and hashing algorithms since it is fast and is easy to implement.

Several systems have also used Rabin fingerprints to detect similarity across files. LBFS [35] uses it to detect similar chunks among different files for the purposes of optimizing bandwidth usage in a distributed files system. It divides the files it stores into chunks and indexes the chunks by the Rabin fingerprint of the chunks. It improves transmitting performance by avoiding transmitting a file chunk if any file on the recipient contains the same chunk. Forman et al. [17] use it to find similar documents to help manage large document repositories. Rabin fingerprints were also used by Earlybird [45] and Autograph [23] to detect common substrings in network traffic to detect Internet worms.

Chapter 3

Problem Definition

3.1 What is a Configuration File?

One of the difficulties in dealing with configuration problems is that the term "configuration file" is itself a nebulous term. To resolve this difficulty, we begin by defining a configuration file for the purposes of this thesis. We first introduce the term *application state*, which describes persistent state that an application maintains throughout its lifetime. We then categorize application state as either *task-dependent*, *time-dependent* or *configuration state*. A task is an application-specific unit of work, such as navigating to a web page with a browser, or editing a file with an editor. Thus, task-dependent state reflects the sequence of tasks an application has performed – an editor's list of recently opened files or the state of a browser's web cache for example. Time-dependent state records the passage of time or counts the number of occurrences of certain events – such as a timer to check for patches or perform a cleanup. Configuration state defines the application's behavior over time and across multiple tasks. With this model consisting of three types of application state, we can finally give our definition of a configuration file:

> **Configuration File:** A file that stores application state that is neither task-dependent nor time-dependent, and whose contents may thus affect an application's availabil-

ity or performance over a period of time and across a number of tasks.

We note that misconfigurations do not necessarily affect *all* tasks because not all configuration state is used for every task. Thus, while configuration state is not task-dependent, the act of accessing configuration state is. For example, directing a web browser to view only web pages on the local disk would not reveal problems stemming from misconfigured web proxy settings.

We found this definition useful when trying to decide whether a borderline case was a configuration file or not. One such case was for files whose contents are set by a remote party as opposed to the user, but which have a configuration-like function for the application. For example, Firefox periodically downloads a blacklist of phishing sites, which it uses to protect the user. This file's contents are neither task- nor time-dependent, but its contents can affect the usability of Firefox by preventing the user from visiting even legitimate websites if incorrectly configured. As a result, our definition considers such files configuration files. If this blacklist does becomes misconfigured, rolling this blacklist back to an earlier version will fix the problem – though possibly only temporarily. In addition, knowing that this file is the cause of a misconfiguration is valuable to the user. Thus, while our definition is broad, we feel it does contain most of the files that are of interest to system administrators who are trying to debug a misconfigured application. We concede that this definition also does not cover every possible source of misconfiguration. For instance, applications may derive their configuration dynamically from a device or network resource without actually writing it to the file system. Since Ocasta can only observe configurations saved to disk, it will not be able to analyze these cases. We leave the debugging of such configuration problems outside the scope of this thesis.

In addition, an application does not always store all of its configuration state in configuration files. Many applications provide default values for their configuration parameters. If a configuration parameter has a default value, the applications often do not store the value in their configuration files, since the default value of a configuration parameter can be defined as a constant in the application's code and thus would probably be saved into the application's executable file. It is also possible that the default value of a configuration parameter is implied by the logic of the application's code. Thus the configuration state of an application includes not only the configuration states that are stored in the configuration file, but also those stored directly as constants or indirectly as code in executable files.

3.2 Identifying Configuration Files

One may ask why it is difficult in the first place for a user to identify the configuration files of an application. For applications whose configuration files are not meant to be edited manually, the application developer has no reason to make the location of the configuration files obvious to the user. As a result, relying exclusively on file names is unreliable. For example, Firefox stores configurations in files such as *pluginreg.dat, secmod.db, prefs.js, localstore.rdf* and *mimeTypes.rdf.* The Macromedia Flash plugin maintains several files named *settings.sol* in various directories, but only one of them actually contains application configuration data (the others contain website-specific data). In addition, it appears that for portability purposes, both of these applications do not use OS-specific conventions for the storage of their configuration state. For example, rather than using the registry in Windows, Firefox stores all its configuration state in a subdirectory under the user's home directory, just as it does on Linux.

Another problem is that configuration files often have a sprinkling of time-dependent and task-dependent state mixed in with the configuration state. For example, Firefox stores timestamps of events, the period between updates, and the exit status of the previous execution (i.e. exited cleanly or crashed) in the *prefs.js* configuration file. Similarly, the GNOME desktop system records timestamps in each GNOME application's *%gconf.xml* configuration files. As a result, it is more appropriate to score files by the proportion of configuration state they contain, rather than try and classify files as entirely configuration or non-configuration. In addition, while the vast majority of the content in such files is configuration state, the vast majority of modifications to the files are actually to these sprinklings of non-configuration state. This adds a great deal of noise to the file system interaction between an application and its configuration files.

Finally, we note that configuration files are not always accessed in a predictable pattern. While many configuration files are read at start up [14], some applications read their configurations on demand. For example, Firefox will only access the *pluginreg.dat* configuration file if plugins are used or modified. Accesses to such configuration files occur neither in every execution of an application, nor do they occur at the beginning of the execution. As a result, any heuristic that assumes configuration files are always read at start up will miss configuration files.

3.3 Misconfiguration Recovery

Unfortunately, without knowledge of which files contain configuration state, recovery from a misconfiguration is a very expensive task. To illustrate, suppose an application has failed due to a misconfiguration and the user would like to determine which files read during the failing run are causing the problem. The user may profile the failing application to determine which files it accesses, and then query the versioning system to find all versions of those files. She may then start rolling back various files, one version at a time, to see if the problem is fixed. We note that while one could use binary search to speed up the task of finding a working configuration state [53], binary search can only identify a point where an application transitioned from a working state to a broken state. Unfortunately, this has the disadvantage that if there was more than one time in which the application's configuration state was not valid, then it cannot guarantee that the working file version it finds is the most recently working state. Since configuration files are not modified atomically, there can be many transient periods when the configuration files are in an invalid state. Thus, the only way to ensure that the most recent working configuration state is found is to do a linear search backwards in time from the current state of the application.

To quantify the difficulty of this task, we collected system traces from two Linux worksta-

Trace	Days	Application	Non-code files	Versioned files	Config	Versions (config/total)
1	22	Firefox	851	507	7	6485/295841
		GNOME	1552	392	16	321/2064
		Flash	37	17	1	7/130
		VMware	337	189	6	160/2926
2	12	Firefox	1247	568	15	234835/751808
		GNOME	2537	493	21	13298/30388
		Flash	53	35	1	15/1995
		JEdit	6876	73	2	46/5002
		Amarok	3086	89	3	43/4782

Table 3.1: Application file usage measurements. For each trace, we give the number of non-code files accessed by the application, the number of files that have been modified and have versions stored on the system, and the number of configuration files the application uses. The last column gives the aggregate number of configuration file and total file versions created by the application over the entire trace.

tions in our lab. The traces were collected with a kernel module that intercepts system calls, versions files using a redo log, and records which applications access each file. A new file version is created on every write system call or memory map of a file. In cases where several contiguous writes are made to a file, this is counted as a single version. Table 3.1 gives data extracted from the traces on several applications that were used by the workstation users. GNOME represents the GNOME suite of desktop applications, Flash is the Macromedia Flash plugin, VMware is VMware workstation, JEdit is a JAVA-based editor and Amarok is an open-source music player. We describe the applications in more detail in Appendix A. We note that Trace 1 has less activity than Trace 2, despite a longer trace period, simply because the user did not use the machine as much. To determine whether a file that was accessed contains configuration state, we use a combination of application documentation and profiling of the

Trace	Days	Write ops	Bytes	Files
1	22	7,884,155	3.01GB	12,983
2	12	62,647,496	21.28GB	96,470

Table 3.2: Versioning storage overhead measurement. We give the number of write operations, bytes written and number of files modified in each trace.

application.

The traces show that without knowledge of which files may contain configuration state, the user would have to sift through hundreds of files that were accessed and potentially try thousands of versions to find the most recently working configuration state. Instead of having to examine each versioned file, knowing which files are configuration files reduces the number of files the user has to consider by one to two orders of magnitude. The number of file versions to try is also reduced by an order of magnitude or more in most cases and by at least a factor of 2 in the worst case.

In addition, versioning only files that are likely to contain configuration state can reduce storage overhead. We estimate the amount of storage necessary to version each of the traces using the size of the redo logs from the trace. As shown in Table 3.2, both traces would require a significant amount of storage to version all the data written to the file system over the length of the trace. While one may argue that storage is cheap, the user effort required to manage the storage and analyze large amounts of data in the storage is expensive. Therefore, the ability to separate configuration state from other types of application state will be beneficial to system configuration management.

Chapter 4

Configuration File Analysis

To the best of our knowledge, no previous work has systematically analyzed configuration files. However, we believe this kind of analysis is necessary to discover an effective and efficient method in identifying configuration files. In light of that, we analyzed various aspects such as format and change patterns of configuration files for 7 popular Linux applications. The goal of our analysis is to find out how configuration files differ from non-configuration files. Our analysis aims to answer the following questions.

- Where does an application store its configuration files and non-configuration files?
- What are the format and size of configuration files and non-configuration files?
- What are the change patterns of configuration files and non-configuration files?

4.1 Methodology

We analyze application behaviors that are related with configuration operations in a way similar to black-box testing. We perform experiments with applications by running them in a controlled environment to collect the traces of file accesses and file system changes made by the applications. Then we study the file system traces from different perspectives in order to understand the nature of configuration files.

4.1.1 Trace Collection

We developed a versioning file system from the code base of Forensix [18] to collect the trace of each application. The versioning file system intercepts file system calls and stores every file system change into versioning logs. Each version log contains versions for one unique file. The versioning file system creates a new version for every write operation to a file. It also keeps the history of the file system calls made by the application. The file system call history is used to determine which files are accessed or updated by which applications, and when these accesses or updates occur.

To compare how the files are changed with and without configuration related operations, we performed both configuration operations and other regular operations with the applications. To be able to easily identify which file changes are related to configuration operations and which file changes are not, we divided our operations for each application into two phases: a non-configuration phase and a configuration phase. We first perform regular operations such as browsing web pages in a web browser or editing documents in a text editor in the non-configuration phase. We do not perform any configuration related operations during this phase. To make the application generates every possible file change, we execute a variety of different operations with the application. Furthermore, we execute the operations long enough to ensure the application produce a reasonably long history of file changes.

We perform configuration operations in the configuration phase after we finish the non-configuration phase. In the configuration phase, we try to make every possible configuration change by modifying the default value of every configuration parameter of the application from the interface provided by the application. These changes should be reflected in file system changes that are isolated from the file system changes in the non-configuration phase by the time of the changes.

4.1.2 Trace Analysis

An important objective of our trace analysis is to compare the change patterns of configuration files and non-configuration files. We achieve this by tracking the change history of each individual *data entry* in a file. A data entry can represent either a configuration parameter or a non-configuration parameter. We designed a simple split-and-match algorithm to approximately identify an individual data entry in a file.

The algorithm first splits a file version into unique *data fragments* by certain specific delimiters. A data fragment is a series of contiguous bytes in the file version. It is a particular instance of a data entry. For example, one configuration file (prefs.js) of Firefox can have two data fragments, each in a different version. One data fragment is "user_pref("javascript.enabled", true);" and the other data fragment is "user_pref("javascript.enabled", true);". These two different data fragments refer to two different values (true and false) of the same data entry that represents the configuration parameter with name "javascript.enabled".

For this study, we choose to use line feed as the delimiter to split a file version and analyze data changes of ASCII files only. As we will shown later, almost all configuration files are ASCII and they usually are organized into lines of text, which contains the name and the value of a configuration parameter. Many non-configuration files are also ASCII. So line feed can be used as a delimiter for most configuration files and many non-configuration files.

To track the change history of a data entry, we need to find the data entry to which a data fragment refers to. This is necessary since a data entry change corresponds to two different values of the data entry and thus two different data fragments. Failing to match the two different data fragments that refer to the same data entry, the data entry change will be recognized as two unrelated events: a removal of a data entry and a insertion of another data entry. Thus the data entry change will not be identified correctly.

To solve this problem, we use longest common substrings algorithm [12] to match two different data fragments that refer to the same data entry. One issue of this method is that two different data fragments that refer to two different data entries may be incorrectly matched if they happen to have a long common substring. We use two orthogonal approaches to eliminate this kind of incorrect matches. First, a match can only be made between a data fragment that is *removed* from a file version, and a data fragment that is *added* to the same version. This is because a data entry change means the data fragment refers to the old value of the data entry will be replaced by the data fragment refers to the new value of the data entry in the file version. In addition, two added data fragments or two removed data fragments can not refer to the same data entry, because this means the version contain two different values of a same data entry, which should not happen for configuration files. Second, we observed that a configuration file can not have two data entries that represent the same configuration information. Otherwise, the meaning of the configuration information will be confusing. Therefore, two different data fragments can not be matched if they ever co-exist in a same file version.

Based on our split-and-match algorithm, we developed a utility to analyze versioning logs that are generated by our versioning system. This utility works on one versioning log at a time. It extracts every version of a file, splits each of them into data fragments, match a data fragment to a data entry, and builds a database of these data entries. It tracks every change to each data entry. The utility collects information on each data entry including how many times the data entry is added in a version, is deleted from a version, is changed from one version to the next version, how many versions the data entry exists in, and the first time the data entry is added, deleted, or changed. This kinds of information describes the change patterns of the data entries in a file.

From the system call history, we find out which directories and files are accessed and updated by which application. For each application, we check which files it accessed contain data entries that are changed only during a configuration phase. These files are candidate configuration files since we make only configuration changes in a configuration phase. To determine which files are exactly configuration files, we manually rollback these candidate files one by one to the beginning of the configuration phase, and check if the rollback undo the configuration changes we have made in the configuration phase. If the rollback of a file undo one or more configuration changes, the file is identified as a configuration file. If the rollback of a file does not undo any configuration changes, this file is identified as a non-configuration file. All the files accessed by an application which do not contain data entries that are changed only during a configuration phase will be identified as non-configuration files.

4.2 Analyzed Applications

We choose 7 popular Linux applications, as shown in Table 4.1. They consist of 2 Web browsers, 3 document editors, 1 media player, and 1 Internet messenger. We intentionally select only desktop GUI applications since it is difficult to identify their configuration files even manually. The software vendors of these application usually provide little documentation on which files are configuration files.

4.3 Analysis Results

Our experiments with these applications are performed on a VMWare virtual machine that runs Ubuntu 8.10 Linux with kernel 2.6.27. We choose to use a virtual machine monitor since it is easier for us to rollback the system to a previous snapshot to undo any unwanted changes made to the system. In this section, we present our analysis on the location, format, size, content, and change patterns of the configuration files of our chosen applications. It should be noted that the configuration files and non-configuration files that we discuss in this section refer to those files of the applications under our analysis. Specially, the non-configuration files refer to only the non-configuration files that are stored in the same locations of the configuration files. We do not consider non-configuration files which are accessed by the applications and are stored in other directories for this study.

Application	Config File Location	Config Files (ASCII/Binary)	Non-config Files (ASCII/Binary)
Firefox	.mozilla	10	3/16
jEdit	.jedit	2	9
Amarok	.kde	1	0
Opera	.opera	3/1	8/13
Texmaker	.config	2	0
Openoffice Word	.openoffice.org2	12	0
Skype	.Skype	1	9

Table 4.1: Location and format of configuration files. The locations of configuration files are all relative to a user's home directory. It shows the number of ASCII and binary configuration files of each application. It also presents the number of ASCII and binary non-configuration files.

4.3.1 File Location and Format

As shown in Table 4.1, there are 32 configuration files for the 7 applications in total. All applications store their configuration files under some hidden directories in the user's home directory. Three applications have more than one configuration files. All except one of the configuration files are in ASCII format including XML format.

Only one application does not have any non-configuration files under the same directory that contains its configuration files. There are 58 non-configuration files under the same directories of configuration files. Half of the non-configuration files are also in ASCII format.

4.3.2 File Size

Figure 4.1 shows the sizes of configuration files and non-configuration files for all the applications. The sizes of the configuration files are distributed in three ranges of sizes: below 2KB, between 2KB and 5KB, and above 5KB. Around 35% of configuration files are less than

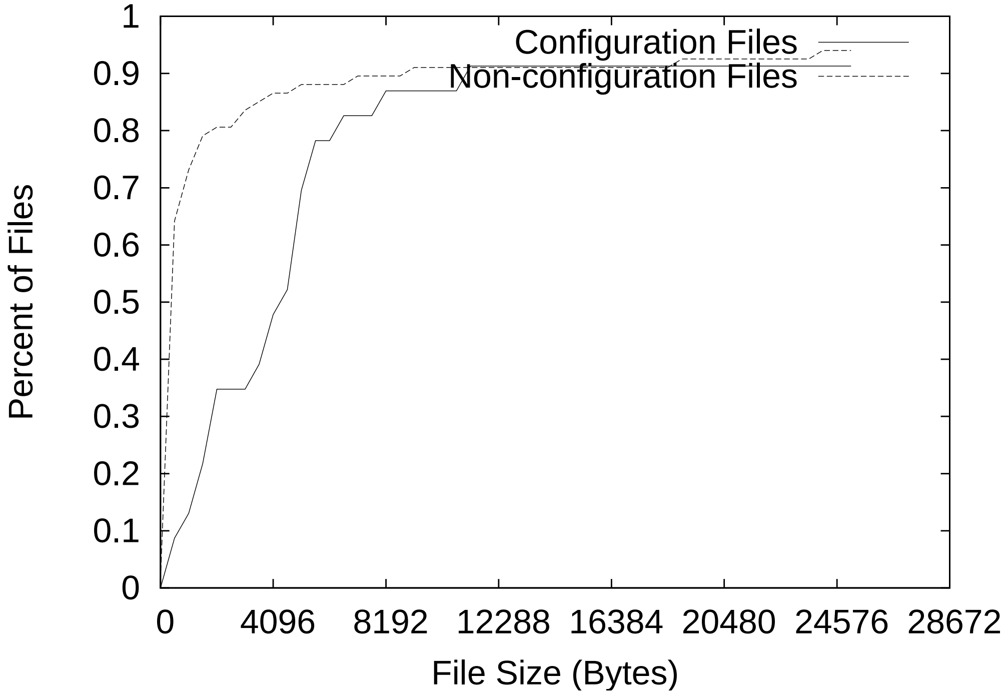

Figure 4.1: Comparison of the sizes of configuration files and non-configuration files.

2KB, while 22% of the configuration files are larger than 5KB. In contrast, most of the non-configuration files are very small. About 80% of non-configuration files are smaller than 2KB, although the largest files are non-configuration files.

4.3.3 Data Entry Type

We classify the type of data entries in a configuration file into either configuration state or non-configuration state. This data entry type has no relation with the concept of programming language data type such as binary, number, or character string. As described in Section 4.1.2, we identify configuration files by identifying data entries changed only during a configuration phase. These data entries are called *configuration data entries* and they make up the configuration state of a file. The percent of configuration state in a file is defined as the ratio of the number of configuration data entries to the total number of data entries of a file.

Figure 4.2 shows the percent of configuration state in each configuration file. Half of the

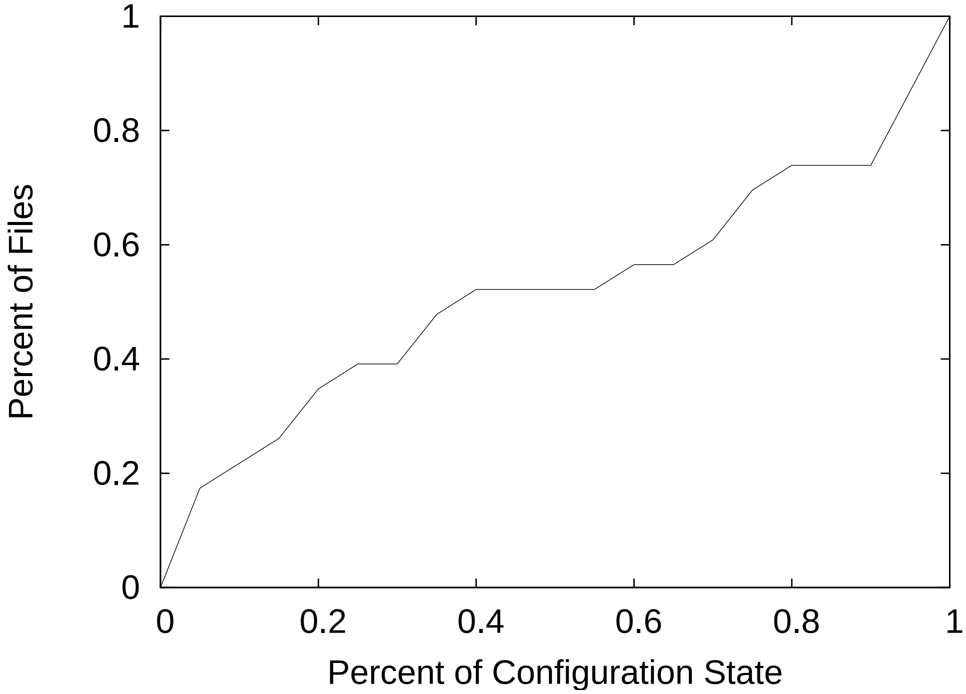

Figure 4.2: Percent of configuration state in all the configuration files.

configuration files (52%) contain less than 40% configuration state, and 25% the configuration files contain more than 72% configuration state. In addition, 30% the configuration files are almost entirely composed of configuration state. This indicates that configuration files contain data related to both configuration state and non-configuration state. Looking into the configuration files, we found that the non-configuration state includes timestamps, list of recently opened files, last user actions, window size of last execution, temporary file name, debugging information, and so on.

4.3.4 Data Fragment Life Time

We measure how much data in a file is changed across the different versions of a file by using the life time of the data fragments of the file. A data fragment's life time is the ratio of the number of versions in which the data fragment appears, to the number of the total versions of the file that contains the data fragment. We choose the configuration file with most versions

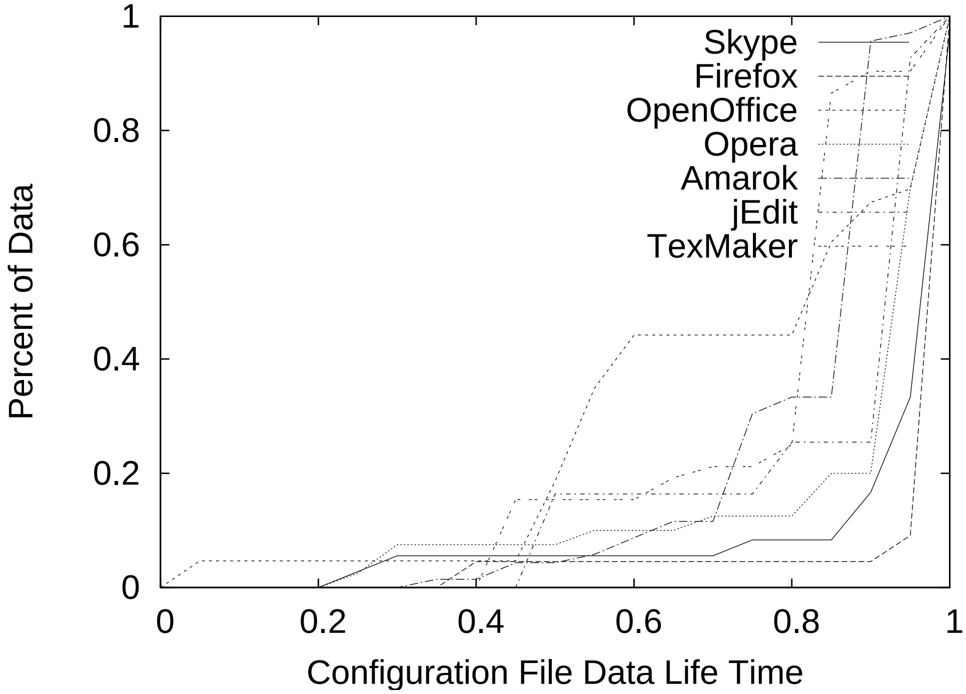

Figure 4.3: Life time of data fragments in configuration files.

for each application. Because some applications do not have non-configuration files under the directories in which their configuration files exist, we choose two non-configuration files for each of the applications that have non-configuration files. Figure 4.3 and Figure 4.4 present the data fragment life time of the configuration files and the non-configuration files, respectively. For the configuration files, all except one of the curves start low and remain flat from the short life time side, and rise steeply near the longest life time. This means most data fragments in the configuration files have a long life time. On the contrary, the curves of the non-configuration files generally rise steeply at the short life time side, which means most data fragments in the non-configuration files have a short life time.

4.4 Discussion

The analysis of configuration files illustrates both challenges and promises of our research in identifying configuration files. First, applications tend to store many non-configuration files

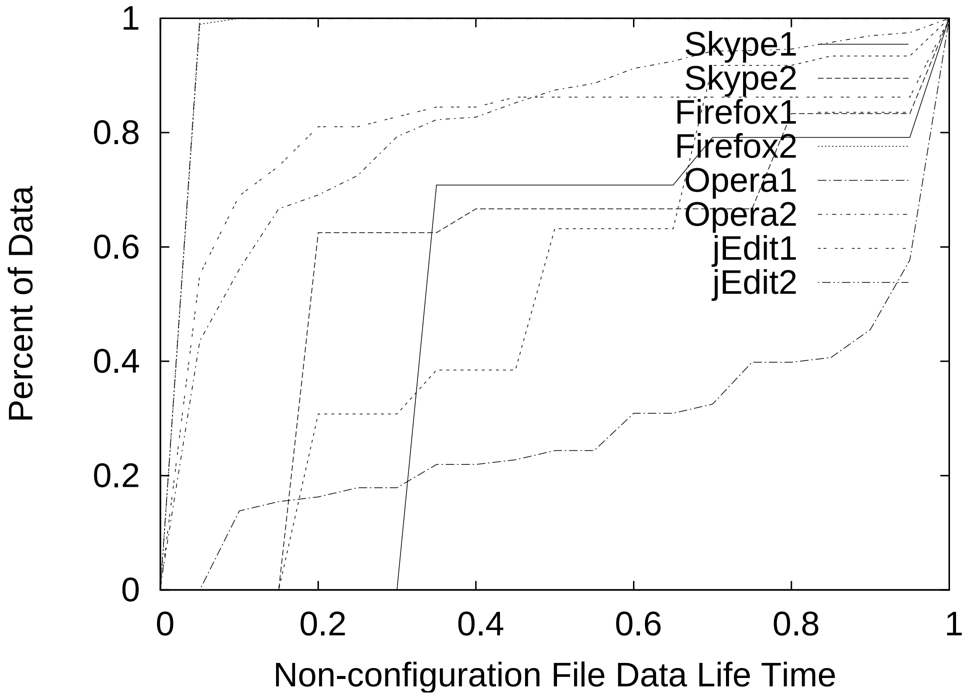

Figure 4.4: Life time of data fragments in non-configuration files.

in the same location where their configuration files are stored, although configuration files are all stored in some hidden directories in the user's home directory on Linux. In addition, both configuration files and non-configuration files can be either ASCII or binary. This makes it difficult to identify configuration files by their locations or formats. Second, configuration files usually contain both configuration related data and non-configuration related data. It is thus more appropriate to score configuration files by the proportion of configuration related data they contain. Finally, the life time of data fragments in a configuration file is much longer than those in a non-configuration file. This indicates that configuration files change much more slowly than non-configuration files. This difference between configuration files and non-configuration files probably can be leveraged to identify configuration files.

Chapter 5

System Architecture and Implementation

5.1 The Design of Ocasta

5.1.1 System Model

Ocasta has two modes of operation. We assume that the system already has a versioning facility to record file versions in place. During regular system operation, Ocasta observes file system activity and assigns a similarity score to each file that the file system versions. When the versioning file system nears its space quota and needs to discard file versions to make additional room, it will start by discarding versions of files with the lowest similarity score first. In addition, in this mode, the resource requirements of Ocasta must be kept to a minimum since Ocasta is sharing compute resources with other tasks on the machine.

When the user discovers a misconfiguration and wishes to initiate recovery, she switches Ocasta into recovery mode. In recovery mode, the similarity scores are used by the user to identify files that contain configuration state. The user will begin by selecting the highest scoring files to try rolling back first, since these are the files that contain the largest amount of configuration state. Ocasta currently makes the simplifying assumption that configuration files can be independently rolled back and restricts its applicability to misconfigurations that can be solved by rolling back a single configuration file. A more complete model would also try rolling back combinations of configuration files simultaneously in order to find the most recent working state. In recovery mode, we assume Ocasta will have more resources available to it since this mode is run only when the user needs Ocasta to help recover from a misconfiguration.

Ocasta assigns similarity scores in two steps. First, Ocasta applies a set of filters to eliminate files that do not hold application state. These filters do not examine the contents of the files and thus, are very efficient. Then, Ocasta computes a similarity metric on the remaining application state files to rank them by the amount of configuration state they contain, thus separating configuration files from files that contain mostly task-dependent or time-dependent state.

5.1.2 Filters

Ocasta uses three filters to screen out files that do not hold application state. By our definition, application state must persist for the lifetime of the application. The first filter removes files that have been deleted by the application, which, as a result, cannot hold state that persists for the application's lifetime. The second filter requires the file to be read before it is written by an application, implying that the state held in that file persists across executions of the application. Any file that does not fall in this category only contains temporary state for a particular execution since there was no information flow from previous executions. Examples of files removed by these filters include lock files and files used as unidirectional communication channels between two processes. There is one exception to this case – many applications will write a default configuration file the first time they are run if the old configuration file is deleted or if there has been a major update. Thus, this filter requires a file to be written before it is read more than 20% of the time it is opened before it is removed from consideration. These two filters eliminate files that only hold temporary execution state and not application state. The final filter removes user data files from consideration. For the applications we analyzed in this paper, a simple filter that removes files that are in user home directories and do not have a directory starting with "." in their path is able to remove all user data files. In practice though,

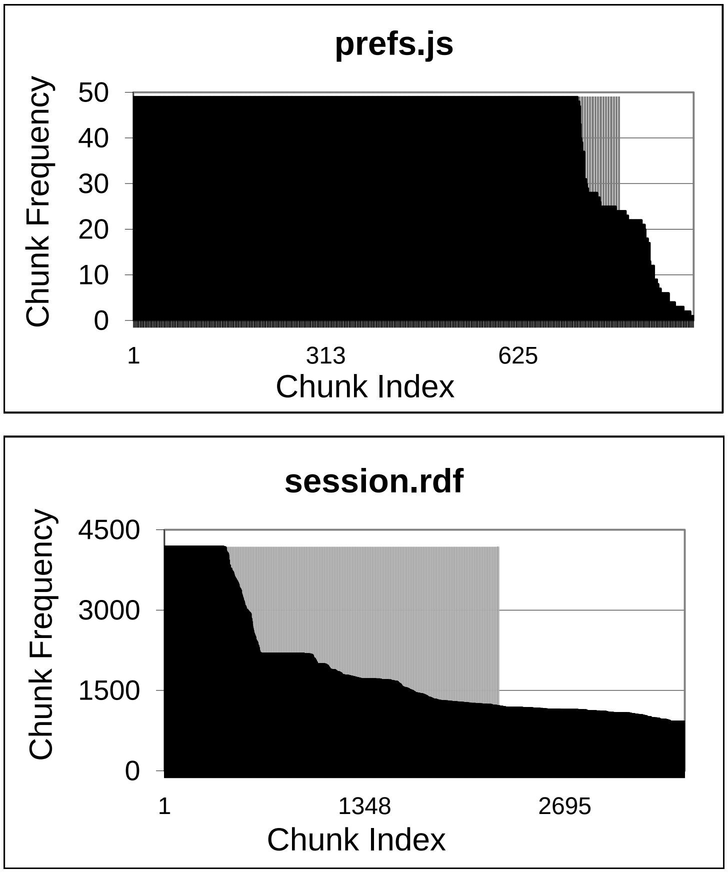

Figure 5.1: Similarity for *prefs.js* (96%) and *session.rdf* (55%). The shaded region denotes $v \times n$ in Equation 5.1. Note that the lower graph has been truncated, and actually has 17287 unique chunk values.

a more sophisticated, domain specific filter would likely be necessary.

5.1.3 Similarity Metric

After the filters have been applied, Ocasta separates configuration state from task-dependent and time-dependent state by computing the similarity among versions of a file. The key insight behind this heuristic is that configuration state changes much more slowly than both task-dependent and time-dependent state. This is because changing tasks and timed events occur regularly in an application's use, but changing configuration is a rare event. Thus, to separate configuration state from other application state, Ocasta identifies file chunks that remain unchanged across versions of a file. A file that contains a high proportion of constant chunks is likely to contain a large amount of configuration state, and thus will be identified as a configuration file.

To correctly capture this similarity, our algorithm must be tolerant to insertions and deletions, since fields in configuration files may change in size. In addition, we have observed that applications tend to read and parse an entire configuration file, modify it while in memory and then periodically write out the entire configuration file. Because of this, independent portions of the file may also be arbitrarily reordered. For example, the Firefox configuration file *pluginreg.dat* contains information about plugins in a list format. The order that the plugins appear in the list has no meaning and changes arbitrarily among versions.

To meet these requirements, we use an algorithm similar to the one used by LBFS to find similar chunks of data for data compression in a distributed file system [35]. Files are divided into variable sized chunks by selecting certain offsets within the file to be *anchors*. The location of the anchors is determined by computing Rabin fingerprints [42] on an 8 byte sliding window over the entire file. We then randomly select 1/16 of the values to be anchors, giving an expected chunk size of 16 bytes. Since the anchors are based on the file contents instead of fixed sized offsets, an insertion or deletion will only affect at most two chunks and leave the remaining chunks in the file unchanged. We note that our window and chunk sizes are considerably smaller than those used in LBFS because our goal is to identify tokens that appear across a file's lifetime, not to optimize the use of network bandwidth. Thus, the chunk size is chosen to be on the order of the expected storage required for a single configuration setting.

We call the contents of each chunk the *chunk value* and compute the frequency of each chunk value over all versions of a file.

A file with high similarity will have many chunk values that appear in a large portion of its versions. Conversely, a file with low similarity will have many chunk values that appear in only a few of its versions. To assign a numeric score to a file's similarity across versions we compute the following ratio:

$$Similarity = \frac{\sum_{i=0}^{n} c_i}{v \times n} \tag{5.1}$$

where n is the average number of unique chunks in each version of the file, c_i is the number of occurrences of the ith chunk when the chunks are sorted in descending order by frequency of occurrence across the different versions, and v is the number of versions of the file recorded. Intuitively, similarity is the ratio between the sum of the occurrences of the n most frequent chunks in the file, over a similar sized file that does not change at all – i.e. each chunk appears in every version. To illustrate, Figure 5.1 compares the chunk distribution for a configuration file, *prefs.js* and a non-configuration file, *session.rdf*. A large proportion of chunks in *prefs.js* appear in every version of the file, so the rectangle representing $v \times n$ is almost entirely covered (96%) by the histogram. On the other hand, *session.rdf* experiences much lower similarity among its versions, and has a long tail of low frequency chunks extending beyond the shaded rectangle. As a result, the histogram covers a much smaller portion of the rectangle (55%).

5.2 Implementation

5.2.1 Computing Similarity

Our prototype consists of a tool to compute similarity of a file's versions and a versioning file system, which is implemented by the redo logs maintained by the kernel module that we used in Section 3.3 to collect our user traces. When running in regular system operation mode, it first applies the three filters to see which file's similarity should be computed, and assigns a similarity score of zero to those files that do not pass the filters. For each file that is not removed by the filters, the similarity computation tool extracts each version of the file from the redo log maintained by the versioning file system and computes overlapping Rabin fingerprints, which it uses to select anchors to delimit file chunks. Because the chunk values have varying lengths, a hash function is applied to them and the result is then inserted into a hash table that maintains a running count of how many times a chunk value has appeared. This process is repeated for each

```
function TRIGGER SIMILARITY
    $i \leftarrow 1$
    $v \leftarrow$ next_version()                                ▷ Get the first version
    while $i < M$ do
        if more_versions() then
            $x \leftarrow$ next_version()
        else
            return NULL                                      ▷ Not enough versions yet
        if timestamp($x$) $-$ timestamp($v$) $> T$ then
            $v \leftarrow x$
            $i \leftarrow i + 1$
    return $v$
```

Figure 5.2: Function for determining when to perform a similarity measurement. If there are not enough versions to get M sampling periods of length T, the function returns NULL.

file version, thus computing how many times each chunk values appears across all versions of the file. If a chunk value appears multiple times in any particular file version, it is only counted once so that the count of any hash value cannot exceed the number of file versions. Finally, these counts are used as described by Equation 5.1 and the score is assigned to the file. Since configuration files do not become non-configuration files or vice-versa, Ocasta only runs the tool once per file, thus keeping the performance requirements of Ocasta low. Once a file is assigned a score it maintains that score until it is deleted from the file system.

5.2.2 Triggering Similarity Measurements

While Ocasta's similarity algorithm is effective at identifying configuration files, an important requirement is that Ocasta must be able to provide online measurements to a versioning file system efficiently. During regular operation, Ocasta only invokes the similarity tool when there are idle cycles available on the processor. This is done to minimize the impact of Ocasta on the usability of the user's machine. While the number of cycles allocated to Ocasta will depend on the load on the machine, we assume that Ocasta must have very meager resource requirements, and must be tolerant to bursty resource availability. As a result, Ocasta cannot take a measurement after every file modification. Despite this, we expect Ocasta to produce accurate measurements in a timely manner so that the versioning file system can make informed decisions about which files to discard when it hits its space quota. Thus, to make the best use of the few cycles allocated to it, Ocasta must trigger a similarity measurement only on files that have had enough activity to produce an accurate similarity score.

Finding the correct time to take such a measurement is an important factor in the effectiveness of Ocasta. On the one hand, waiting too long to take a measurement carries several risks. First, the versioning file system may mistakenly continue to version a non-configuration file and instead discard a configuration file's versions because Ocasta has not given it the information necessary to make a correct decision. Second, Ocasta's run time increases with the number of file versions it must process. Finally, all files that are modified exhibit decreasing similarity over time, but the rate of similarity decrease in configuration files is much lower than in non-configuration files. However, waiting too long to compute a similarity measurement of a configuration file will give it an artificially low score, causing it to be incorrectly discarded.

On the other hand, like all statistical analysis, Ocasta can produce noisy results if run over too little data. As a result, taking a measurement too early can give an inaccurate score due transient events on the file. For example, some configuration files experience a large number of changes when they are first created, and then eventually settle into a more representative pattern of low activity. Taking a similarity measurement during the transient at the beginning would produce an artificially low similarity score. Thus, it is critical that Ocasta picks the correct time to take a measurement of a file and that the time of measurement be normalized across all files.

Our initial thought was that there exists a fixed number of versions after which a similarity measurement would return an accurate result. This turned out to be false because the rate at which new versions are created varies greatly from file to file. As a result, using a fixed number of versions would give files that are modified very often an artificially high similarity score because no task-dependent or time-dependent events will have occurred between versions created extremely close together. In other words, when files are rapidly modified, the changes between each version are small, making the file have an apparently high similarity.

This observation made us realize that an algorithm to pick the correct measurement time must take into account not only the number of versions of a file that exist, but also the time over which the versions are created. Such an algorithm should discount versions created close together since it is unlikely that there was an intervening task-dependent or time-dependent event and reward versions created farther apart in time. To do this, we defer computing a similarity measurement until a file has had M modifications, each of which are at least T hours apart. In other words, there must be at least M non-overlapping time periods of length T that contain at least one new file version.

The point at which Ocasta computes similarity for a file is called the *trigger point* and is found by using the *trigger function* described in Figure 5.2. We call T the sampling period and M the sample length. The trigger function will return the version at which a measurement should be taken if enough versions of the file have been generated in the versioning file system. If there have not been enough versions, then the function returns NULL. Ideally, the sample period should be long enough to contain one task-dependent or time-dependent event. Thus, using file versions across a number of sample periods will contain enough task-dependent changes that non-configuration files that contain task-dependent state will produce a lower similarity score than configuration files that contain mainly configuration state. In practice, we have found that a sample period of 3 hours and a sample length of 12 works fairly well.

In recovery mode, the user may need the score of a file that has not reached its trigger point yet, and thus has not had a similarity measurement performed on it. In this case, Ocasta computes the similarity on all versions up to the point that the recovery is taking place and uses that similarity score instead. This score is only used temporarily during recovery as a best approximation of what the trigger point score would be. It is discarded after the user returns to regular operation and the file must still reach its trigger point before Ocasta will assign a permanent score to it.

5.2.3 Improving Performance

During regular operation, we found that some files took a long time to compute their similarity scores. However, in this mode, Ocasta must produce similarity scores quickly and without the need for a large amount of resources. There are a number of operations that scale with the aggregate size of the file versions used in the computation. For example, the time to extract file versions, the number of Rabin fingerprints to compute and the number of hash computations all increase with the number of versions and the size of the file. Another source of overhead is the the size of the chunk value hash table. For a file with extremely low similarity and large aggregate size of file versions, the number of unique chunk values can become very large. This added memory pressure results in thrashing on the machine and greatly degrades performance.

As a result, the overhead of the similarity computation is determined by the file's similarity, the size of the file and the number of versions used in the similarity computation. Since we can't control the size or similarity of the file, we must reduce the number of versions used to make these files practical to measure. We employ the sampling periods used in the triggering function and use only the first sample of each sample period for the similarity computation, discarding the rest of the versions within the sampling period. In this way, the number of versions used in the similarity computation is bounded by M, which is 12 in our prototype. The intuition behind why this approximation works is that even if the rate of change of a file is significantly faster than the sampling period, the samples will still capture the cumulative changes and produce a similarity score close to the one that would have been achieved by sampling all versions. The only case where the scores may significantly differ is if a file was being modified in a cyclic way, and eventually returned back to a state in a previous version.

We did not observe any files that displayed this behavior. We will compare the scores derived using sampling and using all versions in Section 6.

Chapter 6

Evaluation

6.1 Identifying Configuration Files

We now evaluate Ocasta's ability to identify configuration files. Several metrics are of importance for Ocasta. One such metric is the percentage of non-configuration file versions Ocasta eliminates. This number tells us how many fewer file rollbacks a user must perform when trying to find the most recently working application configuration. As a result, it can be used as an indicator of how much effort Ocasta will save the user during recovery. Another useful metric is the amount of versioning space Ocasta has saved. This saves the versioning file system from versioning files unnecessarily, thus either reducing the space overhead of the versioning file system or allowing it to maintain longer file histories. Finally, we would like to evaluate Ocasta's accuracy by measuring Ocasta's true positive and false positives when identifying files. We define a true positive as when Ocasta correctly identifies a configuration file, and a false positive as when Ocasta incorrectly identifies a non-configuration file as a configuration file. We do not evaluate the overhead of the file system versioning in Ocasta because we have not made any effort to optimize the kernel module that was performing the file versioning for Ocasta. The overheads of optimized versioning file systems have been well studied in the literature [13, 34, 43].

Trace	Application	Versioned Files	Filters			Passed
			Persistent	Read→Write	User file	Filter
1	Firefox	507	228	265	1	13
	GNOME	392	9	363	0	20
	Flash	17	12	0	0	5
	VMware	189	105	7	71	6
2	Firefox	568	449	86	3	30
	GNOME	493	316	151	0	26
	Flash	35	24	5	0	6
	JEdit	73	12	41	14	6
	Amarok	89	29	51	4	5

Table 6.1: Filter results. This table gives the number of files removed by each of the filters. The last column indicates how many files from each application had the similarity computation applied to them.

We evaluate each of the three implementation options described in Section 5.2. First, we evaluate the accuracy of Ocasta when applied to all file versions in our two traces. We then compare the accuracy of this option to Ocasta using the trigger function described in Section 5.2.2 and then finally the option using the sampling performance enhancement described in Section 5.2.3.

6.2 Similarity Evaluation

We began by using Ocasta to measure similarity across all versions of all files in our trace. As described in Section 5.1.1, Ocasta first applies a set of filters to remove non-application state files. We found that these filters play an important role in Ocasta since a large portion of versioned files are actually removed from further consideration by these filters. This is

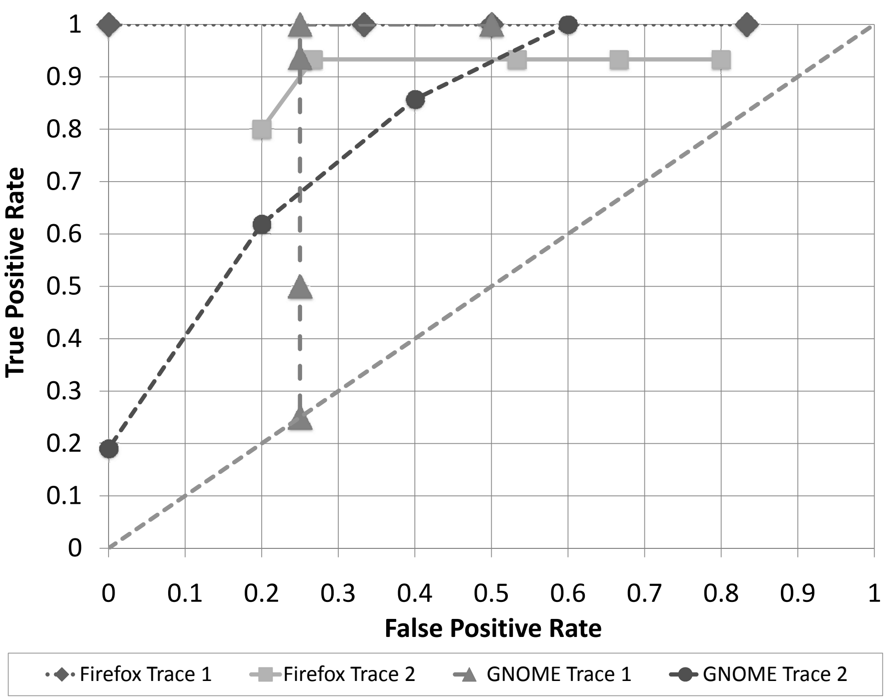

Figure 6.1: Receiver Operating Characteristic curve for Ocasta's Similarity Metric. The points from left to right in each curve represent similarity thresholds of 95%, 90%, 80%, 70% and 60%. We note that Firefox in Trace 1 actually experiences a 100% true positive rate at all thresholds because all Firefox configuration files scored above 95% in that trace.

beneficial because the filters are far more efficient than the similarity tool. Table 6.1 shows the number of files filtered out for each application by each filter. The passed filter column indicates the number of application state files that are not removed by the filter, which will have the similarity calculation applied to them. We verified that none of the files removed by the filters were configuration files. As we can see, the majority of files modified by each application actually consist of temporary and non-persistent files.

Next the tool computes the similarity scores of all the files that passed the filters. Most configuration files scored greater than the majority of non-configuration files. To illustrate, Figure 6.1 shows a receiver operating characteristic (ROC) curve, which shows what the accuracy of Ocasta would be at similarity thresholds of 60%, 70%, 80%, 90% and 95%. Files with scores above the threshold are identified as configuration files and files below the threshold are identified as non-configuration files. The ROC curve plots the true positive rate versus the false positive rate. The true positive rate is defined as the number of true positives over the actual number of configuration files, while the false positive rate is the number of false positives over the number of non-configuration files that were not removed by the filters. For readability, we only show curves for the four applications which have the most versioned files.

In reality, Ocasta's versioning file system does not use an explicit threshold to determine whether a file should be versioned or not. Rather, this threshold is determined implicitly by how much space is available for versioning – the versioning file system simply discards the lowest scoring file when it is low on space. As a result, the more storage is allocated, the lower the implied threshold, although the precise relationship between version storage and similarity threshold will depend on the amount of versioning activity and the size of the files currently in the version store. From the curve we see that as the point moves to the right along the curves, which means threshold is becoming lower, both the false positive and false negative rates will rise. The closer a point is to the upper left corner (100% true positives with no false positive) the better the operating point. The important thing to note is that the curves are all well above the "line of no discrimination" ($x = y$), indicating that Ocasta generally has a higher true positive rate than false positive rate. By computing the Euclidean distance between each of the operating points and the upper left corner, we determine that the optimal operating point for the files in our trace is a similarity threshold of 82%. 90% of all configuration have similarity scores above this threshold and the false positive rate at this operation point is 35%. At this point Ocasta will eliminate 85% of all non-configuration file versions and save 77% of versioning space in the redo logs. We note that with a threshold of 55% Ocasta is capable of identifying all configuration files while still correctly eliminating 30% of non-configuration files. At this operating point, Ocasta suffers only 33 false positives across both traces and is able to eliminate roughly 66% of non-configuration file versions from its redo log.

To evaluate the number of versions Ocasta can eliminate and space Ocasta can save, we will

Trace	Application	TP	FN	FP	Versions eliminated	(%)	Space Saved (MB)	(%)
1	Firefox	7	0	3	258505/289356	89.3	1025/1151	89.1
	GNOME	15	1	1	1737/1743	99.7	62.8/62.8	100
	Flash	1	0	0	123/123	100	0.0138/0.0138	100
	VMware	4	2	0	2766/2766	100	32.0/32.0	100
2	Firefox	14	1	8	422066/516984	81.6	4733/5060	93.5
	GNOME	18	3	2	17056/17090	99.8	1267/1267	100
	Flash	1	0	2	1190/1980	60.1	0.784/1.46	53.7
	JEdit	2	0	1	4940/4956	99.7	51.5/51.5	99.8
	Amarok	3	0	1	4735/4742	99.9	194/194	100

Table 6.2: Performance of Ocasta over all versions in each trace. Here we use a similarity threshold of 80% to differentiate configuration files from non-configuration files. TP denotes the number of true positives, FN, the number of false negatives (configuration files that were not correctly identified) and FP, the number of false positives. The Versions Eliminated column gives the number of non-configuration file versions eliminated over the total number of non-configuration file versions. Similarly, the Space Saved column gives the amount of versioning space in MB that was eliminated over the total potential space that is used to version non-configuration files.

use the 80% similarity threshold as our assumed operating point. Table 6.2 gives the number of true positives, the number of false positives and the number of false negatives (configuration files wrongly identified as non-configuration files). To evaluate the amount of effort the user saves during rollback, we provide the number of non-configuration file versions eliminated over the total number of non-configuration file versions. The amount of space saved is given by the amount of non-configuration file space saved in the redo logs over the total space used to version non-configuration files. Any of the versions eliminated or space that was not saved is a result of false positives. We can see that Ocasta will help the user tremendously during recovery for a majority of the applications by correctly identifying all non-configuration files and eliminating them from consideration during recovery, leaving only actual configuration files for the user to try to recover with. Similarly, nearly all the versioning space that would have been wasted on non-configuration files is saved. The only exception was the Flash application in Trace 2. Here, the vast majority of all versions not eliminated were the result of a single false positive file, *clearspring.sol*, which had 787 versions and a similarity score of 87%.

Only Firefox had significant numbers of false positives. Of these, two files, *cookies.sqlite* and *places.sqlite*, account for 99.9% of the misidentified versions, which were not eliminated as a result. These files have a common characteristic – they are histories of user activity and behave as a slowly changing cache for that activity. As a result, even though the entries in these files are task-dependent, some can remain in the file for a long time either because they are used frequently or there is insufficient activity in the application to evict them. We note that *urlclassifier3.sqlite* and *pluginreg.dat*, the two problematic configuration files described in Appendix A, were both correctly identified by Ocasta.

Both false negatives in VMware were in a file that VMware uses to store the user's favorite VMs. While this is configuration state, VMware also uses this file to store a large amount of task-dependent state, causing the file to have low similarity. We note that, with the exception of *clearspring.sol*, the false positive in the Flash application, many of the false positives have negligible impact on the number of versions and the amount of space saved. This is because the false positives often turned out to be short files with few versions. We will discuss the reasons for this in Section 6.5.

6.3 Trigger Function Evaluation

To evaluate the effectiveness of the trigger function, we modify our similarity measurement tool to only compute similarity for file versions up to the file's trigger point and compare its accuracy at identifying configuration files with the original tool that uses all versions available

Trace	Application	All Versions		Trigger		Sampling	
		TP	FP	TP	FP	TP	FP
1	Firefox	7/7	3/6	7/7	**4/6**	7/7	**4/6**
	GNOME	15/16	1/4	15/16	1/4	15/16	1/4
	Flash	1/1	0/4	1/1	0/4	1/1	0/4
	VMware	4/6	0/0	4/6	0/0	4/6	0/0
2	Firefox	14/15	8/15	14/15	**9/15**	14/15	**8/15**
	GNOME	18/21	2/5	**19/21**	2/5	**21/21**	2/5
	Flash	1/1	2/5	1/1	2/5	1/1	2/5
	JEdit	2/2	0/4	2/2	0/4	2/2	0/4
	Amarok	3/3	0/2	3/3	0/2	3/3	0/2

Table 6.3: Evaluation of Trigger Function and Sampling. We show the effects of both mechanisms on the accuracy of Ocasta. In most cases, these optimizations had no effects on the accuracy. In cases where there was an effect, the change is highlighted in bold.

in the trace. The results are given in Table 6.3. When a file's trigger point does not occur before the end of the trace, we use all the versions gathered in the trace. This simulates Ocasta's recovery mode behavior, which is to use all available versions for files that have not reached their trigger points. On average, using a trigger point increases the similarity score of a file by 2±6%. This was expected since most files have a decreasing similarity score over time, so taking a measurement earlier will increase a file's similarity score on average. As we can see from the results, this has a negligible effect on the effectiveness of Ocasta. In two cases, a non-configuration file had its similarity score increased to exceed the 80% threshold and become a false positive. In both these cases, the file was the user's Firefox cookie jar, which increased from 79% to 98% in Trace 1 and from 73% to 85% in Trace 2. Increasing similarity scores can also have positive results – the similarity score of one GNOME configuration file was increased from 79% to 83% and went from a false negative to a true positive.

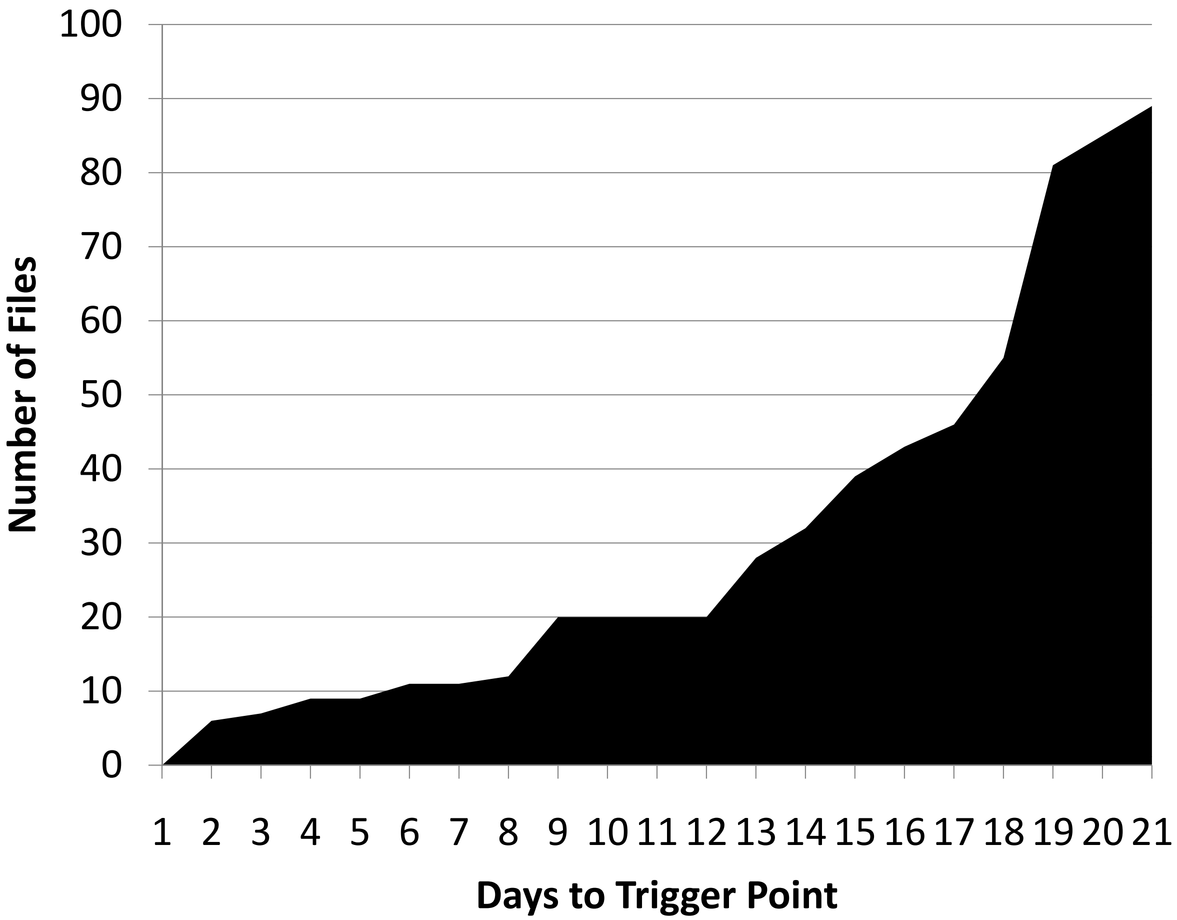

Figure 6.2: Cumulative distribution of time for files to reach their trigger points in Trace 1.

The benefit of the trigger function is that Ocasta can provide results to the versioning file system much earlier. Of the 96239 versioned files in both traces, 149 files reached their trigger point before the end of the trace. The cumulative distribution of times to reach the trigger point for both traces is given in Figure 6.2 and Figure 6.3. In Trace 2, which had more activity, many more files were able to reach their trigger points with several days of activity while in Trace 1, which had less activity, half the files took 18 or more days. While the time for the trigger point to occur does depend heavily on how much the application that accesses the file is used, we take these numbers to show that Ocasta can get accurate measurements for a fair number of files within the scope of several weeks.

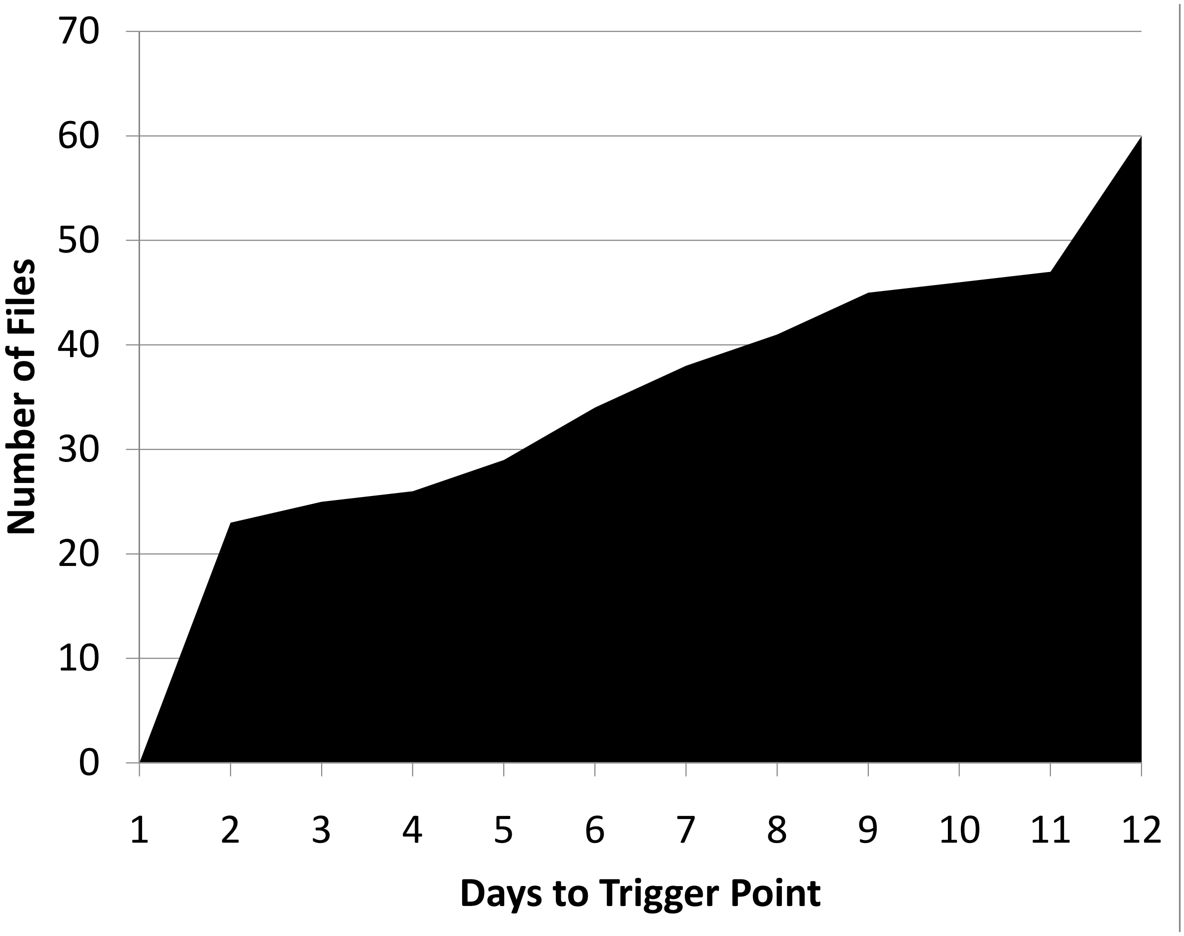

Figure 6.3: Cumulative distribution of time for files to reach their trigger points in Trace 2.

6.4 Sampling Evaluation

While sampling reduces the resource requirements of Ocasta, we also wanted to measure how sampling perturbs Ocasta's similarity score as compared to computing the score with all versions up to the trigger point. Table 6.3 also shows the differences in performance when we configure Ocasta to only use one file version from each sampling period up to the trigger point versus using all versions up to the trigger point. Sampling perturbs the similarity score of a file by 2±7% on average and as a result has essentially no measurable affect on the accuracy. Anecdotally, two configuration files, each of which had a similarity score of 78%-79% when computed using all samples up to the trigger point, had their similarity scores increased to 80%-84% by sampling. One false positive introduced by using the trigger function also decreased below the threshold, becoming a true negative again. Using sampling periods decreases the time required to analyze the files by 86% overall. In particular, some of the larger files saw the time to compute their similarity score drop by over 90%.

6.5 Discussion

Because Ocasta's similarity metric relies on heuristics, it can make mistakes when these heuristics fail. As we have seen, some non-configuration files may also exhibit high similarity by coincidence – for example they may record user histories. However, the impact of such errors is lessened by two properties of Ocasta. First, Ocasta is more likely to classify a non-configuration file as a configuration file than the reverse. This is because while there is non-configuration state that can change slowly, we have not observed any configuration state that changes frequently. We believe that such a situation would only arise if a user changed configurations as often as she changed tasks, or a bug in the application caused configurations to spontaneously change on their own – both are exceptional circumstances. Hence, a misidentification will cause more than necessary disk space to be used and extra effort during recovery, but will not result in important file versions getting discarded. Second, because Ocasta relies on identifying common chunks in file contents over time, Ocasta can misclassify files when there have not been enough versions, or when the file is very small. In such cases, results can have quite a bit of error associated with them. Fortunately, it is not expensive to version small files with few versions, so one solution may be to augment the file system's versioning policy to retain such files regardless of their similarity score.

Chapter 7

Conclusions

Without knowledge of which files contain configuration state, trying to revert a misconfigured application back to a working state is a daunting task. In addition to configuration state, applications maintain a variety of task-dependent and time-dependent application state strewn across files, all of which are difficult to distinguish from configuration state. Ocasta solves this problem by using a statistical approach that measures the similarity of a file over its versions. We find that with 3 simple filters that remove non-persistent files, temporary files and user data files, and a similarity metric that measures how chunks persist over the lifetime of a file, Ocasta is able to differentiate configuration files from non-configuration files. By identifying configuration files, Ocasta reduces the number of files the user must examine for rollback by one to two orders of magnitude.

Ocasta can operate at various similarity thresholds, which are implicitly determine by the amount of versioning storage space. Depending on the threshold, Ocasta can identify between 60-100% of configuration files for most applications, while suffering a false positive rate of 20-60%. With a threshold of 80%, 90% of configuration files are correctly identified and 60% of non-configuration files are eliminated, removing an aggregate of 713129 file versions and 7GB of storage across two traces. Ocasta's resource requirements are low due to two mechanisms. A trigger point function that determines an optimal measurement point by ensuring that a file's versions have covered a sufficiently long period of time, allows Ocasta to take only one measurement per file. Further, the cost of similarity computation is reduced by using a sampling method that performs the computation on fewer versions. Both of these enhancements have negligible effects on the accuracy of Ocasta, so we believe they should be used continuously during regular operation.

Bibliography

[1] Bhavish Aggarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N. Padmanabhan, and Geoffrey M. Voelker. Netprints: diagnosing home network misconfigurations using shared knowledge. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 349–364, Berkeley, CA, USA, 2009. USENIX Association.

[2] P Anderson and A Scobie. LCFG: The next generation. In *Proceedings of the UKUUG Winter Conference*, 2002.

[3] Paul Anderson, Patrick Goldsack, and Jim Paterson. Smartfrog meets LCFG: Autonomous reconfiguration with central policy control. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 213–222, Berkeley, CA, USA, 2003. USENIX Association.

[4] Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 281–286, Berkeley, CA, USA, 2008. USENIX Association.

[5] Mary Baker and Mark Sullivan. The recovery box: Using fast recovery to provide high availability in the unix environment. In *In Proceedings USENIX Summer Conference*, pages 31–43, 1992.

[6] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, 14(1):80–107, 1996.

[7] Aaron B. Brown and David A. Patterson. Rewind, repair, replay: three r's to dependability. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 70–77, New York, NY, USA, 2002. ACM.

[8] Aaron B. Brown and David A. Patterson. Undo for operators: building an undoable e-mail store. In *ATEC '03: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[9] M. Burgess. Cfengine: a site configuration engine. *USENIX Computing systems 8*, 3, 1995.

[10] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

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

[12] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, pages 312–314. The MIT Press, 1998.

[13] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. Wayback: A user-level versioning file system for Linux. In *Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track*, pages 19–28, July 2004.

[14] Olivier Crameri, Nikola Knezevic, Dejan Kostic, Ricardo Bianchini, and Willy Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 221–236, New York, NY, USA, 2007. ACM.

[15] Mszeredi Dzsekijo. Filesystem in Userspace. http://fuse.sourceforge.net, 2003.

[16] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[17] George Forman, Kave Eshghi, and Stephane Chiocchetti. Finding similar files in large document repositories. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD)*, pages 394–400, August 2005.

[18] Ashvin Goel, Kamran Farhadi, Kenneth Po, and Wu-chang Feng. Reconstructing system state for intrusion analysis. *SIGOPS Oper. Syst. Rev.*, 42(3):21–28, 2008.

[19] Jim 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, 1986.

[20] Jim Gray. A census of tandem system availability between 1985 and 1990. volume 39, pages 409–418. IEEE, 1990.

[21] Apple Inc. What is Mac OS X - Time Machine. http://www.apple.com/macosx/what-is-macosx/time-machine.html. Last accessed: 9/1/2009.

[22] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding customer problem troubleshooting from storage system logs. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 43–56, Berkeley, CA, USA, 2009. USENIX Association.

[23] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *Proc. of the 13th USENIX Security Symposium*, pages 271–286, August 2004.

[24] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC '05: Proceedings of the annual confer-*

ence on USENIX Annual Technical Conference, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[25] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[26] Nick Kolettis and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, page 381, Washington, DC, USA, 1995. IEEE Computer Society.

[27] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[28] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery domains: an organizing principle for recoverable operating systems. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 49–60, New York, NY, USA, 2009. ACM.

[29] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

[30] E. Marcus and H. Stern. *Blueprints for High Availability*. John Willey & Sons, 2000.

[31] James Mickens, Martin Szummer, and Dushyanth Narayanan. Snitch: interactive decision trees for troubleshooting misconfigurations. In *SYSML'07: Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[32] Microsoft Corporation. Explore the features: Shadow copy. http://www.microsoft.com/windows/windows-vista/features/shadow-copy.aspx. Last accessed: 9/1/2009.

[33] Kiran-Kumar Muniswamy-Reddy and David A. Holland. Causality-based versioning. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 15–28, Berkeley, CA, USA, 2009. USENIX Association.

[34] Kiran-Kumar Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. pages 115–128, March 2004.

[35] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, October 2001.

[36] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[37] Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and validating database system administration. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[38] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[39] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. In *OSDI '02: Proceedings*

of the 5th symposium on Operating systems design and implementation, pages 361–376, New York, NY, USA, 2002. ACM.

[40] Zachary Peterson and Randal Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, 2005.

[41] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, New York, NY, USA, 2005. ACM.

[42] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computer Technology, Harvard University, 1981.

[43] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, December 1999.

[44] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 37–48, New York, NY, USA, 2009. ACM.

[45] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 45–60, December 2004.

[46] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *FAST '03: Proceedings of the 2nd USENIX*

Conference on File and Storage Technologies, pages 43–58, Berkeley, CA, USA, 2003. USENIX Association.

[47] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 237–250, October 2007.

[48] Ya-Yunn Su and Jason Flinn. Automatically generating predicates and solutions for configuration troubleshooting. In *ATC'09: USENIX 2009 Annual Technical Conference on Annual Technical Conference*, Berkeley, CA, USA, 2009. USENIX Association.

[49] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, 1991.

[50] Kalyanaraman Vaidyanathan and Kishor S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Trans. Dependable Secur. Comput.*, 2(2):124–137, 2005. Member-Vaidyanathan, Kalyanaraman and Fellow-Trivedi, Kishor S.

[51] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 245–258, December 2004.

[52] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 159–172, Berkeley, CA, USA, 2003. USENIX Association.

[53] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 77–90, December 2004.

[54] 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. volume 40, pages 375–388, New York, NY, USA, 2006. ACM.

[55] Wei Zheng, Ricardo Bianchini, and Thu D. Nguyen. Automatic configuration of internet services. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 219–229, New York, NY, USA, 2007. ACM.

Appendix A

Application Information

Firefox is a full featured web browser. While Firefox spreads its configuration over a large number of files, two of those files[1], *urlclassifier3.sqlite*, which contains a blacklist of phishing sites and *pluginreg.dat*, which records associations of plugins with MIME types account for over 95% of the configuration file versions in Trace 1 and over 99% of the configuration file versions in Trace 2. Strangely, we observed that Firefox frequently changes the order in which the MIME associations appeared in the file, even though the order of appearance has no effect on their semantic meaning – an example of the arbitrary and senseless behavior that all applications exhibit to some degree. While the meaning of the file contents has not changed, these updates cause the file contents to change and thus results in a new version being created. Some examples of Firefox non-configuration files include *sessionstore.js*, which records task-dependent session information, which is used to reinitialize the state of the browser after a crash, as well as files that store task-dependent browsing state such as a history of form field values and the user's cookie jar.

GNOME represents the GNOME desktop system, which includes session manager, window manager, power manager, screen saver, GConf configuration system, as well as GNOME applications come with the default GNOME desktop installtion. Instead of managing their own

[1] A reference for Firefox files can be found at `http://kb.mozillazine.org/Profile_folder_-_Firefox`

configuration settings, many GNOME applications choose to use the GConf configuration system to access, modify, and maintain their configurations. In both Trace 1 and Trace 2, all the GNOME configuration files were accessed and modified by GConf configuration system.

Flash represents the Macromedia Flash plugin. This plugin only has one global configuration file and caches many website specific settings. These website specific settings are updated less frequently than the global configuration file.

VMware workstation is a popular hypervisor. It has 3 config files and the user in Trace 1 used it under two users for a total of 6 configuration files in the trace. VMware creates a large number of temporary lock files and log files.

JEdit is a JAVA-based editor. It has two configuration files and several history files that store recently opened files and a history of user actions.

Amarok is an open-source music player that uses the KDE framework. It has two configuration files. It stores user listening habits and song rankings in an sqlite database file which we classify as a non-configuration file because its contents are largely task-dependent.