Replication-based Cyber Foraging and Automated Configuration Management

by

Ya-Yunn Su

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

Doctoral Committee:
Associate Professor Jason N. Flinn, Chair
Professor Peter M. Chen
Assistant Professor Zhuoqing Mao
Assistant Professor Clayton D. Scott

To my family.

ACKNOWLEDGEMENTS

Graduate school is a long journey and I am very thankful to many people who have helped me along the way. I would like to take this opportunity to express my sincere gratitude towards each individual who has helped me one way or the other.

First, I would like to thank my advisor, Jason Flinn. Your passion and dedication to research is a true inspiration for me. You have guided me to do research in a creative and disciplined manner. You have taught me the art of building system software and the skills to communicate precisely. Doing research is an endless learning process but you have built the foundation for me.

I would like to thank my thesis committee, Dr. Peter Chen, Dr. Morley Mao, and Dr. Clayton Scott. Your input and feedback were invaluable and have greatly helped to improve the quality of this work. I would also like to thank my mentors, Dr. Leo Luan and Danny Tan, for guiding me during my summer internship at IBM.

I would like to express my appreciation for my undergraduate advisor, Dr. Yeali Sun, for encouraging me to pursue a Ph.D. in the United States. Without you, I would not have been here.

Graduate school can be frustrating and stressful some times. I feel very lucky to have my office mates who have made my graduate school life so much more enjoyable. We had discussions on various subjects, ranging from technical issues, international affairs, to the economy. These discussions have broadened my views in so many different aspects in life. Many thanks to Manish Anand, Mona Attariyan, Brett Higgins, Edmund Nightingale, Jon Oberheide, Daniel Peek, Sushant Sinha, Kaushik Veeraraghavan, and Benjamin Wester.

Last, I would not be able to finish this thesis without the love and support from

my family. My dad has always been the role model of a dedicated researcher since I was little. My mom, my sister, and my brother for giving me the strength and comfort when I most needed them. I am thankful to Katharine Chang, who is like a sister to me. I have shared so many happy and sad times with you, and my graduate school would have been so much harder if I did not have you. I would like to especially thank David Liu. Your tremendous patience and support have helped me through the last mile of this "thesis marathon". Many thanks to my other friends from college and graduate school, including, but not limited to: Galen Chen, Jessie Chou, Mike Chu, Ashlesha Joshi, Jinyuan Li, Sing-Rong Li, Howard Tsai, Dreamfish Tsao, Pei-Chao Weng, and Weiting Yen.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

CHAPTER 1

Introduction

This thesis addresses two research problems. The first problem is how to run demanding applications on mobile computers. Mobile computers, such as smart phones and personal digital assistants (PDAs), have become more prevalent these days. However, the size of these computing devices constrain the computing resources, such as processing power and memory, they can carry. This thesis explores using publicly available compute servers to run demanding applications for mobile computers.

The second problem this thesis addresses is how to make configuration management easier for users. Users enjoy the flexibility of customizing their software applications. However, software programs are not self-contained; they depend on shared libraries, configuration data, and services. When a user makes configuration changes that cause software to function incorrectly, it is often difficult to troubleshoot the root cause of the configuration problem. In this thesis, we address this problem by first building tools to automate many of the configuration management tasks, and then developing heuristics to automatically generate test cases that can test if the configuration of a system state.

1.1 Improving performance for mobile computers through replicated cyber foraging

It is difficult to design demanding applications for mobile computers, such as smart phones and PDAs. On one hand, these mobile computers are portable and can be carried everywhere. On the other hand, their size and weight limits constrain their computing resources, such as processor, battery, and memory. Limited computing resources make it hard to run demanding applications, such as a language translator or a speech recognizer, on these mobile computers. Unfortunately, the interactive applications that are resource-intensive are often of highest value when used in mobile settings. For example, when a user travels to a foreign country with her smart phone, a language translator could help her ask for directions.

To provide portable mobile computers the ability to run demanding applications, we designed and built a *cyber foraging* infrastructure. Cyber foraging is the concept of dynamically augmenting the computing resources of a wireless mobile computer by exploiting wired hardware infrastructure [73]. Our cyber foraging infrastructure, Slingshot, replicates applications on both *surrogate computers* and the user's own home machine to improve performance for mobile computers. A surrogate computer is a public compute server that provides computing resources similar to how free wireless network access is provided. The replica running on a surrogate computer co-located with the mobile computer is more likely to provide a better response time than a replica running on user's home machine. However, the replica on the home machine provides a safe repository for application state when the surrogate computer crashes or becomes unavailable.

Slingshot makes surrogate computers easy to manage by installing minimal software programs and keeping application state only during the period a user is actively using it. It encapsulates each application in a virtual machine so that surrogate computer owners do not need to pre-install software packages required by an application. Slingshot replicates user changes to the home machine so the user need not worry about application state loss when surrogate computers become unavailable.

We demonstrated the usefulness of Slingshot by adapting two applications for mobile computers, a speech recognizer and remote desktop software. In the evaluation in Section 2.5, we show that with a typical workload in a wireless hotspot network environment, Slingshot improves response time by 2.6 times when compared to remote execution on the home machine.

1.2 Improving software configurations through automated configuration management

This thesis next examines the problem of making configuration management easier. Managing software configurations for general purpose computers is difficult because software programs are complex. Software programs depend on environment resources, such as shared libraries and environment variables, and interact with other software programs through many different types of channels. All of this complexity means that whenever a user changes the system state, that change might affect a software program in an unexpected way.

To make software programs easier to manage, we first built tools that can automate many configuration management tasks. Our tool, called AutoBash, can automate troubleshooting misconfigurations and test if the system is in a healthy state. AutoBash uses *predicates* to automatically test the health of a system state. Predicates are like test cases for software testing, and they return true when the system is in a healthy state and false otherwise. When predicates do not exist, users have to manually write them. Manually writing predicates is a burden on users because writing predicates requires a lot of user effort and expert knowledge. To remove the burden of manually writing predicates, we next developed heuristics to automatically generate predicates by inferring them from traces of users fixing specific configuration problems.

1.2.1 AutoBash: tools to automate configuration management

Managing configuration is a task that all users with different system administration skills need to perform. To better meet these different needs, AutoBash has three different modes: observation, replay, and health monitoring. AutoBash's observation mode is designed for system administrators or expert users who would like to manage configurations for their computers themselves. It is a command-line interface that records users' input commands as they examine the system state and make state changes. After a user types in a command, AutoBash automatically runs predicates to test if the system is in a healthy state. AutoBash's observation mode also provides rollback capability for each command so that the user can safely explore different ways of changing the system state.

AutoBash's replay mode is designed for users who are not experienced in configuring their computers. When the user needs to fix a configuration problem, AutoBash's replay mode first takes potential *solutions* and predicates as input and then tries to find a solution that fixes a problem exemplified by the predicates. A solution is a set of actions that transforms the system state from an unhealthy state in which one or more predicates return false to a healthy state in which all predicates return true. As AutoBash sees more misconfiguration instances, it has more solutions upon which to draw. If AutoBash cannot automatically find a solution, the user must fall back to AutoBash's observation mode to fix the configuration problem manually.

Finally, AutoBash's health monitoring mode periodically runs predicates to check if the system is in a healthy state. If a configuration problem is detected, AutoBash's health monitoring mode can be set to invoke its replay mode to find a solution.

AutoBash uses *OS-level speculative execution* provided by Speculator [62] to speculatively try a solution or execute a predicate and to safely roll any state changes back when a solution does not fix a configuration problem. As the number of applications increases, AutoBash needs to execute more predicates to determine if a solution fixes a configuration problem. This could significantly increase the time to find a solution.

AutoBash uses *causality tracking* to reduce the number of predicates that need to be tested for each potential solution by only executing predicates on which a solution has causal effects.

We evaluated three different applications with AutoBash's replay mode: the Concurrent Versions System (CVS) [1], the GCC cross-compiler [2], and the Apache Web server [7]. For each application, we identified ten common configuration problems by searching through on-line forums, FAQs, and mailing lists. For each configuration problem, we created a script that injected the problem and a solution script that fixed the problem. We ran AutoBash's replay mode to search for a solution. We compared the time it takes AutoBash's replay mode to find a solution for three different versions: the first version uses neither speculative execution nor causality tracking, the second uses only speculative execution, and the last one uses both. The results show that speculative execution adds overhead up to 3% but provides the safety of rollback. The version with both speculative execution and causal tracking outperforms the version without either by an average of 35%.

1.2.2 Inferring predicates and solutions from user traces

Predicates play an important role in AutoBash because AutoBash cannot automatically test a system state without them. When AutoBash was first developed, we wrote all predicates manually. We found that manually writing predicates is tedious, time consuming, and requires expert knowledge. Even if expert users spend the effort writing standard predicates, these standard predicates might not be reusable by users who heavily customize their software programs.

To reduce the burden of manually writing predicates, we explored the feasibility of automatically generating predicates. Our solution to automatically generate predicates is to observe users fixing a specific configuration problem and to extract sets of commands that can be used as predicates. We developed a heuristic that searches for repeated commands in a user trace that differ in two out of three output features: the exit code, the error messages contained in the screen output, and the set of objects

causally affected by the command. Differences in these output features suggest that the command was executed before and after the system state transitioned from an unhealthy state to a healthy one. This heuristic also uses these output features to determine when a predicate should return true and when it should return false.

To evaluate our heuristic, we conducted a user study to collect traces of commands that real users used to fix several CVS and Apache Web server configuration problems. We ran our heuristic on the traces collected from the user study and showed the predicates generated by the heuristic. The results show that for both CVS and Apache traces, our heuristic finds correct predicates for four to eight out of eleven traces.

1.3 Thesis road map

This thesis is composed of six chapters and an appendix. Chapter 2 describes how we enable mobile computers to run demanding software applications. To increase computing capacity for mobile computers, we built Slingshot, a cyber foraging infrastructure that allows mobile computers to leverage surrogate computers to run resource-intensive applications.

Chapter 3 describes how we make it easier to manage configurations for flexible software systems. We first designed and built AutoBash, which are a set of tools that automate many tedious parts of managing software configurations.

Chapter 4 extends AutoBash by automating the process of predicate generation. We first describe the user study we conducted to collect traces of users fixing configuration problems. Next, we describe our heuristic that can infer predicates from user traces. We then evaluated the quality of predicates generated by the heuristic.

Chapter 5 describes the related work and Chapter 6 discusses my thesis contributions and conclusions. Appendix A provides the documentation that was used in the user study described in Chapter 4, which includes the informed consent and the handout given to each participant.

CHAPTER 2

Improving Performance for Mobile Computers through Replicated Cyber Foraging

This chapter addresses the problem of how to enable mobile computing devices to run demanding applications. We describe Slingshot, an infrastructure that allows mobile computers to leverage surrogate computers to run applications. Slingshot replicates application state on surrogate computers and a user's home machine. This provides the advantage that the replica on the surrogate computer provides good response time and the replica on the home machine provide data safety.

2.1 Introduction

Mobile handheld computing devices, such as smart phones and PDAs, have become more popular over the years. However, creating applications that execute on these small, mobile computers is challenging. On one hand, the size and weight constraints of handheld and similar computers limit their processing power, battery capacity, and memory size. On the other hand, user's appetites are driven by the applications that run on desktops; these often require more resources than a handheld provides. A solution to this challenge is remote execution using wireless networks to access compute servers; this combines the mobility of handhelds and the processing power of desktops.

Although Internet connectivity is increasingly ubiquitous due to widespread de-

ployment of wireless hotspots, the backhaul connections (the wireline access links) between hotspots and the Internet backbone are communication bottlenecks [37]. The uplink bandwidth from a wireless hotspot can be quite limited (e.g., 1.5 Mb/s for a T1 line). Further, this bandwidth must be shared by all hotspot users. The network round-trip time between a hotspot and a remote server may be large due to the use of firewalls and other middleboxes, as well as the vagaries of Internet routing. For interactive applications such as speech recognition and remote desktops, the combination of high latency and low bandwidth is prohibitive; mobile users cannot achieve acceptable response times when communicating with remote servers.

In this chapter, we describe Slingshot, a new architecture for deploying mobile services at wireless hotspots. Slingshot is an instance of *cyber foraging* [11] - utilizing compute resources available in the environment to augment the capabilities of mobile devices. Slingshot replicates applications on *surrogate computers* [10] located at hotspots. A first-class replica of each application executes on a remote server owned by the mobile user. Slingshot instantiates second-class replicas on surrogates at or near the hotspot where the user is located. A proxy running on a handheld broadcasts each application request to all replicas; it returns the first response it receives to the application. Second-class replicas improve interactive response time since they are reachable through low-latency, high-bandwidth connections (e.g., 54 Mb/s for 802.11g). At the same time, the first-class replica is a trusted repository for application state that is not lost in the event of surrogate failure.

Slingshot also simplifies surrogate management. It uses virtual machine encapsulation to eliminate the need to install application-specific code on surrogates. Further, replication prevents the loss of application state when a surrogate crashes or even permanently fails. The performance impact of surrogate failure is mitigated by other replicas, which continue to service client requests.

We have implemented two Slingshot services: a speech recognizer and a remote desktop. Our results show that instantiating a second-class replica on a surrogate lets these applications run 2.6 times faster. Our results also show that replication lets Slingshot move services between surrogates with little user-perceived latency and

recover gracefully from surrogate failure.

2.2 Design principles

We next describe the three design principles we followed in the design of Slingshot.

2.2.1 Location, location, location

Server location can be critical to the performance of remote execution. Consider a handheld connected to the Internet at a wireless hotspot. If the handheld executes code on a remote server, its network communication not only passes through the wireless medium; it also traverses the hotspot's backhaul connection and the wide-area Internet link. In a typical hotspot, the backhaul connection is the bottleneck. For instance, the nominal bandwidth of a 802.11g network (54 Mb/s) is more than an order of magnitude greater than that of a T1 connection (1.5 Mb/s). If the handheld could instead execute code on a server located at the hotspot, it could avoid the communication delay associated with the bottleneck link. For interactive applications that require sub-second response time, server location can make the difference between acceptable and unacceptable performance.

Network latency is also a concern. A server that is nearby in physical distance can often be quite distant in network topology due to the vagaries of Internet routing. Firewalls, virtual private networks (VPNs), and network address translation (NAT) middleboxes add additional latency when connections cross administrative boundaries. For mobile users, a journey of only a few hundred yards can dramatically increase the round-trip time for communication with a remote server. In contrast, a server located at the current hotspot is only a network hop away.

2.2.2 Replicate rather than migrate

The desire to locate services near mobile users implies that services need to move over time. When a handheld user moves to a new location, a surrogate at the new

hotspot will often offer better response time than a surrogate at the previous hotspot.

What is the best method to move functionality? One option is migration: suspend the application on the previous surrogate, transmit its state to the new surrogate, and resume it there. This approach has a substantial drawback: the application is unavailable while it is migrating. Slingshot uses an alternative strategy that instantiates multiple replicas of each service. While a new replica is being instantiated, existing replicas continue to serve the user.

Slingshot replication is a form of primary-backup fault tolerance; i.e., it tolerates the failure of any number of surrogates. For each application, Slingshot creates a first-class replica on a reliable server known to the mobile user—this server is referred to as the *home server*. Slingshot ensures that all application state can be reconstructed from information stored on the client and the home server. This allows all state on a surrogate to be regarded as soft state. Even if all surrogates crash, Slingshot continues to service requests using the first-class replica on the home server. In contrast, a naïve approach that migrates applications between surrogates might lose state when a surrogate fails.

We note that Slingshot handles both *stateful* and *stateless* applications. The result of a remote operation for a stateful application depends upon the operations that have previously executed. Slingshot assumes that applications are mostly deterministic; i.e., that given two replicas in the same initial state, an identical sequence of operations sent to each replica will usually produce identical results. This means that since a replica state on a surrogate will usually not diverge from the first-class replica state on the home server, Slingshot can optimistically return the result from a surrogate replica and verify the result after the main replica returns the result. As we discuss in Section 2.4.2, Slingshot adopts an approach similar to that of Rodrigues' BASE [71] in eliminating non-determinism with wrapper code.

Slingshot instantiates a new replica by checkpointing the first-class replica, shipping its state to a surrogate, and replaying any operations that occurred after the checkpoint. Instantiation of a new replica takes several minutes since the state must travel through the bandwidth-constrained backhaul connection. However, existing

replicas mitigate the perceived performance impact. Until the new replica is instantiated, existing replicas service application requests.

Each replica runs inside a virtual machine so a natural way to implement replay would be adopting ReVirt [33] to manage replicas in Slingshot. We did not adopt this approach as our main concern was the difficulty of capturing all non-determinants and enforcing the same state transitions. This incurs additional latency and would decrease performance improvement. Since we assume replica divergences due to non-determinism are rare, we chose replay on the application level to improve performance by allowing some non-determinism in correct replies. Another reason we chose application level replay is that we would like surrogate computers to be as easy to manage as possible. Requiring an additional replay mechanism on surrogate computers would make management more complicated.

2.2.3 Ease of maintenance

We see the business case for deploying a surrogate as being similar to that of deploying a wireless access point. Desktop computers (without monitors) cost only a few hundred dollars today, not much more than an access point. Further, our results show that surrogates can provide significant value-add to wireless customers in terms of improved interactive performance.

However, surrogates must be easy to manage if they are to be widely deployed. Since we envision surrogates at hotspots in airport lounges, coffee shops, and bookstores, they must require little to no supervision. They should be appliances that require little configuration; most problems should be fixable with a reboot.

To make surrogates easy to manage, Slingshot:

- **minimizes the surrogate computing base.** Each replica runs within its own virtual machine, which encapsulates all-application specific state such as a guest OS, shared libraries, executables, and data files. The surrogate computing base consists of only the host operating system (Linux), the virtual machine monitor (VMware Workstation), and Slingshot. No configuration or

setup is needed to enable a surrogate to run new applications—each VM is self-contained.

- **uses a heavyweight virtual machine**. While para-virtualization and other lightweight approaches to virtualization offer scalability and performance benefits [16, 65, 83], they also restrict the type of applications that can run within a VM. In contrast, by using a heavyweight VMM (VMware Workstation), Slingshot runs the two applications described in Section 2.4 without modifying source code, even though their guest OS (Windows XP) differs substantially from the surrogate host OS (Linux).
- **places no hard state on surrogates.** Because surrogates have only soft state, a reboot does not lead to incorrect application behavior or data loss. If a surrogate crashes or is rebooted, the only impact a user sees is that performance declines to the level that would have been available had the surrogate never been present.

2.3 Implementation

This section discusses our Slingshot implementation. We first give an overview of the implementation, then more details about each software component, and finally the applications that we have modified to run on Slingshot.

2.3.1 Overview

Figure 2.1 shows an overview of Slingshot. For simplicity of exposition, this figure assumes that the mobile client is executing a single application and that a single surrogate is being used. In practice, we expect a Slingshot user to run only one or two applications concurrently, with each service replicated two or three times.

Each Slingshot application is partitioned into a *local component* that runs on the mobile client and a *remote service* that is replicated on the home server and surrogates. Ideally, we partition the application so that resource-intensive functionality

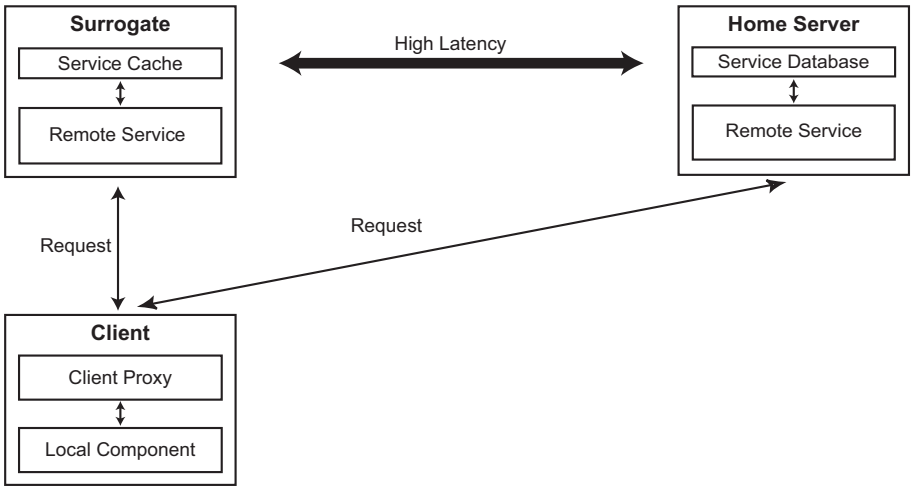


Figure 2.1: Slingshot architecture

executes as part of the remote service; the local component typically contains only the user interface. This partitioning enables demanding applications to run on clients such as handhelds that are highly portable but also resource-impoverished. The applications that we have studied so far (speech recognition and remote desktops) already had client-server partitionings that fit this model. For some applications, the best partitioning may not be immediately clear—in these cases, we could leverage prior work [12, 13, 34, 45] to choose a partition that fits our model.

In Figure 2.1, a first-class replica executes on the home server and a second-class replica executes on the surrogate. The home server, described in Section 2.3.2, is a well-maintained server under the administrative control of the user, e.g., the user's desktop or a shared server maintained by the user's IT department. In contrast, surrogate computers, described in section 2.3.3, are co-located with wireless access points. They are administered by third parties and are not assumed to be reliable.

Slingshot creates the first-class replica when the user starts the application—this replica is required for execution of stateful services. As the application runs, Slingshot dynamically instantiates one or more second-class replicas on nearby surrogates. These replicas improve interactive performance because they are located closer to the user and respond faster than the first-class replica on the distant home server. If

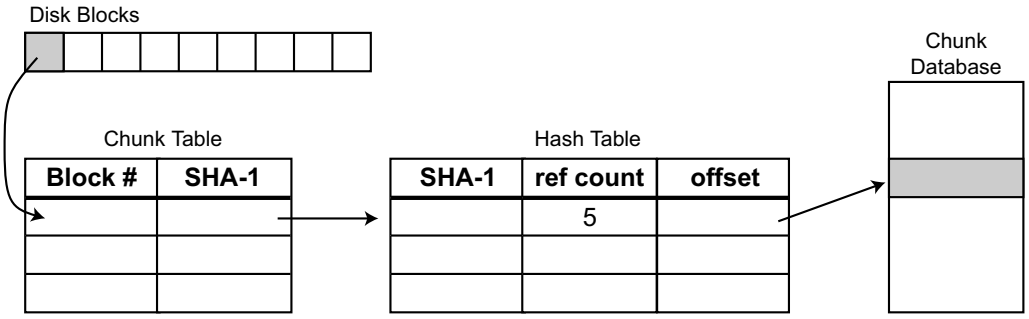


Figure 2.2: Reading a chunk from the service database

no second-class replicas are instantiated, Slingshot's behavior is identical to that of remote execution.

Each replica executes within its own virtual machine. Replica state consists of the *persistent state*, or disk image of the virtual machine, and the *volatile state*, which includes its memory image and registers. To handle persistent state, we use the Fauxide and Vulpes modules developed by Intel Research's Internet Suspend/Resume (ISR) project [51]. These modules intercept VMware disk I/O requests. On the home server, we redirect these requests to a *service database* that stores the disk blocks of every remote service. On a surrogate, VMware reads are first directed to a *service cache*—if the block is not found in the cache, it is fetched from the service database on the home server.

The client proxy is responsible for locating surrogates, instantiating second-class replicas, and managing communication with all replicas. It presents the local component with the illusion that it is using a single remote service by broadcasting each request to all replicas and forwarding the first reply it receives to the local component. Later replies from other replicas are checked for consistency, as described in Section 2.3.4.

If a mobile computer has a high-capacity storage device, such as a flash card or a microdrive, Slingshot reduces the time to instantiate replicas by storing checkpoints on the mobile computer. As described in Section 2.3.6, the client logs all operations that occur after the checkpoint and replays them to bring a new surrogate up-to-date.

2.3.2 The home server

A Slingshot user defines a single, well-known home server that stores and executes the first-class replicas for all of her remote services. Each service is uniquely identified by a *serviceid* string assigned by the user when the service is created. The service database on the home server manages the persistent and volatile state associated with each service. The *director* instantiates and terminates first-class replicas. We describe these components in the next two sections. The home server also runs the VMware virtual machine monitor. Each Slingshot service runs within a separate VM that is dynamically instantiated when a user starts its associated application on the client.

2.3.2.1 The service database

The home server stores the state of every service under its purview in its service database. Previous research in virtual machine migration by Sapuntzakis [72] and Tolia [77] has shown that content-addressable storage is highly effective in reducing the storage costs of virtual machine disk images. We adopt their approach by dividing the disk image of each virtual machine into 4 KB chunks and indexing each chunk by its SHA-1 hash value. As shown in Figure 2.2, each service has a *chunk table* that maps the chunks in its virtual disk image to the SHA-1 hash of the data stored at each location.

The service database assumes that any two blocks that hash to the same value are identical. It maintains a hash table of the SHA-1 values of all chunks that it currently stores. When it receives a request to store a new chunk whose SHA-1 value matches that of a chunk it already has stored, it increments a reference count on the existing chunk. This method of eliminating duplicate storage has been shown to substantially reduce disk usage [30, 61, 67] due to similarities between the disk state of different computers. We expect such similarities to be common in our environment, since a single user may create many remote services from a generic OS image. For example, we created the speech recognizer and remote desktop services discussed in Section 2.4 from the same Windows XP image.

As shown in Figure 2.2, when a first-class replica reads a block from its virtual disk, the Fauxide/Vulpes ISR modules intercept the request and pass the associated logical block number to the service database. The database looks up the block number in the service's chunk table to determine the SHA-1 value of the chunk stored at that location. It then looks up the SHA-1 value in the hash table to find the location of the chunk in the database.

Requests that modify blocks follow a similar path. The database locates the chunk in the service's chunk table. It then indexes on the old SHA-1 value and decrements the reference count associated with the chunk in its hash table. If the reference count drops to zero, it deletes the chunk. The service database next looks up the new SHA-1 value of the modified block in its hash table. If the modified chunk is a duplicate of an existing chunk, the service database increments the reference count of the existing chunk. Otherwise, it stores the chunk and adds its SHA-1 value to its hash table.

Since the volatile state is likely unique to each service, content-addressable storage offers little benefit. Thus, the service database stores the volatile state of each remote service in a file named by its serviceid.

2.3.2.2 The home server director

When a mobile user starts a Slingshot application, the client proxy asks the director on the home server to instantiate the first-class replica. The director uses the serviceid provided by the client proxy to retrieve the volatile state from the service database. It starts a VMware process, resumes the virtual machine with the volatile state, and replies to the client proxy. The persistent state is retrieved on demand from the service database as the first-class replica executes.

When the user terminates the application, the client proxy tells the director to halt the replica. The director suspends the virtual machine, which causes VMware to write its volatile state to disk. It then terminates the virtual machine.

The volatile state is large (e.g., 128 MB) because it contains the entire memory image of the virtual machine. We use Waldspurger's *ballooning* technique [79] to reduce its size. When we create a new service, we place a script in the guest OS that

allocates a large amount of highly-compressible (almost entirely zeroed out) memory pages. When VMware suspends the virtual machine, this script runs to force unused memory pages to disk and replace them with more compressible pages. The director compresses the volatile state with `gzip` before storing it in the service database—this reduces storage and network costs.

2.3.3 Surrogates

The surrogate architecture is similar to that of the home server, except that we replace the service database with the service cache described in Section 2.3.3.2. The director also plays a slightly different role on a surrogate.

2.3.3.1 The surrogate director

The surrogate director currently accepts connections from any client that wishes to instantiate a second-class replica. Potentially, the director could enforce access-control policies similar to those enforced by access points today. The client proxy passes to the director the IP address of its home server and the serviceid of the remote service it wishes to instantiate. The director contacts the home server and requests the volatile state and chunk table for the requested service.

Usually, the home server is already executing the first-class replica of the service in question. For a stateful service, this means that Slingshot must generate a coherent checkpoint that represents the current execution state of that replica. The home server creates this checkpoint by suspending and resuming the virtual machine containing the replica; this causes a new volatile state and chunk table to be written to the service database.

The home servers ships copies of the volatile state and chunk table to the surrogate. Even after compression, this information is quite large (e.g., 32 MB for a remote desktop service)—thus, it can take several minutes to transfer. Next, the director starts a new virtual machine and resumes it using the volatile state. As the replica executes, its disk I/O is intercepted by the ISR modules and redirected to the service

cache described below.

When the client disconnects from the surrogate, the director terminates the virtual machine. Since surrogate replicas are second-class, the service state is logically discarded at this point. However, persistent state chunks remain in the service cache until they are evicted due to storage limitations. This improves response time if the service is later restarted on the surrogate.

2.3.3.2 The service cache

The service cache is a content-addressable store of data chunks. As with the service database, each chunk is indexed by its SHA-1 hash value, and storage of duplicate chunks is eliminated. This lets users benefit from similarities among the disk images of their replicas. For instance, two people using Windows-based services may have similar disk images. Chunks cached by one user can be used by the other.

When the service cache receives a request to read a chunk, it first tries to service it locally. If the chunk is not cached, it asks the service database associated with the replica for the chunk.

A subtle problem occurs because Slingshot assumes determinism at the application level. There is no guarantee that two different replicas will write the same data to the same location on disk. A naïve implementation might ask the service database for an uncached chunk, only to find that it had been over-written by a store performed by the first-class replica. We therefore need to ensure that the service database keeps all chunks that might potentially be requested by second-class replicas.

Slingshot uses a copy-on-write approach for both stateful and stateless services. When a surrogate starts a second-class replica, the service database copies the service's chunk table—the new copy increments the reference count for each of its entries. When the second-class replica is terminated, the service database deletes its chunk table and decrements the reference count for each entry. Thus, even if the first-class replica modifies or deletes a chunk, that chunk is not deleted until after the second-class replica terminates.

A similar concern arises for chunks modified by the second-class replica. The

modified chunks may not be written to the service database by the first-class replica due to non-determinism at the disk I/O level. The surrogate cache therefore pins modified chunks for the duration of a replica's execution—this ensures that they will never need to be fetched from the service database.

The surrogate cache uses an LRU eviction algorithm that exempts chunks that are currently pinned. Since chunks remain cached even after a service is terminated, it is likely that the chunks of a frequent visitor to a hotspot will remain cached between visits.

2.3.4 The client proxy

The client proxy is a stand-alone process responsible for surrogate discovery, instantiating and destroying replicas, and coordinating communication with each replica. It uses Universal Plug and Play (UPnP) [57] to discover new surrogates in its surrounding network environment. Currently, the decision to instantiate a new second-class replica is a made by the user. to add heuristics for monitoring network

On startup, the local component sends the client proxy its serviceid. The proxy immediately instantiates a first-class replica on the home server. It subsequently instantiates second-class replicas on nearby surrogates when requested by the user.

The client proxy maintains an *event log* of requests sent by the local application component. The client proxy spawns a thread for each replica; the thread sends logged events to the replica in the order they were received. Events may optionally have application-specific preconditions that must be satisfied before they can be sent to a replica. For instance, our Virtual Network Computing (VNC) application specifies a precondition that ensures that the remote desktop is ready to accept each key stroke and mouse click event before that event is sent. Services that must process events sequentially to ensure determinism specify that the previous event must complete before an event is sent.

The client proxy records the replies received from each replica in the event log. When the first reply is received, it is returned to the local component. Later replies

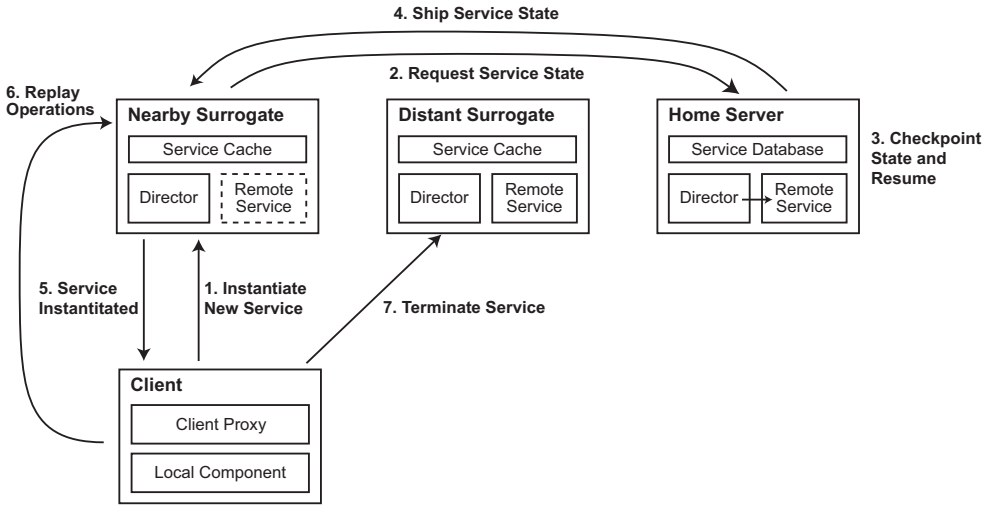


Figure 2.3: Instantiating a new replica

are compared with the first reply to ensure that the replicas have equivalent outputs. If the reply from a second-class replica differs significantly (as determined by an application-specific function) from the reply from the first-class replica, the second-class replica is terminated. Such divergence could be due to a bug in the application-level equivalence function, or it could be the result of a faulty or malicious surrogate. Note that the client proxy may already have received a reply that is later determined to be faulty. In this case, the application is notified via an upcall so that corrective action can be taken. This strategy is similar to those employed in the SUNDR file system [52] and in Brown's operator undo [20]. We could also adopt a strategy similar to Nightingale *et al.*'s external synchrony [63] by buffering the output from the user and allowing the local component to proceed. After receiving the reply from the home server and verifying the buffered output, Slingshot releases the output to the user.

2.3.5 Instantiating new replicas

In Figure 2.3, we show how Slingshot responds to a user moving between hotspots, assuming that a surrogate is located at each hotspot. The client proxy first asks the nearby surrogate to instantiate a replica. The surrogate requests the service state from the home server; the home server checkpoints the first-class replica and ships

the compressed chunk table and volatile state to the surrogate. Note that the distant surrogate can process events for the client during checkpointing. This hides almost all delay associated with suspending and resuming the VM on the home server. The client proxy queues events for the first-class replica while it is being checkpointed and sends them after the replica resumes execution.

The new surrogate uses the checkpoint to resume the service within a new virtual machine. The client proxy brings the new replica up-to-date by replaying all events in the event log that were sent by the application after the checkpoint was created. Once the new replica is up-to-date, it improves interactive response time for the application by responding swiftly to new events sent by the local component. At this point, the client proxy terminates the replica at the previous hotspot.

The benefit of replication is that the user sees little foreground performance impact due to the use of a new surrogate. After checkpointing, the first-class replica on the home server services requests while the new second-class replica is instantiated and brought up-to-date. In contrast, a naïve migration approach would leave the service unavailable while state is being shipped—as we show in Section 2.5, application state can take several minutes to ship over limited backhaul connections. Although the first-class replica is unavailable while it is being checkpointed, that operation is relatively short (approximately ten seconds). Even that cost can be masked if another second-class replica exists.

Slingshot performs two optimizations if a service is marked as stateless. It skips checkpointing the service on the home server (since its state is static). It also does not replay events (since the replica is up-to-date).

2.3.6 Leveraging mobile storage

Migration can be time consuming due to the need to ship state from the home server (step 4 in Figure 2.3). For a typical service, the size of the compressed volatile state and chunk table is 30–40 MB. If the home server is connected to the Internet via a DSL link with 256 Kb/s uplink bandwidth, it takes over 20 minutes to ship the

state.

Given sufficient storage capacity, Slingshot reduces the time to ship state by storing checkpoints on the mobile computer. We observed that the service-specific event log can be used to roll forward replica state from *any* prior checkpoint, not just one that is created at the beginning of replica instantiation. Thus, by storing a checkpoint on a mobile computer and logging all events that occur after that checkpoint, Slingshot can instantiate a replica without shipping state from the home server. Instead, it ships the state from the mobile computer over the high-bandwidth wireless network at the hotspot. Most of this bandwidth should be unused since the capacity of the wireless network is typically much greater than that of the backhaul connection, yet most communication from computers located at the hotspot is with endpoints located outside the hotspot (and thus limited by the backhaul bandwidth).

When a user returns to her home server, she can tell Slingshot to create new checkpoints of her applications on a high-capacity storage device such as a microdrive. Each checkpoint contains the volatile and persistent state. The volatile state, chunk table and hash table are stored in separate files; the chunks that comprise the persistent state are stored in a content-addressable cache on the mobile computer. The event log is empty when the user creates a new snapshot. As the application is used on the road, Slingshot appends each request to the log. This enables Slingshot to instantiate a new replica of a stateful service by first restoring the checkpoint represented by the volatile state, and then deterministically replaying the event log. For stateless services, Slingshot neither records nor replays an event log.

When a new replica is instantiated on a nearby surrogate, the mobile computer tries to find a checkpoint on its local storage device. If a checkpoint is found, the mobile computer ships the volatile state, chunk table, and hash table for its local chunk cache to the surrogate. One reason that we transmit the hash table to the surrogate is that the surrogate can usually maintain this information in memory, whereas a resource-constrained mobile computer cannot. When a disk I/O request misses in the service cache, the surrogate fetches the chunk from the mobile computer if it is available there; otherwise, the chunk is fetched from the home server.

As operations accumulate, so does the time to bring a new second-class replica up-to-date. This means that there exists a break-even point where it takes less time to create a new checkpoint from the first-class replica on the home server and download it over the Internet than it takes to instantiate a replica from client storage.

2.4 Slingshot applications

We have adapted the IBM ViaVoice speech recognizer [84] and the VNC remote desktop [4] to use Slingshot. Due to Slingshot's use of virtual machine encapsulation, we did not need to modify the source code of either application. All Slingshot-specific functionality is performed within proxies that intercept and redirect network traffic.

2.4.1 Speech recognizer

We chose speech recognition as our first service because of its natural application to handheld computers. We used IBM ViaVoice in our work. We created a server-side proxy that accepts audio input from a remote client and passes it to ViaVoice through that application's API. ViaVoice returns a text string which the proxy sends to the client. ViaVoice and our server run on a Windows XP guest OS executing within a VMware virtual machine. The local component of this application displays the speech recognition output.

We chose to implement speech recognition as a stateless service. One can certainly make a reasonable argument that speech recognition should be a stateful service in order to allow a user to train the recognizer. However, we wanted to explore the optimizations that Slingshot could provide for stateless services.

2.4.2 Remote desktop

VNC allows users to view and interact with another computer from a mobile device. In the case of Slingshot, the remote desktop is a Windows XP guest OS executing within a VMware virtual machine. This allows users to remotely execute

any Windows application from their handhelds. This is clearly a stateful service; i.e., a user who edits a Word document expects the document to exist when the service is next instantiated.

Adapting VNC to Slingshot presented interesting challenges. First, the VNC server sends display updates to the client in a non-deterministic fashion. When pixels on the screen change, it reports the new values to the client in a series of updates. Two identical replicas may communicate the same change with a different sequence of updates. The resulting screen image at the end of the updates is identical but the intermediary states may not be equivalent. A second challenge is that some applications are inherently non-deterministic. One annoying example is the Windows system clock; two surrogates can send different updates because their clocks differ.

We noted that some non-determinism is unlikely to be relevant to the user (e.g., a slightly different clock value). Unfortunately, other non-determinism affects correct execution. For example, a key stroke or mouse click is often dependent upon the window state. If a user opens a text editor and enters some text, the key strokes must be sent to each replica only after the editor has opened on that replica. If this is not done, the key strokes will be sent to another application. To solve this problem, we associate a precondition with each input event. When the user executes the event, we log the state of the window on the client to which that event was delivered. When replaying the event on a server, we require that the window be in an identical state before the event is delivered. Since each event is associated with a screen coordinate, we check state equality by comparing the surrounding pixel values of the original execution and the background execution. In the above example, this strategy causes Slingshot to wait until the editor is displayed before it delivers the text entry events.

A second issue with VNC is that its non-determinism prevents us from mixing updates from different replicas. We designate the best-performing replica as the foreground replica and the remainder as background replicas. Only events from the foreground replica are delivered to the client. If performance changes, we quiesce the replicas before choosing a new foreground replica. Two replicas are quiesced by ensuring that the same events have been delivered to each, and by requesting a

full-screen update from the new foreground replica to eliminate transition artifacts. New events are logged while quiescing replicas. Note that the foreground replica is rarely the first-class replica since nearby surrogates provide better performance in the common case.

We were encouraged that VNC can fit within the Slingshot model, since its behavior is relatively non-deterministic. Based on this result, we suspect that application-specific wrappers can be used to cover the non-determinism for many applications and make divergences among replicas a rare event. For those applications where this approach proves infeasible, we could use a VMM that enforces determinism at the ISA level as is done in Hypervisor [19] and ReVirt [33].

2.5 Evaluation

Our evaluation answers the following questions:

- How much do surrogates improve interactive response time?
- What is the perceived performance impact of instantiating a new replica?
- How much does the use of mobile storage reduce replica instantiation time?

2.5.1 Methodology

The client platform in our evaluation is an iPAQ 3970 handheld running the Linux 2.4.19-rmk6 kernel. The handheld has an XScale-PXA250 400 MHz processor, 64 MB of DRAM, and 48 MB of flash. It uses a 11 Mb/s Cisco 350 802.11b PCMCIA card for network communication and a 4 GB Hitachi microdrive for bulk storage. Unless otherwise noted, the home server and surrogates are Dell Precision 350 desktops with a 3.06 GHz Pentium 4 processor running RedHat Enterprise Linux version 3.

We use a Cisco 350 802.11b wireless access point. We emulate the topology in Figure 2.4 by connecting all computers and the access point to a Dell desktop running the NISTNet [22] network emulator. This topology emulates a scenario where the handheld client is located at a wireless hotspot equipped with a surrogate com-

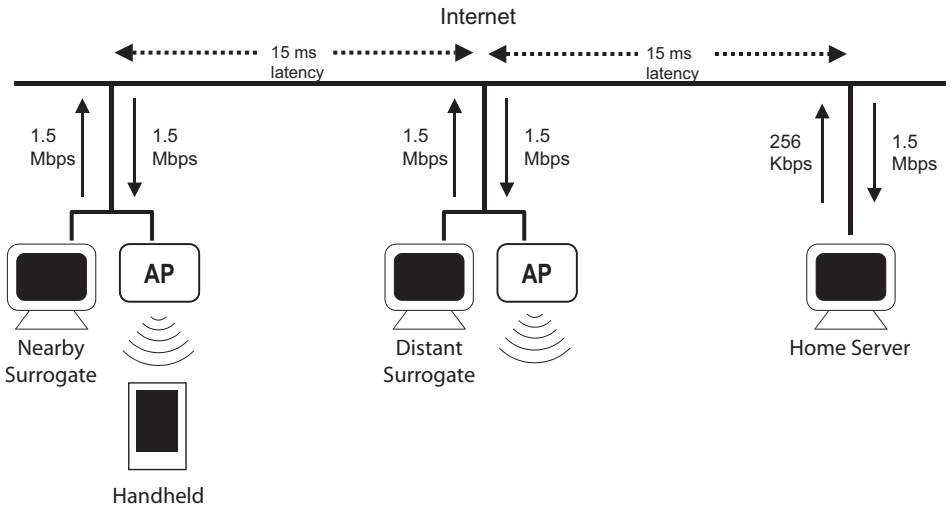


Figure 2.4: Network topology used in evaluation

puter. Hotspots are connected to the Internet through T1 connections with 1.5 Mb/s uplink and downlink bandwidth. A distant surrogate at another hotspot is accessible with latency of 15 ms. The home server is connected through an emulated DSL connection—the latency between the handheld's hotspot and the home server is 30 ms.

We execute the IBM ViaVoice speech recognizer as a stateless service and VNC as a stateful service. For repeatability, the local component of each application executes a fixed, periodic workload. For speech, each iteration of the workload recognizes a phrase and pauses three seconds before beginning the next iteration. For VNC, each iteration opens Microsoft Word, inserts text at the beginning of a document, saves the document, closes Word, and pauses ten seconds before the next iteration begins. The client uses the same heuristics described in Section 2.4.2 to wait until Word opens before inserting text, and to wait until the window is fully closed before beginning the ten-second pause between iterations.

Each service runs within a separate VM configured with 128 MB of memory and 4 GB of local storage. We create each service from a vanilla Windows XP installation. We install the ballooning script described in Section 2.3.2.2 and the application comprising the remote service. We start the application so that it is ready to receive incoming connections, then suspend the VM. We repeat each experiment three times

Service	Remote Execution	Slingshot				
		Steady-State	Creating 1st Replica		Creating 2nd Replica	
			Warm Cache	Cold Cache	Warm Cache	Cold Cache
Speech w/o MD	0.67 (0.67–0.67)	0.24 (0.24–0.24)	0.69 (0.68–0.70)	0.69 (0.67–0.73)	0.52 (0.51–0.52)	0.52 (0.51–0.52)
Speech with MD	0.67 (0.67–0.67)	0.24 (0.24–0.24)	0.80 (0.79–0.81)	0.80 (0.79–0.81)	0.65 (0.65–0.65)	0.65 (0.63–0.68)
VNC w/o MD	18.9 (18.9–19.0)	7.4 (7.2–7.5)	22.8 (21.5–23.6)	22.2 (19.9–23.6)	9.8 (9.7–9.9)	10.0 (9.9–10.0)
VNC with MD	18.9 (18.9–19.0)	7.4 (7.2–7.5)	24.1 (23.9–24.3)	23.1 (21.6–24.4)	13.8 (13.0–14.4)	14.6 (14.4–14.8)

This table summarizes the average response time (in seconds) for all experiments. MD stands for microdrive. Each entry shows the mean of three trials, with the low and high trials given in parentheses. The second column shows response time for remote execution on the home server. The third column shows steady-state performance for Slingshot with a replica on the nearby surrogate. The remaining columns show response time while instantiating a replica on the nearby surrogate with and without a replica running on the distant surrogate.

Table 2.1: Summary of response time results

Service	Creating 1st Replica		Creating 2nd Replica	
	Warm Cache	Cold Cache	Warm Cache	Cold Cache
Speech w/o microdrive	28:06 (27:50–28:27)	27:55 (27:50–28:04)	28:10 (28:05–28:11)	27:57 (27:57–27:58)
Speech with microdrive	3:35 (3:32–3:40)	3:27 (3:26–3:28)	3:39 (3:34–3:45)	3:33 (3:32–3:34)
VNC w/o microdrive	27:48 (27:07–28:28)	27:58 (27:12–28:45)	31:16 (30:57–31:31)	31:08 (31:00–31:25)
VNC with microdrive	6:37 (6:20–7:13)	7:29 (6:59–8:27)	8:59 (8:01–10:00)	8:20 (6:47–9:00)

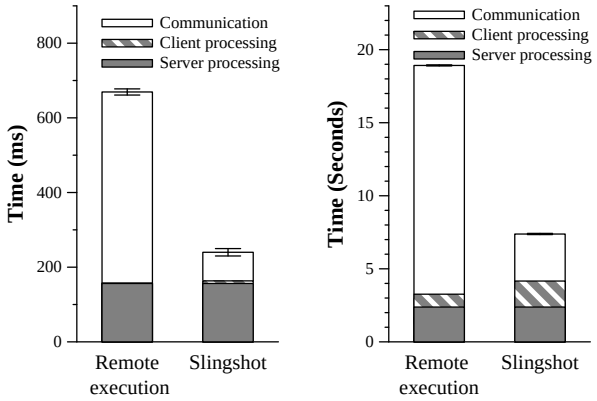
This table summarizes the time (in minutes) to create a new replica on the nearby surrogate for all experiments. Each entry shows the mean of three trials, with the low and high trials given in parentheses. The second and third columns show the time to instantiate a replica when no replica runs on the distant surrogate. The last two column show results with a replica on the distant surrogate.

Table 2.2: Summary of replication time results

and report mean results over all iterations during the three trials. Tables 2.1 and 2.2 summarize all results described in this section.

2.5.2 Benefit of Slingshot

We first measured the benefit of using Slingshot for our two applications. The left bar in each data set in Figure 2.5 shows the average time to perform an iteration of



(a) Speech (b) VNC

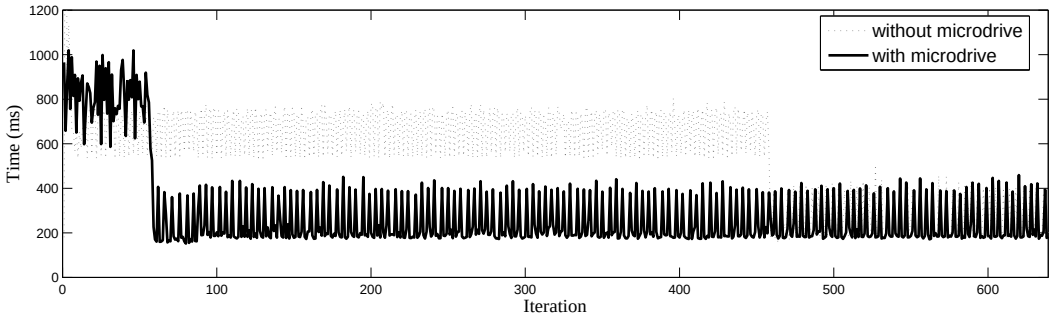
This graph compares the average time to execute the speech and VNC workloads when using remote execution and when using Slingshot. Each bar shows mean response time—the error bars are 90% confidence intervals.

Figure 2.5: Benefit of using Slingshot

the workload when the service is remotely executed on the home server. The right bar shows the average time using Slingshot when a second-class replica executes on the nearby surrogate. We let each application run for several iterations before measuring performance; this eliminates startup transients and shows steady-state performance.

Both the stateless speech service and the stateful VNC service execute 2.6 times faster with Slingshot than with remote execution. The shadings within each bar show the time consumed by server processing, client processing, and communication. For speech, Slingshot increases client processing time since it manages multiple network connections and aggregates responses. For VNC, it also logs requests and responses to local storage. Slingshot's performance benefit comes from reducing the time the application blocks on network communication.

Remote execution performance is affected by both high latency and limited bandwidth. For speech, a back-of-the-envelope calculation shows that 229 ms are required to transfer the 44 KB utterance through the bottleneck 1.5 Mb/s T1 link at the hotspot. Further, since communication is intermittent, TCP slow start causes several 60 ms round-trip delays during transmission. Thus, the remote execution results include 511 ms of network communication time. In contrast, Slingshot uses only 77 ms



This graph shows how response time changes during the instantiation of a speech replica on the nearby surrogate. All chunks are in the service cache prior to each experiment.

Figure 2.6: Speech replication with warm cache

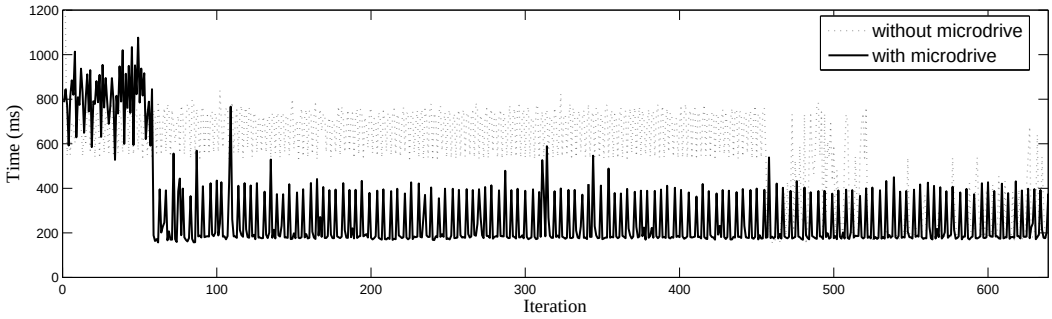
for network communication.

Latency impacts VNC performance more than bandwidth. Because the client waits for remote actions such as button clicks and key presses to complete before initiating the next action, there are many round-trip delays during the VNC workload. In addition, client polling in VNC leads to more round-trip delays than are strictly necessary. For this workload, remote execution on the home server requires 15.6 seconds for network communication, while Slingshot requires only 3.2 seconds.

2.5.3 Stateless service replication

We next examined the impact of instantiating stateless second-class replicas. In this experiment, a user with a first-class replica running on the home server arrives at the hotspot on the left in Figure 2.4 and decides to instantiate a replica on the surrogate there. For repeatability, we do not measure the latency of UPnP service discovery. We examine behavior when the service cache is cold (no chunks are initially cached) and warm (all chunks are initially cached). The warm cache scenario is most likely if the user has recently visited the hotspot; the cold cache scenario is the worst cache state possible.

We first ran three trials without a microdrive attached to the iPAQ. Since the handheld has limited storage, the service state must be loaded from the home server



This graph shows how response time changes during the instantiation of a speech replica on the nearby surrogate. No chunks are in the service cache prior to each experiment.

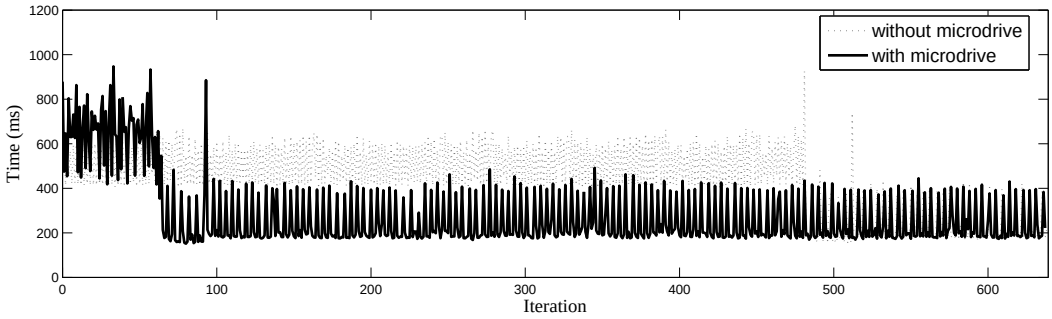
Figure 2.7: Speech replication with cold cache

as described in Section 2.3.5. We then ran three trials with the microdrive; in this case, the state of the speech service is loaded from the iPAQ as described in Section 2.3.6. Figures 2.6 and 2.7 show results for representative trials with a warm and cold cache, respectively.

In Figure 2.6, the sharp drop in response time for both lines is a result of the completion of replica instantiation. Before the replica is instantiated, speech requests must be serviced by the distant home server; after instantiation, the new second-class replica provides quicker response time. Without the microdrive, it takes 28:06 minutes to ship the service state from the home server. However, replica instantiation exhibits only a minimal impact on application performance—average response time during replication is only 2% greater than response time with remote execution on the home server.

When the replica is instantiated from state stored on the client's microdrive, the new second-class replica is instantiated in only 3:35 minutes (7.8 times faster). However, the performance impact of replica instantiation is more substantial: average response time increases by 20% compared to remote execution. Shipping a large amount of data over the wireless network causes queuing delays at the access point and on the handheld that adversely affect application performance.

The cold cache scenario in Figure 2.7 exhibits a less clear difference in perfor-



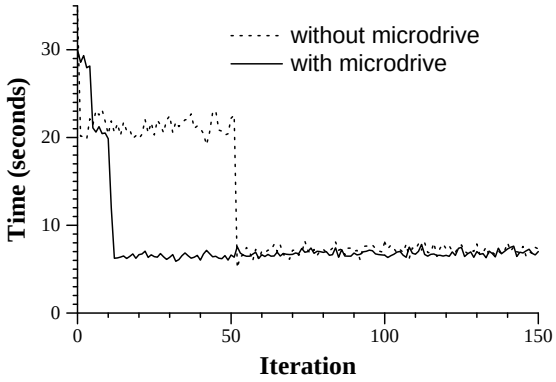
This graph shows how response time changes during the instantiation of a speech replica on the nearby surrogate while another replica executes on the distant surrogate. All chunks are in the the service caches prior to each experiment.

Figure 2.8: Speech: Moving to a new hotspot

mance before and after replication completes. After the new replica is instantiated, it fetches chunks of its persistent state on demand from the home server or iPAQ; this occasionally delays its responses. Note that the first-class replica on the home server mitigates the performance impact—if the second-class replica is substantially delayed by fetching state, the first-class replica responds faster.

2.5.4 Instantiate another stateless service

We next examined a scenario in which the user of the speech service moves from one wireless hotspot to another. This experiment begins with the user located at the middle hotspot in Figure 2.4. A second-class replica is running on the surrogate at that hotspot and a first-class replica is running on the home server. At the beginning of the experiment, the user moves to the left hotspot and decides to instantiate another replica on the surrogate at that hotspot. While this new replica is being created, both the second-class replica on the distant surrogate and the first-class replica on the home server service application requests. As soon as the new replica is instantiated, Slingshot terminates the replica on the distant surrogate. Since we did not have three identical servers with which to run this experiment, the home server is a slightly less-powerful Dell Optiplex 370 with 2.8 GHz Pentium 4 processor running RedHat 9.



This graph shows how response time changes during the instantiation of a VNC replica on the nearby surrogate. All chunks are in the service cache prior to each experiment.

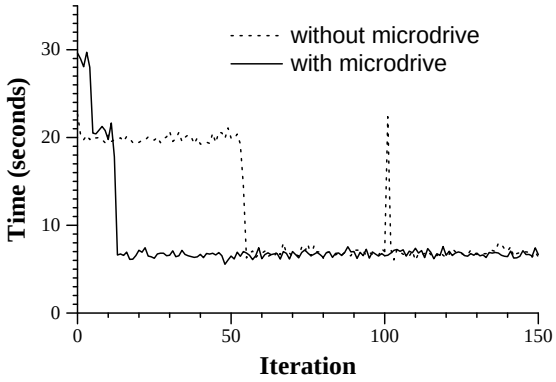
Figure 2.9: VNC replication with warm cache

Figure 2.8 shows how the average time to perform an iteration of the speech recognition workload varies during this experiment—we show only warm cache results here. Compared to the previous experiment, the time to instantiate a new replica is relatively unchanged. However, response time during replication is improved because the existing second-class replica responds faster to requests than the replica on the home server. Without the microdrive, application response time is reduced by 23% compared to remote execution; with the microdrive, application response time is reduced by 2%. These results show that a surrogate can still provide significant benefit even when not located at the user's current hotspot.

2.5.5 Stateful service replication

We next repeated the experiment in Section 2.5.3 for the stateful VNC service. Prior to the experiment, we perform 30 iterations of the VNC workload. We then begin the experiment by instantiating a replica on the nearby surrogate. Figures 2.9 and 2.10 show results from the warm and cold cache scenarios, respectively.

Without the microdrive and with a warm service cache, Slingshot takes 4 seconds to checkpoint the VNC service—this is reflected in the higher response time for the first iteration. Slingshot takes 22:42 minutes to ship the checkpoint to the surrogate



This graph shows how response time changes during the instantiation of a VNC replica on the nearby surrogate. No chunks are in the surrogate cache prior to each experiment.

Figure 2.10: VNC replication with cold cache

and 5:02 minutes to replay the logged operations. During replication, average response time is 20% higher than when using remote execution on the home server. This increase is due to the background traffic associated with shipping state from the home server interfering with the latency-sensitive foreground traffic of VNC.

In contrast to the prior results for the speech service, the VNC results show little difference between the warm and cold cache scenarios. In particular, VNC performance markedly improves in the cold cache scenario as soon as the client starts using the second-class replica. Most of the chunks needed by the service are read from the service database during the replay of logged operations.

When the handheld stores a VNC service checkpoint on its microdrive, Slingshot takes 3:19 minutes to ship the state from the client, and 3:18 minutes to replay the log. These two phases are clearly visible in the "with microdrive" line in Figure 2.9, where response time degrades by 52% compared to remote execution while state is being shipped, and by 9% while the log is replayed. Note that the log replay with the microdrive includes the 30 iterations that occurred prior to the experiment. Since the microdrive checkpoint is taken when the user leaves home, all logged operations after that point must be replayed. However, since shipping state takes less time with the microdrive, the user generates fewer logged operations during migration. Overall,

Slingshot instantiates the replica over 4 times faster when a checkpoint exists on the microdrive.

On the other hand, the nature of the VNC service makes it more difficult to switch between surrogates; the current foreground replica must be quiesced before Slingshot designates another replica as the foreground one. Thus, in contrast to the speech service, Slingshot can not take the fastest response from any surrogate for any individual VNC request; instead it must take a more heavyweight action to switch surrogates if the response time of the foreground surrogate consistently lags that of a background surrogate.

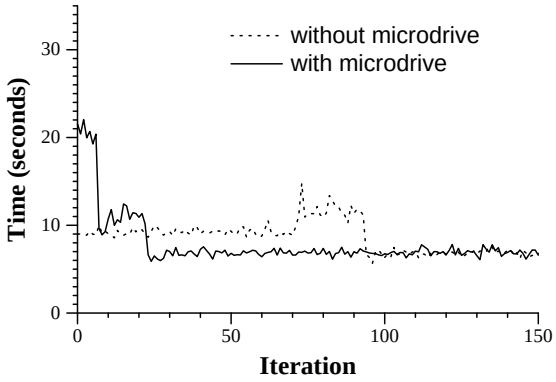
2.5.6 Instantiate another stateful service

We also repeated the experiment in Section 2.5.4 for VNC. Prior to the experiment, we create a second-class replica of the VNC service on the distant surrogate and a first-class replica on the home server. We then execute 30 iterations of the VNC workload. The experiment begins when we start to instantiate another second-class replica on the nearby surrogate.

As shown in Figure 2.11, the presence of another second-class replica on the distant surrogate substantially improves performance during replication. Compared to remote execution, Slingshot provides VNC response times during replication almost twice as fast without the microdrive, and 70% faster when state is fetched from client storage.

2.6 Discussion

Our design principles in Section 2.2 lead us to our final design in Slingshot. In this section, we first examine the design space for cyber foraging infrastructures with different design principles, such as fast instantiation and privacy. We next discuss some privacy concerns in Slingshot and potential solutions to address these concerns.



This graph shows how response time changes during the instantiation of a VNC replica on the nearby surrogate while another replica executes on the distant surrogate. All chunks are in the the service caches prior to each experiment.

Figure 2.11: VNC: Moving to a new hotspot

2.6.1 Design alternatives

To make surrogates easy to maintain, we chose virtual machines to replicate applications in Slingshot. Using virtual machines has the advantages described in Section 2.2 but also has some disadvantages. First, even though we only need to transfer the memory image, it takes around twenty minutes to instantiate a new virtual machine. Recent work in fast virtual machine migration [28, 70] can be used to improve service instantiation time but it still works best in local area network. Second, a virtual machine created from a user's home computer may include a user's private data on it, and such virtual machines could be a threat to user's privacy.

Process migration [14, 25, 32] and process domain migration [65] are two abstractions that could be used to replicate or migrate remote services in Slingshot. Migrating at both process and process domain levels has the advantage that only relevant state needs to be migrated. This can substantially reduce the amount of data to be migrate and improves migration time. Since less data is migrated, it is easier to verify that sensitive personal data is not transferred to a surrogate machine. However, both migration schemes require all surrogate computers to support the same migration infrastructure. Hence, users can only run remote services constructed to run for that specific migration infrastructure.

2.6.2 Privacy concerns

We design surrogate computers to be easy to maintain so that surrogate computers can be like appliances that do not need system administrations. However, user might worry that running applications on these public compute servers might disclose their private data. We did not address privacy issues in this thesis but examine two aspects that we can leverage existing security techniques to improve a user's privacy on Slingshot.

Users interact with surrogate computers through sending requests to remote services and getting responses back from them. One aspect to improve privacy is to reduce private data transferred to surrogate computers by partitioning an application in a secure fashion. Chong *et al.* [26] demonstrated how Swift can automatically partition web applications into a client and server program. Swift places confidential information on the server and computation that requires responsiveness on the client. Slingshot could deploy a similar technique to partition applications so that sensitive data is placed on the client component and code that requires computation resources is on the remote service.

We envision surrogate computers to be shared among users at the same wireless hotspot, therefore, techniques that allow computer systems to isolate each user's application are desirable [38, 5]. These techniques all assume that tamper-resistance hardware platforms exist [76].

2.7 Summary

In this chapter, we demonstrated how to improve performance for portable mobile computers through replicated cyber foraging. Handhelds can improve interactive response time by leveraging surrogate computers located at wireless hotspots. Slingshot's use of replication offers several improvements over a strategy that simply migrates remote services between computers. Replication provides good response time for mobile users who move between wireless hotspots; while a new replica is being instantiated, other replicas continue to service user requests. Replication also

lets Slingshot recover gracefully from surrogate failure, even when running stateful services.

Slingshot minimizes the cost of operating surrogates. For surrogate computers to be of maximum benefit, they must be located at wireless hotspots, rather than in machine rooms that are under the supervision of trained operators. Slingshot uses off-the-shelf virtual machine software to eliminate the need to install custom operating systems, libraries, or applications to service mobile users. All application-specific state associated with each service is encapsulated within its virtual machine. Further, Slingshot's replication strategy means that surrogates need not provide 24/7 availability. If a surrogate fails or is rebooted, no state is lost.

CHAPTER 3

Improving Software Configurations through Automated Configuration Management

Software applications today provide a wide range of options that allow users to customize them. However, software applications are often difficult to configure correctly due to their dependencies on shared libraries, configuration data, and environment variables. This chapter addresses this problem by introducing AutoBash, which consists of a set of tools that can automate many configuration management tasks.

3.1 Introduction

Users spend too much time configuring their computers [43]. Currently, configuration management is often an isolated process, in which each user discovers on her own the particular incantations that are required to transform her particular computer system from a misconfigured state to a correct one. Configuration management can be especially frustrating because, in the process of trying to reach a more desirable system state, the user may take some action that leaves the system in a state worse than the one that existed before.

While autonomic systems that can configure themselves are an admirable goal [48], experience to date suggests that it is very hard to completely hide the complexity of modern computer systems. Thus, a less ambitious goal may be in order. Rather than try to eliminate configuration entirely, we adopt the more pragmatic approach

of providing users and system administrators with a set of interactive tools, which we call AutoBash, that helps them perform configuration management tasks.

For instance, consider the actions of a typical user faced with a configuration problem. She might search the Web and query colleagues to find others who have encountered and solved a similar problem. She might then try the most likely solution and test her system to see if the problem has been fixed. If the system still does not operate correctly, she will carefully try to undo any changes that she has made and then try the next possible solution. AutoBash can help this user by automating many of the most tedious and frustrating steps in the above process. It uses causal dependency tracking and analysis implemented within the operating system kernel to speed the search for solutions to configuration problems, and it uses speculative execution to transparently apply and test possible solutions while retaining the ability to undo incorrect actions.

AutoBash has three modes: observation, replay, and health monitoring. In all three modes, AutoBash expresses system health as the result of executing a set of predicates. In observation mode, AutoBash records the actions of users as they adjust their systems to fix a particular problem. AutoBash logs input and output as the user probes the system, makes state changes, and tests the new state. In replay mode, AutoBash tries to fix the same problem on different systems by applying actions that fixed the problem previously. AutoBash lets users learn from each others' experiences: as AutoBash sees more correct solutions to a problem, it has a greater database of potential solutions to draw upon. In health monitoring mode, AutoBash periodically tests the correctness of a computer system in the background to diagnose configuration bugs before they become critical.

AutoBash leverages causality support within the Linux kernel to understand the outputs (causal effects) and inputs (causal dependencies) of executing configuration actions. This leads to the following benefits:

- Causal tracking reduces the amount of testing that must be performed to determine whether a particular action fixed a problem or caused a new problem. AutoBash expresses system health as the result of executing a set of (deter-

ministic) predicates. When it performs a configuration action, it tracks the set of entities modified as a result of the action. If none of the entities serves as an input to a predicate, then the evaluation of that predicate will not change as a result of the configuration action. Thus, the predicate need not be retested.

- Causal tracking provides a compact representation of interactive user activity. When a user interacts with an editor or GUI application, AutoBash could potentially record his actions by logging inputs (keystrokes and mouse events) or outputs (system calls). Unfortunately, either approach often produces a large amount of data that is difficult to understand or replay deterministically. In contrast, AutoBash uses causal tracking to represent interactive activity as a minimal set of changes made to persistent state during the activity.
- Causal tracking helps explain to users how a problem was fixed. When a user fixes the problem interactively, AutoBash shows him which actions possibly contributed to the solution and which did not contribute. Further, it shows the changes to intermediate entities (e.g., diffs of configuration files) that possibly led to the solution. When AutoBash autonomously searches for a solution to a configuration problem, it shows a similar report to the user before committing any solution it finds. This allows the user to understand and confirm any change to configuration state suggested by AutoBash.
- AutoBash transparently recovers from incorrect actions by tracking all causal effects of each action. We use Nightingale *et al.*'s Speculator [62] to roll back all effects of incorrect actions. While tools exist for checkpoint and rollback of persistent registry or file system state, such systems are incomplete unless they also roll back transient state such as process memory. For instance, a corrupted application can cause a problem to remain until the system is rebooted. Even worse, a corrupted application may subsequently write incorrect data to the file system, making the effects of an incorrect action durable.
- Causal tracking, when combined with speculative execution, provides isolation for configuration activities. If a configuration action is found to be incorrect

and is rolled back, AutoBash guarantees that any other activity that observed the incorrect state will also be rolled back. This strategy provides optimistic concurrency for speculative configuration actions — an approach that lets AutoBash perform potentially time-consuming configuration actions in the background while the user continues to use the computer system for normal activities.

Our current AutoBash prototype assists in solving configuration bugs that are confined to a single computer system, such as a home computer, personal workstation, or stand-alone server. Our results show that AutoBash dramatically decreases the amount of user interaction required to fix configuration problems in the Concurrent Versions System (CVS), the GNU Compiler Collection (GCC) cross compiler, and the Apache HTTP server by automating many of the most tedious activities. Our results also show that operating system causality analysis decreases the amount of time needed by AutoBash to automatically find solutions by an average of 35% and the time spent on predicate testing by an average of 70%.

3.2 Design

This section first discusses the design overview of AutoBash and then describes a configuration problem we encountered that inspired AutoBash. Although we manually solved that configuration problem, it allowed us to envision how AutoBash could be very useful for both novice users and system administrators when they troubleshoot configuration problems.

3.2.1 Overview

The goal of AutoBash is to help its users find solutions to the configuration problems they are facing. AutoBash divides configuration activities into *actions* that modify system state and *predicates* that test system correctness. We define a *solution* to be a sequence of actions that transforms a system from an incorrect state, in which

one or more predicates evaluate to false, to a correct state, in which all predicates evaluate to true.

Predicates are guaranteed to have no side effects; that is, they are not allowed to visibly modify system state such as files and processes. AutoBash ensures the absence of side effects by speculatively executing each predicate and rolling it back on completion. Predicate execution, return codes, and causal inputs are visible to AutoBash but not to any other process running on the system. (More precisely, any entity that observes predicate execution is rolled back to its state prior to the observation once the predicate completes, and external effects are not visible to the user or other computers.) There are several ways we can obtain predicates. Predicates are similar to test cases and may be provided as part of the software (similar to Windows configuration wizards or test suites for program testing). Alternatively, they may be created by a community of users. Users can also add predicates that demonstrate symptoms that they would like to fix. Since a predicate is simply a Unix process, the normal actions that users take to verify whether a solution is working (e.g., loading a Web server test page) can be transformed into predicates with minimal effort. A predicate can be written as a shell script or can be a binary executable with a return value indicating true or false. Chapter 4 explores generating predicates by inferring them from traces of users fixing actual configuration problems.

AutoBash maintains a predicate database that contains the tests associated with all applications under its purview. Table 3.1 shows a sample predicate database. AutoBash organizes predicates with the following schema: name, application, predicate file location, last execution result, the time the predicate was last executed, input record file location, and average predicate execution time. The *application* field refers to the application a predicate is designed to test. *Input record file location* is the location of the file containing the inputs a predicate is dependent upon. AutoBash stores the *average execution time* so that it can warn a user if predicate execution takes much longer than usual. Based on our evaluation of three applications in Section 3.5, five to ten predicates per application seem to cover most of the configuration bugs we encountered.

Name	App.	File location	Last exe. result	Last exe. time	Input record file location	Ave. execution time (seconds)
pred.cvs.0	CVS	/AB/	Succeed	08-20-2007 19:21:05	/AB/input.pred.cvs.0	5.30
pred.cvs.1	CVS	/AB/	Succeed	08-20-2007 19:21:10	/AB/input.pred.cvs.1	5.04
pred.cvs.2	CVS	/AB/	Succeed	08-20-2007 19:21:15	/AB/input.pred.cvs.2	4.68
pred.gcc.0	Cross compiler	/AB/	Succeed	08-20-2007 19:21:19	/AB/input.pred.gcc.0	2.10
pred.gcc.1	Cross compiler	/AB/	Succeed	08-20-2007 19:21:22	/AB/input.pred.gcc.1	1.58
pred.apache.0	Apache	/AB/	Succeed	08-20-2007 19:21:22	/AB/input.pred.apache.0	9.25
pred.apache.1	Apache	/AB/	Succeed	08-20-2007 19:21:32	/AB/input.pred.apache.1	10.08
pred.apache.2	Apache	/AB/	Succeed	08-20-2007 19:21:41	/AB/input.pred.apache.2	9.76
pred.apache.3	Apache	/AB/	Succeed	08-20-2007 19:21:51	/AB/input.pred.apache.3	8.30

Table 3.1: Sample predicate database

AutoBash has three modes of operation: observation, replay, and health monitoring. In its observation mode, AutoBash helps a user manually resolve a configuration problem. The AutoBash shell is a version of the standard Linux `bash` shell that we have modified to support speculative execution and causality tracking. Each command given to the shell is considered an action that may contribute to an eventual solution. AutoBash executes each action speculatively, which allows its user to roll back an action's effects if the action later proves to be incorrect. AutoBash automatically tests for system correctness by executing predicates after each action completes. It also tracks causality to explain how a problem was fixed and what actions are related once a solution is found. The solutions found during observation are canonicalized and saved so that they can be used later to fix similar configuration bugs.

In its replay mode, AutoBash automatically searches for solutions to a configuration problem. Its search space is the set of solutions that were previously found for similar problems. Similar to predicates, potential solutions can come from many

sources: the user's prior experience, solutions developed by vendors or a community of users. Chapter 4 describes how we can automatically infer solutions by observing users trying to fix configuration problems. While AutoBash does not currently provide a distributed system for locating potential solutions, other projects, such as the Friends Troubleshooting Network [44], have shared configuration information through peer-to-peer networks. These projects have also addressed security and privacy concerns related to sharing configuration information. AutoBash stores solutions and their metadata in a local repository similar to how it stores predicates. For each solution, AutoBash keeps track of the solution location and the application which the solution is associated with.

AutoBash's replay mode speculatively executes a solution followed by predicate testing. If any predicate evaluates to false, AutoBash decides that the solution does not fix the configuration bug and rolls back that solution. AutoBash then tries the next solution in its solution database until all predicates evaluate to true. Speculative execution isolates AutoBash's replay activity from other non-configuration tasks — a user may continue to use her computer while AutoBash searches for the solution in the background without worrying that results will be corrupted by observing state that is in the process of being reconfigured. After AutoBash finds a solution, it explains the actions it took, their causal effects, and how they relate to the configuration predicates. Its user can examine this information before committing the configuration changes; if the user disagrees with AutoBash's solution, AutoBash will roll back the candidate solution and continue searching for another fix.

The predicate database is also consulted whenever an application-specific solution is found during observation or replay mode. When a solution is found but before it is committed, AutoBash verifies that all predicates that previously succeeded still do; this checks for "solutions" that fix one problem but cause another. AutoBash tracks causality to record and store the system state observed by each predicate. Thus, if a solution does not affect the state observed by a predicate, AutoBash recognizes the predicate need not be run. This limits the number of predicates that must be tested for each solution.

In its health monitoring mode, AutoBash periodically tests all predicates in its predicate database. If any predicate fails the check, AutoBash can be set to enter replay mode to find and suggest a potential solution to the user.

3.2.2 Usage scenario

This section describes a sample misconfiguration we encountered, as well as how we envision AutoBash to be useful for fixing such configuration problems. Our research team set up a CVS repository to store our source code and publications. We created a CVS group that owned the repository and added all team members to the CVS group. When we used the CVS repository, we found that other team members would get a "permission denied" error when they attempted to check out any project they had not checked in.

We manually fixed this problem by first analyzing the error message and examining the current system state. We then tried various actions to modify the system settings. After each action, we tried to check out the project to test if the modification solved the problem. After a few trials, we realized that the file permission error was due to an incorrect file ownership – the default group for a user is his original user group and hence any files added to the repository would be owned by the user and inaccessible to other users despite the repository being accessible to all. We finally applied the correct action: `chmod` the SGID bit of the CVS repository directory. This action causes CVS to set the group id of a newly added file or directory in the CVS repository directory to the same as the parent directory instead of the current process.

There are several disadvantages to this approach. First, any system state modification might adversely affect CVS or other applications running on the machine. If the modification is incorrect, we have to manually undo our changes — this could be difficult as we often do not always know the effects of changing system state. Further, if a system change solves the problem, we do not know if the solution breaks other features of the application, or even other applications running on the system. Finally, after we have solved a configuration problem, we often do not know what actions we

took contributed to the solution.

As an alternative, we can use AutoBash to diagnose and solve this problem. If CVS is packaged with a set of standard solutions and predicates, we can run AutoBash in replay mode to search for a solution. AutoBash first runs all predicates from the standard predicate sets to determine which predicates exemplify the buggy scenario. In the scenario above, any predicate that involves checking in a project as a user and checking out as a different user is sufficient. AutoBash then goes through each solution from the standard solution set and executes it speculatively. After executing a potential solution, AutoBash first tests predicates that demonstrate the problem. If any such predicate fails, AutoBash rolls back the solution and tries the next one. If a solution causes all those predicates to succeed, AutoBash next tests the remaining predicates for CVS and other applications. If a solution exists, AutoBash can identify it and ensure that the solution does not break other existing configuration of CVS and other applications. AutoBash's replay mode is a useful tool for users who do not participate much in system administration work.

If CVS does not have standard predicate and solution sets, the user can still run AutoBash in observation mode to manually fix the configuration problem. First, the user needs to specify one or more predicates that expose faulty application behavior. The user can also write predicates to test basic functionality for CVS and other applications. In our sample misconfiguration, a predicate that checks in a new file as one user and checks out the same file as a different user can expose the faulty behavior. Next, the user tries different actions in the AutoBash shell; AutoBash executes each action speculatively and tests predicates after each action to determine if the action and prior actions fix the problem. If all user-specified predicates succeed, AutoBash tests against the remaining predicates to check that the fix does not break the existing configuration for CVS and other applications. While running AutoBash in observation mode, the user can roll back any prior action. If subsequent actions observe the effects of a prior action, AutoBash informs the user of this dependency and rolls back those actions too. Through this mechanism, AutoBash in observation mode can be a useful tool for system administrators to try out different actions safely.

After the administrator fixes a problem, AutoBash also analyzes causal information tracked by the kernel to explain which actions contribute to solving the problem and how they relate to each other.

3.3 Implementation

This section describes the implementation of AutoBash. We first present a background review on Speculator, upon which we built AutoBash, and then we discuss how we implemented AutoBash.

3.3.1 Speculator background

Speculator is a system within the kernel that supports speculative execution through causal dependency tracking and lightweight checkpoint and rollback. A process invokes Speculator to checkpoint its state and continues execution. Later, if the speculation is found to be incorrect, Speculator rolls back the process execution and the process is restarted using the checkpoint state. If the speculation is found to be correct, Speculator discards the checkpoint, marks the process as non-speculative, and continues the process execution. Speculator supports speculative execution throughout the operating system, tracking causal dependencies as a speculative process interacts with the kernel, file system, and other processes. Thus, a speculative process can safely change system state; all modifications can later be undone should the speculation prove wrong.

Speculator adds two new data structures to the kernel. A *speculation* object tracks the set of kernel objects, such as processes and files, that depend on a speculative operation. Each speculative kernel object has an *undo log* that tracks the state needed to undo speculative modifications to that object. As speculative processes interact with kernel objects by executing system calls, Speculator uses these data structures to track causal dependencies. For example, if a speculative process writes to a file, Speculator creates an entry in the file's undo log that refers to the speculations on which the writing process depends. If another process reads from the speculative

file, Speculator creates an undo log entry for the reading process that points to all speculations on which the file depends.

Speculator ensures that speculative state is never *externalized*, i.e. made visible to a user (terminal output) or any external device (network, disk, etc.). If a speculative process attempts to externalize state, Speculator buffers its output in the kernel until the outcome of the speculation it depends on is decided. If a speculative process performs a system call that Speculator cannot handle by either propagating causal dependencies or buffering output, Speculator blocks the process until it becomes non-speculative.

3.3.2 Tracking causality

In designing AutoBash, we considered several possible methods of tracking causality. Our goal was to capture sufficient information to allow us to reason about the causal effects of configuration actions and how they relate to predicates that test system health, while also minimizing performance impact on foreground applications and keeping the amount of causal state manageable. We want to track causal effects of actions and causal dependencies of predicates so that we only re-run those predicates affected by an action, reducing the time to find a solution.

We define causality in the operating system kernel in the following way. Processes, files, directory entries, sockets, pipes, and signals are all considered first-class entities. When a process interacts with other entities by executing system calls, we use Speculator to track the causal relationships among the entities. Specifically, if a process observes another entity (e.g., it reads a file or receives a signal), we say that the process becomes causally *dependent on* the observed entity. If a process modifies another entity, the modified entity becomes causally dependent on the modifying process. If a process observes an entity while modifying it (for instance, when a process writes to a file, it checks access permissions of a file before writing to it and therefore becomes dependent on the file), the process and the modified entity become mutually dependent.

We track causality for predicates and solutions at the point we start executing them because we believe that most of the relevant interactions usually happen within the execution interval. We only track causal effects and dependencies for entities that are affected by the initial predicate or solution process. The process being tracked marks the start and end of each tracking interval by making `ioctls` on a pseudo-device. The first `ioctl` causes the OS to begin tracking causality. Subsequently, the causal information (the set of recorded inputs or outputs) can be queried by issuing another `ioctl`. Tracking is terminated by issuing a final `ioctl` that releases the data structures used by the OS to maintain causal information. This interface allows user-level AutoBash tools to ask specific questions about the causal events of processes that they are tracking, while minimally affecting the performance of other processes that are not being tracked. In this manner, we can obtain the most relevant causal interactions, and the overhead of causality tracking is only paid during configuration management and maintenance.

3.3.2.1 Tracking output sets

We use Speculator to track causal effects. When a user-level process asks that its causal outputs be tracked, Speculator assigns a unique id to the request and creates an *output set* that contains pointers to entities that depend on the subsequent execution of the calling process. Initially, the only member of the output set is the process that made the `ioctl`. As the process interacts with other entities by executing system calls, we add to the output set new entities that come to depend on an entity already in the output set (using the definition of "depends on" from Section 3.3.2). For instance, when a process that is a member of the output set modifies a file, that file is added to the output set. If another process reads the file, the reading process is also added to the output set.

The implementation of output sets is minimal; each entity has a list of pointers to the output sets on which it currently depends. Each output set created by Speculator has a list of reverse pointers to the entities it contains. Thus, adding a new entity to an output set requires only that Speculator add a pointer to two lists. Speculator can

track multiple output sets simultaneously to track dependencies for different ranges within the same process or ranges that occur in different processes.

3.3.2.2 Tracking input sets

A user-level process may also ask that its causal inputs be tracked. We have modified Speculator to create an *input set* that contains all processes, files (both content and metadata) and directory entries that the tracked process comes to depend on during the execution of subsequent system calls. For instance, if a process receives a signal from another process, the sending process is added to its input set. Similarly, if it reads a file, the file and the directory entries used to look up the file within the file system are added to its input set. We handle inter-process communication (IPC) entities (e.g. signals and sockets) like we handle other entities (e.g. processes and files), but IPC entities are not reported in the final input sets because they are not persistent entities within a computer system.

As defined so far, an input set contains only the causal inputs observed by a single process. However, user-level tasks often do not map directly to individual processes; for instance, a shell may fork a child to perform a specific task, which in turn will fork other children or interact with other processes via IPC to perform subtasks. In such situations, the input set should capture all the causal inputs of the *collection* of processes that are cooperating to perform the task.

We calculate the input set for cooperating processes as follows. First, any member of the output set of the tracked process is considered to potentially be cooperating to perform a high-level user task — we create an individual input set for each such entity. When an entity in the output set comes to depend on another entity in the output set, its input set becomes the union of the input sets of the two entities. If an entity that is not a member of the output set comes to depend on an entity that is a member, it joins the output set and its input set becomes equal to that of the entity on which it depends.

When a query is issued, Speculator returns the individual input set of the calling process. The set returned consists of not only the entities directly observed by the

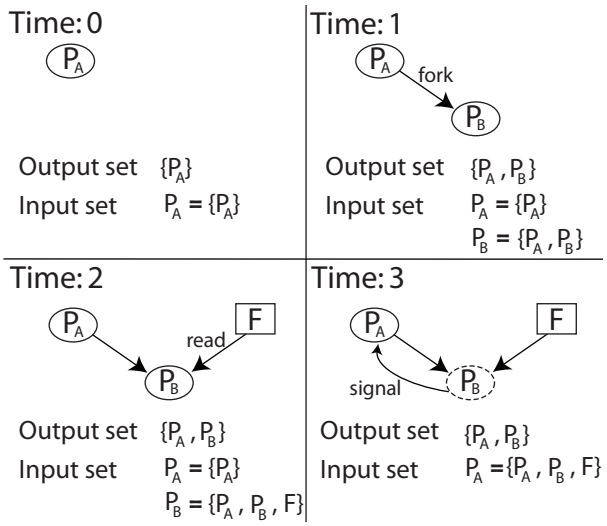


Figure 3.1: An input sets tracking example

querying process but also some entities observed by other processes. For instance, if the tracked process forks a child process, the entities observed by the child become part of the parent's input set when the child exits and the parent receives its termination signal. On the other hand, if the child does not interact further with its parent (e.g., it might be a command executed in the background), the entities it observed are not part of its parent's input set; this reflects the intuition that the parent cannot depend on the child if it does not observe its execution in any way. Similarly, if a tracked process makes a remote procedure call to a server, only the entities observed by the server after it receives the request but before it replies to the tracked process will be returned as part of the input set. A pipe, socket, or localhost packet transfers input sets between the client and server but is not part of the input sets.

Figure 3.1 shows an example of how we calculate input sets for cooperating processes. Processes are shown in ellipses and files are shown in rectangles.

- At time 0, process P_A requests that its input and output sets be tracked. Both the output set and P_A's input set contain only the process itself.
- At time 1, P_A forks a child process P_B. Since P_B is dependent on P_A, P_B is added to the output set and its input set is the union of P_A's input set and itself.

- At time 2, P_B reads file F and becomes dependent on F so F is added to P_B's input set. Since F is not affected by any object in the output set, it is not added to the output set.
- At time 3, P_B exits and a signal is sent to its parent P_A; therefore, P_A is now dependent on P_B. Since P_B is already in the output set, P_A's input set becomes the union of P_A's and P_B's input sets.

Of course, this definition of the input set is a heuristic — a precise characterization of the input set would require a semantic knowledge of application behavior that seems quite difficult to achieve in an operating system kernel. Nevertheless, our results show that this heuristic performs extremely well in practice for the configuration management tasks in which we have employed it.

3.3.3 Observing user actions

In its observation mode, AutoBash assists its user in manually fixing a problem. The user begins configuration management by launching the AutoBash shell. Next, the user specifies which predicates are related to the problem he is attempting to fix. These can be drawn from AutoBash's predicate database by specifying the application that is being configured. Alternatively, the user may use the AutoBash shell to specify new predicates on the fly; this can be useful when the user is attempting to fix a rare problem that he has never previously encountered. AutoBash verifies that at least one of the specified predicates evaluates to false; if all evaluate to true, the user must specify at least one more predicate that exemplifies the problem he is trying to solve.

The user then enters configuration actions using the AutoBash shell. The shell records the command line input for each action. For simple actions, the command line information is all that is needed to capture the action; for instance, the `adduser` and `chown` utilities can be precisely characterized by the inputs entered on the command line. However, for actions that start interactive applications, such as a text editor, the command line information is woefully inadequate to characterize what the user is doing. For such interactive applications, AutoBash extracts state deltas as described

in Section 3.3.4.

After each action completes, AutoBash automatically tests to see if the prior actions have corrected the configuration problem. It first runs the predicates initially specified by the user. AutoBash executes each predicate speculatively by forking a child process and invoking Speculator to make the child speculative. After the child exits, the AutoBash shell reads its return code and input set, then asks Speculator to roll back the child's speculation.

If the return codes of the initially specified predicates indicate success, AutoBash next runs the remaining predicates in its database. This checks for solutions that fix one configuration problem but cause another. The initially specified predicates are run first to reduce testing time: since these predicates exemplify the configuration problem the user is trying to fix, they are most likely to fail and eliminate the need to test remaining predicates.

Since executing predicates is time consuming, AutoBash analyzes causality to limit the number of predicates that are tested. During predicate execution, AutoBash records the predicate's input set. It also records the output set of each action. If an action's output set does not intersect the input set of a predicate, AutoBash does not re-run the predicate because a predicate that depends on the same input set must return the same answer as from the last execution.

Each action is run as a separate speculative execution. AutoBash forks a child process to perform the action, then asks Speculator to execute the child speculatively and track its output set. When the child exits, Speculator provides AutoBash with the output sets of which the child is a member. If the child is a member of a prior action's output set, it has come to depend on that action by observing some entity that depends on the prior action. From this information, AutoBash determines which actions depend on which prior actions.

Using the AutoBash shell, the user can roll back any prior action. While actions can be rolled back individually, the rollback of an earlier action requires that subsequent actions that depend on that action also be rolled back (since they observed incorrect output of that action). AutoBash can use the action interdependency infor-

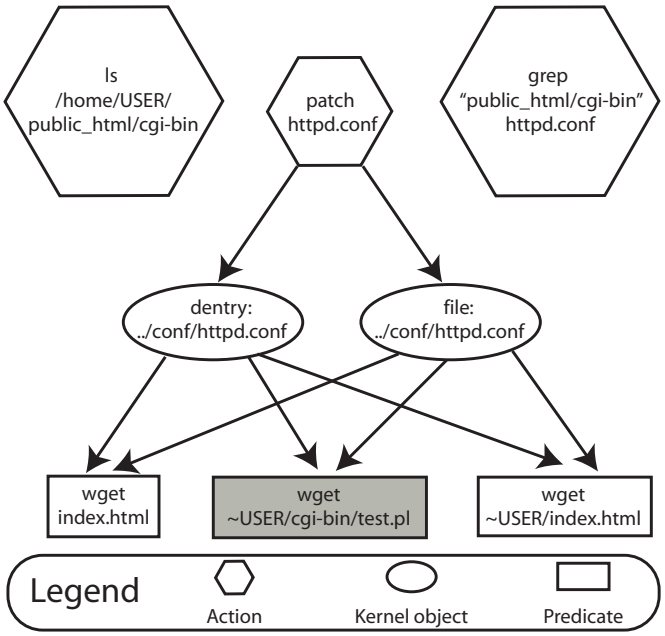


Figure 3.2: Sample causal explanation

mation it collects to inform the user which actions will be rolled back. Alternatively, a user may simply choose to roll back all actions that occurred after a specified action.

We have often encountered users who have fixed a particularly nasty configuration problem but are not quite sure of what actions they performed that contributed to the solution. To help such users, AutoBash can provide a causal explanation once a solution is found. Figure 3.2 shows an example explanation produced by AutoBash after an Apache Web server configuration bug was fixed. The symptom of the misconfiguration was that a user can `wget` her own home page but cannot `wget` the results of a CGI script in her home directory. The problem was caused by Apache's configuration file not allowing users to execute CGI, and the fix was to add the appropriate line to the configuration file. The explanation in Figure 3.2 lists actions (hexagons) that were performed and not rolled back. Each action might affect kernel objects (ellipses) that causally depend (shown by directed arrows) on it. Finally, the explanation lists predicates (rectangles) used to verify that the problem is solved. In Figure 3.2, the three predicates, from left to right, are `wget` the default home page, `wget` the result of the CGI script in the user's directory and `wget` the

user's home page. The grayed predicate evaluates to false before applying any action, while the other predicates evaluate to true. By examining the explanation, we can infer that certain actions (`ls` and `grep`) modify no kernel objects that these predicates depend on; from this, we can conclude that they do not contribute to the solution. Since `patch` is the only action that affects kernel objects that all predicates depend on, we can also conclude that patching httpd.conf is what fixed the bug.

When a solution is found, AutoBash saves it in a form that can later be used by the replay tool. The saved version contains only those actions that were executed and not later rolled back. Like PeerPressure [80], AutoBash canonicalizes solutions to replace specific identifiers such as userids, home directories, hostnames, and IP addresses with generic variables.

By default, Speculator does not allow any output that depends on speculative actions to be externalized. While theoretically correct, this policy does not work well with an interactive debugging tool such as AutoBash. Therefore, we modified Speculator to allow speculative output to the screen when the user runs AutoBash in observation mode. However, predicate testing normally does not externalize any output to the screen. We only use this capability on predicates when we are testing and developing them.

3.3.4 Extracting state deltas

For non-interactive applications, the command line information recorded by AutoBash is sufficient to describe how the action can be replayed later to fix a similar problem. However, for interactive applications such as editors and GUI configuration tools, AutoBash must capture how the tool is used in order to perform similarly during replay.

Potentially, AutoBash could record all inputs to an interactive application; e.g., it could record key strokes, mouse movements, and button presses for a local application, or network packets for a distributed one. Such an approach can lead to AutoBash recording a large amount of information. More problematically, such low-level inputs

are not easily understandable by a user. Further, Section 2.4.2 showed that it can be quite hard to deterministically replay graphical input since events such as button presses must be timed to occur after widgets such as drop-down menus and screen windows appear.

An alternative approach would be to record *outputs* such as all system calls made by the interactive application. During replay, the recorded system calls could be reissued to try to duplicate the observed behavior of the interactive application. Like the technique of recording user inputs, this approach can generate a large amount of data that is difficult to understand. Further work would be required to abstract away non-deterministic OS variables such as process identifiers and file handles.

Instead of recording outputs directly, AutoBash captures a *state delta*, which is the difference in the state of the system caused by the execution of the interactive application. Note that state delta can be generated for the execution of any command but here we only use them for interactive applications. When AutoBash shell receives the signal from an exiting child process executing a configuration action, it queries Speculator to determine if the process has received any input from an interactive source (e.g., from a console, network interface, or similar external device). If no such input has been received, the command line information is assumed to be sufficient to characterize the action. If not, AutoBash generates a state delta.

First, AutoBash queries the output set for the action. This lists all entities that causally depend on the execution of the action. Next, AutoBash computes a diff for each entity. For files, it uses the standard Linux `diff` tool to generate a patch file, and supplements the information with the changes made to the file attributes. For directory entries, it records entries added to and deleted from the directory, as well as the directory attributes. For processes, it records the sequence of causal inputs to the process generated by the user action. For example, it records signals sent to the process and data communicated via IPC mechanisms such as pipes.

AutoBash uses several optimizations to reduce the size of state deltas. First, it eliminates temporary entities that are created and destroyed during the action such as temporary files and processes that are forked and terminate. Next, it eliminates

all events that precede a deletion. For example, not all input sent to a process that terminates is saved — only the final termination event is retained. Finally, events that cancel each other are eliminated; e.g., two `chown` operations that change a file's attributes to a new value and then back to the original value. This process is like log optimization in the Coda file system [58], except that we apply such optimizations to OS kernel entities rather than just file system objects.

After optimization, the state delta gives a terse representation of the causal effects of an action. For instance, using a GUI tool to change the settings of a server might produce a state delta that contains only a diff showing changes made to a configuration file and a signal sent to the server to cause it to re-read the configuration file.

3.3.5 Finding a good solution

AutoBash's replay mode searches through a database of potential solutions to find one that fixes a problem being encountered on a user's computer. Such solutions may come from peers who have encountered similar problems, from software developers who are supporting their product, or from the user previously fixing the same problem. Since solutions can be scripts or binaries, AutoBash handles a wide variety of replay inputs.

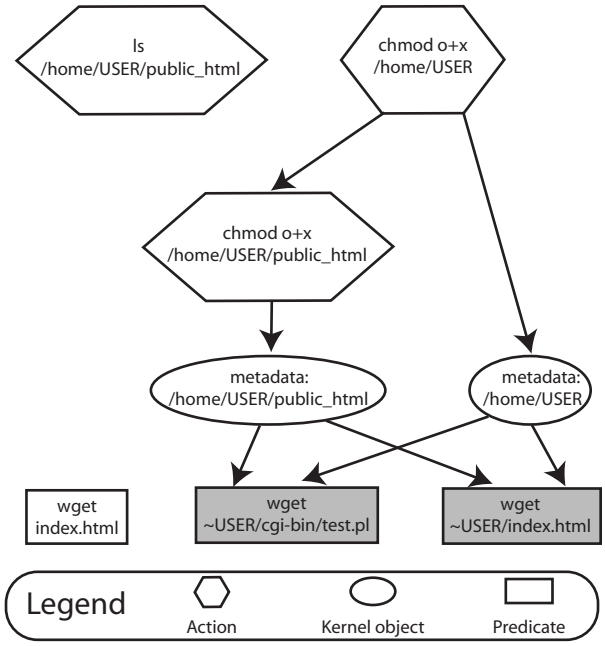
AutoBash's replay mode is designed to execute in the background; the user can still use her computer while AutoBash reconfigures. Speculative execution of predicates and potential solutions provides isolation: if any process observes the causal effects of executing a predicate or configuration action, that process is transparently rolled back to the point before the observation and re-executed. Since Speculator does not externalize speculative state, if a foreground application comes to depend on AutoBash's background task and tries to externalize output, Speculator buffers the foreground task's output until the outcome of the speculation is decided. Thus, the effects of speculative execution are not visible to non-AutoBash processes, the user, other computers, or any entity external to AutoBash and the kernel. Of course, speculative execution consumes resources on the computer and some work performed by

other processes may need to be rolled back, so the performance of non-configuration tasks will be impacted by a background AutoBash execution. The performance impact will depend on how resource-intensive the background task is.

When using AutoBash in replay mode, the user first specifies the predicates that exhibit faulty application behavior. As described in Section 3.3.3, these predicates can be drawn from those used to test the application in the computer's existing predicate database, or they can be new predicates specified by the user. AutoBash first runs all user-specified predicates and records whether each succeeds or fails. Speculator is used to roll back each predicate after execution and to report the input set for each predicate. If no predicates fail, AutoBash asks the user to specify an additional predicate that exemplifies the problem being debugged. We call this process *initial predicate testing*.

AutoBash next iterates through the solutions in the solution database to find one that makes all specified predicates succeed. AutoBash speculatively executes a potential solution by forking a child process and invoking Speculator to make it speculative. It then waits for the child process and queries Speculator for the solution's output set. If the output set of the solution and the input set of a predicate do not intersect, AutoBash does not re-run the predicate. Assuming that the predicate depends on the same input set, its behavior will not change as a result of executing the solution. Thus, if there is no intersection between the output set of the solution and the input set of a failed predicate, AutoBash immediately considers the solution unsuccessful (it did not fix at least one predicate).

After applying a solution, AutoBash first tests the predicates that failed in the initial predicate testing, and then it tests the rest of the predicates in the database. AutoBash only re-runs those predicates whose input sets intersect with the solution's output set. If all such predicates succeed, AutoBash declares the solution successful. If any predicate fails, AutoBash declares the solution a failure, rolls back that solution, and tries the next solution. AutoBash currently applies only one solution at a time in replay mode, though it could potentially execute combinations of solutions. plan to explore the effectiveness of this idea and



This figure shows how AutoBash leverages causality analysis to determine which predicates to run after applying a potential solution to an Apache configuration bug.

Figure 3.3: Sample causality analysis

Once a solution is found, AutoBash outputs the solution to the user as a potential fix. AutoBash provides a causal explanation similar to the one in Figure 3.2 that shows the actions executed and all predicates affected. After viewing this information, the user can confirm the fix, in which case AutoBash commits the speculation and exits, or decline the fix, in which case AutoBash rolls back the speculation and continues searching for a solution.

Figure 3.3 shows how AutoBash determines what predicates to run after applying a potential solution for an Apache Web server configuration bug. The symptom of the misconfiguration is that a user cannot `wget` her own home page nor the result of a CGI script in her home directory. The configuration problem is that the Apache Web server process does not have search permission to access the user's home directory where the user's home page and CGI script are located. The action to fix this configuration problem is to give search permission to others for this user's home directory. The notions used in this figure are the same as in Figure 3.2. The grayed predicates failed in

the initial predicate testing, while the predicate shown in white succeeded. Figure 3.3 illustrates that the action `chmod o+x /home/USER/public_html` is dependent on the action `chmod o+x /home/USER`. Also, both `chmod` actions have causal effects on the kernel objects which the grayed predicates depend on. From this causality analysis, AutoBash infers that it only needs to re-run the last two predicates.

We expect that many applications will have a set of standard predicates that can be used to validate application configuration. As part of initial predicate testing, AutoBash records whether each standard predicate succeeds or fails. The results of the initial predicate testing are aggregated across multiple runs and packaged with each solution.

AutoBash uses its knowledge of which standard predicates succeed and fail to order the solutions it tries. If the user does not specify any predicates to test and hence does not hint at which application might be failing, the first step in AutoBash's replay mode is to run the standard predicate sets for all applications. AutoBash executes all standard predicates, $\{P_0, P_1, ..., P_n\}$, and aggregates their results as a binary result vector $R_{current} = \{1, 0, ..., 1\}$ (1 = succeed, 0 = fail). AutoBash also maintains a set of solutions S_i from $\{S_0, S_1, ..., S_m\}$ from prior replays with their own standard predicate results R_i. Next, AutoBash compares the current result vector $R_{current}$ to the result vectors from prior replays, R_i, and computes their Hamming distance. AutoBash uses this Hamming distance to order the set of solutions it tries, starting with the solution that has an R_i closest to $R_{current}$. The intuition behind this approach is that solutions that fix a particular problem tend to repair the same set of failing predicates; thus, by observing the results of standard predicate execution, AutoBash can guess which solutions are most likely to succeed.

More sophisticated heuristics could potentially be applied. For instance, Attariyan and Flinn [9] record and compare input sets for each standard predicate to generate signatures for misconfigurations. Their intuition is that similar configuration bugs will cause predicates to fail in similar ways. Thus, there is substantial similarity between the set of kernel objects observed by each predicate on computer systems that are exhibiting the same buggy behavior.

3.3.6 Validating system health

Much like a check-up with a family physician, AutoBash's final mode of operation validates that a properly configured system is continuing to behave correctly. AutoBash periodically (as a daily `cron` job) runs all the predicates in a computer's database. If any of these predicates fail, it produces a report that alerts the computer's administrator about the problem.

The AutoBash `cron` job can be configured to run at night, so as to minimally perturb the performance of foreground applications running on the computer. Speculative predicate execution assures that there is no causal impact of periodically running predicates on the rest of the computer system.

The input set and result of each predicate is collected after a predicate is run. This information is used during replay mode to determine which predicates previously failed prior to attempting a solution, and also to determine which predicates need not be re-run while testing a potential fix.

3.4 Limitations

We assume that a predicate is written to evaluate certain desired properties of the system correctly; that is, the predicate should deterministically evaluate to true if those properties hold and false otherwise. If a predicate does not return the correct answer, AutoBash may miss a correct solution or falsely identify an incorrect solution. Also, AutoBash currently does not support debugging non-deterministic errors. Combining AutoBash with a system such as Rx [66] might help tease out such non-determinism. Potentially, one could intentionally vary non-deterministic inputs such as thread scheduling and the time of day to help explain errors to an administrator or user.

AutoBash is intended to debug user-level configuration errors on a single computer. Since AutoBash uses a layer-below approach implemented inside the operating system to track causality and roll back modifications, it cannot track causal effects for or undo kernel-level actions such as the insertion of a kernel module. Potentially, a

multi-level approach to rollback, for instance, by using a virtual machine monitor to checkpoint and roll back kernel state [82], could allow AutoBash to handle such scenarios. AutoBash also does not support configuration of distributed applications that span more than one computer. However, if Speculator could be extended so that the operating systems of multiple computers could share speculative state as in systems such as Time Warp [46], AutoBash could tackle distributed configuration management.

AutoBash is currently only targeting "functional" problems associated with misconfiguration and does not handle performance problems. To correctly diagnose performance problems, we need accurate accounting of shared resources (CPU, memory, etc.) within the system. Since AutoBash also consumes these resources for speculation and roll back, it would interfere with performance debugging.

Finally, AutoBash assumes that all configuration actions happen under its purview. One can imagine, for instance, a delayed configuration action such as a process that reads a configuration file once every 24 hours. AutoBash would not observe this interaction since it limits causality tracking to the time period when the AutoBash tools are being employed to fix a problem. While one could track *all* causal interactions in the system to capture such activities as discussed in Section 3.3.2, it would be quite difficult to separate out configuration activity from other causal interactions that have no effect on the problem being debugged.

3.5 Evaluation

Our evaluation answers the following questions:

- How well can AutoBash identify solutions to configuration problems?
- How effective is causal dependency analysis in reducing predicate testing?
- Does tracking state delta make it easier to understand the effects of interactive applications?

Bug	CVS configuration problem description
1	Repository not properly initialized
2	User not added to CVS group
3	CVS performs unwanted keywords substitution
4	Setgid bit not set on repository, so group for new files is incorrect
5	$TMPDIR environment variable set incorrectly
6	$CVSROOT misconfigured for a CVS user
7	$CVSROOT not set for a different CVS user
8	$CVSROOT variable set but not exported correctly
9	Repository permissions allow global access
10	Repository created using wrong group
Bug	GCC cross compiler problem description
1	Cross compiler tools not in the default path
2	Cross compiler setup overwrites default path instead of appending
3	Dangling libcrypt.so symlink does not point to correct library
4	Archive tool (ar) not in the default location
5	Kernel header module.h contains wrong content
6	Compiler cannot invoke linker due to bad location
7	Cross compiler specs file does not contain xscale architecture definitions
8	Cross compiler not configured to accept -pthread option
9	C compiler configured correctly, but C++ compiler is not
10	Cross compiler specs file does not contain version flag information
Bug	Apache Web server problem description
1	Apache cannot search a user's home directory due to incorrect permissions
2	Apache cannot read CGI scripts due to incorrect permissions
3	Symlink used to point to CGI scripts in a user's home directory, but Apache is not configured to follow symlinks
4	Apache configuration does not allow CGI execution in user home directories
5	Misconfiguration treats CGI scripts as regular Web pages
6	Apache not configured to load PHP module
7	Handler not set for PHP pages
8	Apache not configured to use index.php as default
9	User has insufficient permission to use .htaccess authorization
10	File .htaccess configured incorrectly

Table 3.2: Description of injected configuration bugs

- Can AutoBash identify solutions to configuration problems involving multiple applications?

3.5.1 Methodology

All of our experiments were run on a Dell Precision 370 desktop computer with a 3.00 GHz Pentium 4 processor and 2 GB of memory. The computer runs a Red Hat Enterprise 3 Linux kernel version 2.4.21. All trials using AutoBash run with a modified version of the kernel that includes Speculator — all other trials run with the default 2.4.21 kernel.

To validate AutoBash, we injected configuration bugs into our computer system for three applications: the CVS version control system, the GCC arm-linux cross compiler, and the Apache Web server. We identified 10 common configuration bugs for each application by searching through FAQs, manuals, and trouble reports on the Web. Table 3.2 contains a short description of each bug we used. We also created 5–8 predicates that test the configuration of each application — these predicates are shown in Table 3.3. These predicates are sufficient to cover all the configuration bugs in Table 3.2; that is, each bug causes at least one (and often several) predicates to fail.

For each bug, we created a script that injected the bug, as well as a solution script that fixed the problem. In these experiments, we do not assume that there exists a set of standard predicates for the application as described in Section 3.3.5 and hence AutoBash does not have standard predicate results for potential solutions. Therefore, AutoBash uses the following heuristic to order the solution search space. As AutoBash has no hint about which application is misconfigured, it runs all predicates from Table 3.3 in the initial predicate testing. Based on the results of this initial predicate testing, AutoBash tries solutions from the application with the most failed predicates. When verifying the correctness of a solution, AutoBash runs the failed predicates first and only re-runs those predicates whose input sets intersect with a solution's output set.

Predicate	CVS predicate description
1	a user checks in a project and checks it out again
2	a user checks in a project, and a different user checks it out
3	same as predicate 1, but assumes a default repository is defined
4	unauthorized users cannot access repository
5	checks if CVS performs unwanted keyword substitutions
Predicate	GCC cross compiler predicate description
	Note: For all predicates, we check that the compilation succeeds and the compiled executable is the right file format
1	take a "hello world" .c file, compile it with explicit path names
2	take a "hello world" .c file, compile it using default paths
3	take a kernel module .c file, compile it
4	take a .c file, compile it, link it to a shared cryptography library
5	take several .c files, compile them into object files, archive the object files into a static library, compile a program that links in the static library
6	take a .cc file, compile it with a c++ cross compiler
7	take a .c file, compile it, statically link in a math library, check if the compilation succeeds and the compiled executable is statically linked to the math library
8	take a multi-threaded .c file, compile it for the XScale architecture
Predicate	Apache Web server predicate description
1	wget Apache's default home page
2	wget a user's default home page
3	`wget` the result of a CGI script from Apache's default root directory and `diff` the output with the expected output
4	`wget` the result of a CGI script from a user's home directory and `diff` the output with the expected output
5	wget the result of a PHP test page
6	wget a PHP test page that is set to be the default page

Table 3.3: Description of predicates for each application

We evaluated the correctness of AutoBash by injecting each configuration bug from Table 3.2 into our test computer system and using AutoBash in replay mode to fix the bug. In every case, AutoBash was able to find a solution that corrected the misconfiguration.

3.5.2 Effectiveness of AutoBash

We evaluated the performance impact of speculative execution and the benefit gained by causality analysis as compared to a baseline AutoBash implementation. We created three versions of AutoBash's replay mode:

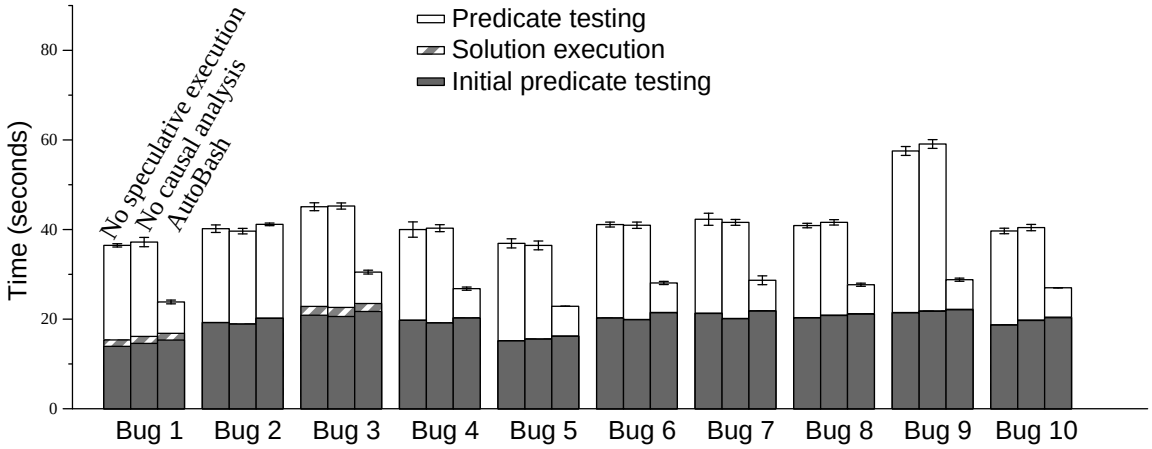
- **No speculative execution**: This version uses neither speculative execution nor causality analysis. We manually created scripts to undo each solution and predicate. Without speculative execution, we found it can sometimes be quite hard to undo the effects of predicate testing because some applications do not provide an interface to undo actions. For example, to undo the effects of a CVS predicate testing, our undo script removes the directory added in predicate testing from the CVS repository and removes lines from CVS's history file.

- **No causal analysis**: This version uses speculative execution but not causality analysis.

- **AutoBash**: This version uses both speculative execution and causality analysis.

The trouble we had in developing manual undo scripts for the "No speculative execution" version demonstrated to us the importance of speculative execution; with AutoBash, the operating system tracks the causal effects of a predicate and undoes them after testing the predicate.

We evaluated these three versions as follows. For each bug in Table 3.2, we compare the total search time to find the solution. Our results for each version are shown by the three bars for each bug in Figures 3.4, 3.6, and 3.8. For each bar, we break the total search time into three parts: initial predicate testing (before any



This figure shows the time to find a solution for the 10 CVS bugs in Table 3.2. The left bar in each data set shows the time to find a solution without speculative execution or causality analysis, the middle bar shows the time with only speculative execution, and the right bar shows the time with both. Each result is the mean of five trials — the error bars show 95% confidence intervals.

Figure 3.4: AutoBash performance for CVS benchmark

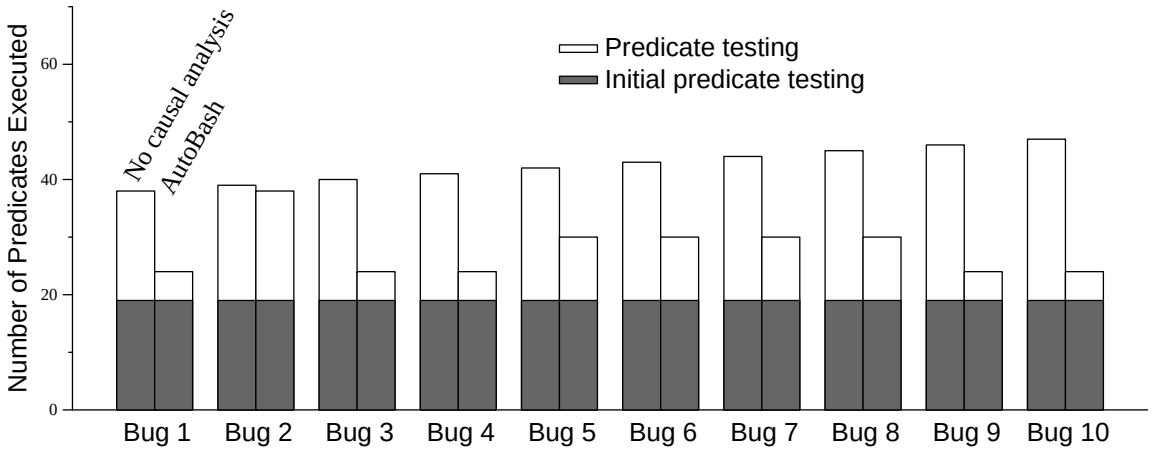
solution is tried), solution execution and undo, and predicate testing (to verify a potential solution).

Comparing the execution time of the first and second bars in each dataset gives the performance overhead of speculative execution. Comparing the execution time of the second and third bars in each dataset shows the amount of time saved by tracking output sets of solutions and input sets of predicates.

We also show the number of predicates run in initial predicate testing and solution searching in Figures 3.5, 3.7, and 3.9. We coalesce the first two versions into "No causal analysis" because they run the same number of predicates.

3.5.2.1 CVS

Figure 3.4 shows the time needed to solve the ten CVS configuration bugs described in Table 3.2. Comparing the first two bars in each dataset shows that speculative execution adds minimal overhead for this benchmark – the maximum overhead of speculation is 2.7% for bug 9. The initial predicate testing time is unavoidable because we assume that AutoBash receives no hints from the user and hence runs through all predicates. We found that solution execution time is negligible. Hence,



This figure shows the number of predicates executed by AutoBash while finding a solution for the 10 CVS bugs in Table 3.2. Five trials were run for each bug; however, the number of predicates executed was identical in each trial.

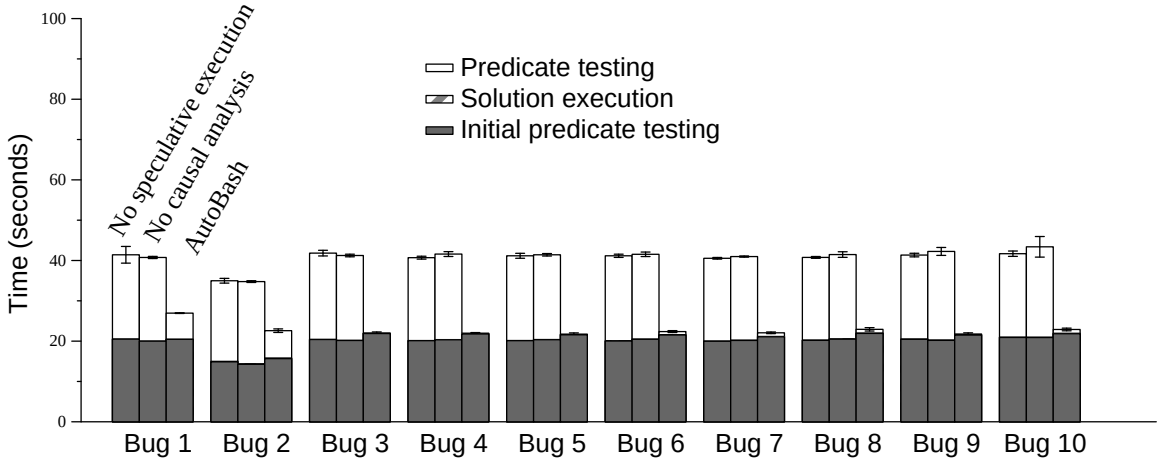
Figure 3.5: Number of predicates executed for CVS benchmark

the only component of our replay mode that can be improved is predicate testing. This observation led us to focus on using causality analysis to reduce the number of predicates that need to be tested.

Without causality analysis, the number of predicates that must be tested tends to increase roughly in proportion to the number of solutions tried. Since AutoBash tries the solution from the application with the most failed predicates in the initial predicate testing, AutoBash is able to find the right solution in an average of 4.5 trials. Also, AutoBash tests the failed predicates first after trying a solution so it is able to quickly determine if the solution is incorrect.

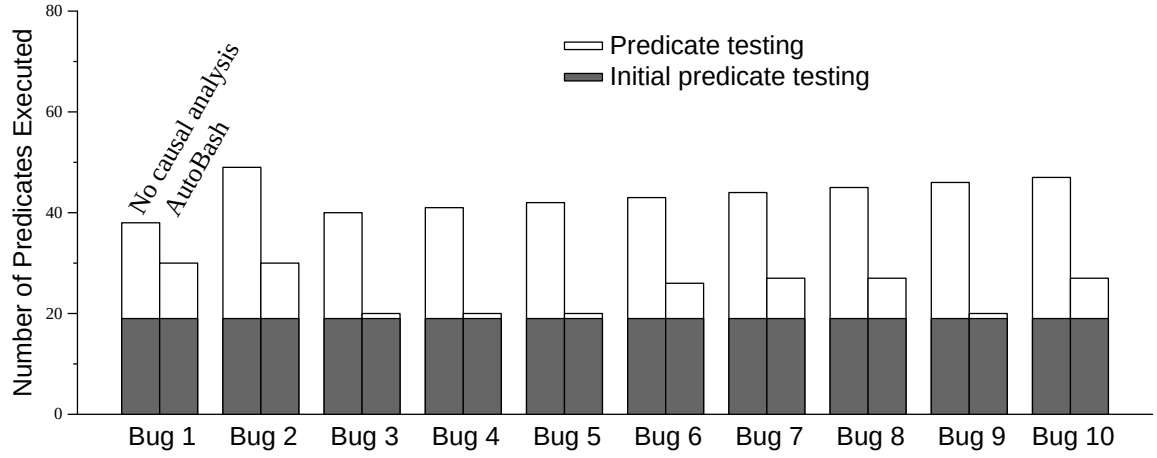
An interesting observation from our evaluation is that the subtlety of the bug impacts execution time — for instance, bug 9 allows unauthorized access to the repository; therefore, even though AutoBash only needs to re-run one predicate to determine a solution does not work, it takes longer for that predicate to detect a problem. However, bugs 1 and 5 are catastrophic; since all CVS actions fail, any predicate fails immediately.

Comparing the second and third bar in each dataset shows the performance benefit of causality analysis. For most bugs, causality analysis reduces the time to find a solution by 31–51% and predicate testing time by 67–82%. Figure 3.5 clarifies this



This figure shows the time to find a solution for the 10 GCC bugs in Table 3.2. The left bar in each data set shows the time to find a solution without speculative execution or causality tracking, the middle bar shows the time with only speculative execution, and the right bar shows the time with both. Each result is the mean of five trials — the error bars show 95% confidence intervals.

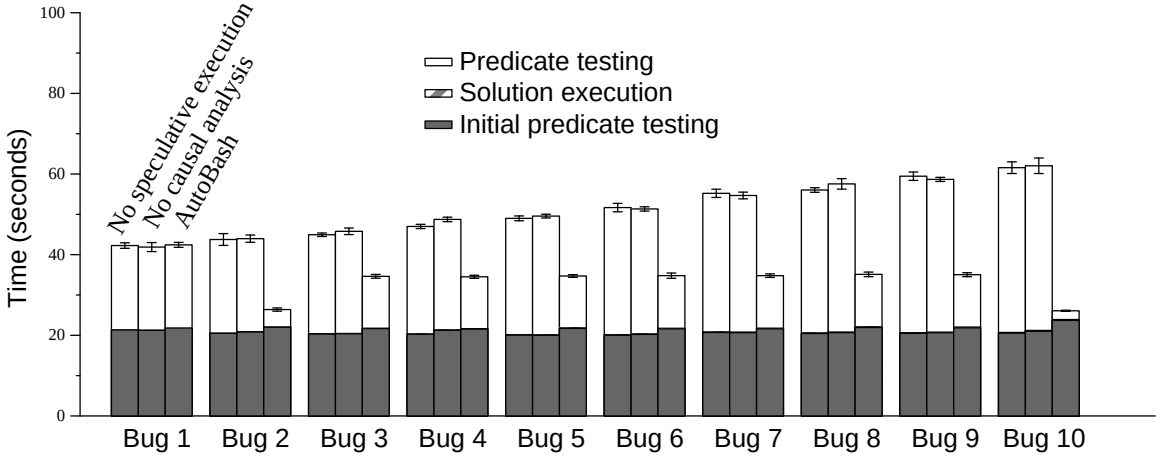
Figure 3.6: AutoBash performance for GCC benchmark



This figure shows the number of predicates executed by AutoBash while finding a solution for the 10 GCC bugs in Table 3.2. Five trials were run for each bug; however, the number of predicates executed was identical in each trial.

Figure 3.7: Number of predicates executed for GCC benchmark

benefit by comparing the number of predicates run with and without causality analysis. With causality support, AutoBash runs fewer predicates. Thus, the performance of AutoBash tends to degrade much more slowly with the number of solutions tried.



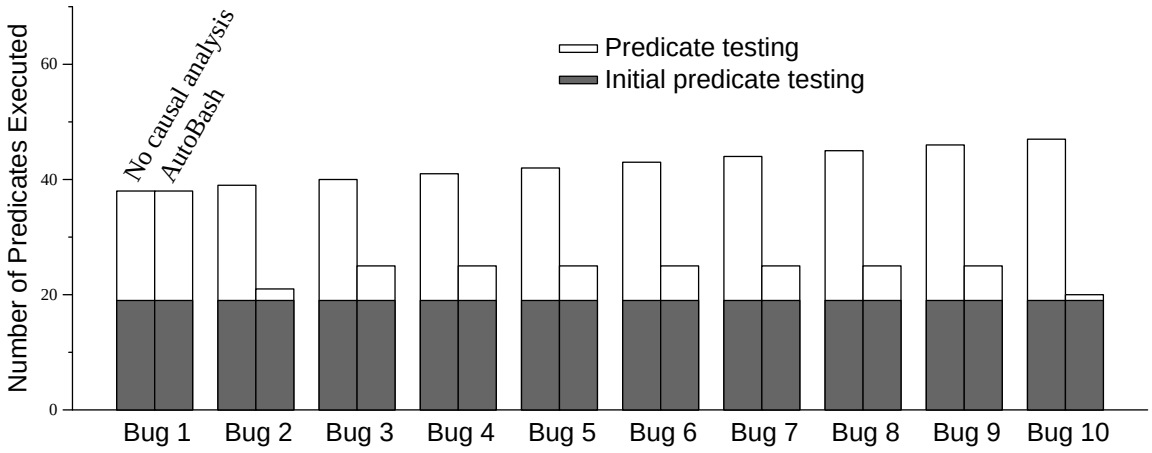
This figure shows the time to find a solution for the 10 Apache bugs in Table 3.2. The left bar in each data set shows the time to find a solution without speculative execution or causality tracking, the middle bar shows the time with only speculative execution, and the right bar shows the time with both. Each result is the mean of five trials — the error bars show 95% confidence intervals.

Figure 3.8: AutoBash performance for Apache benchmark

3.5.2.2 GCC cross compilation

Figures 3.6 and 3.7 show results for the ten GCC cross compiler bugs in Table 3.2. The performance impact of speculative execution, as shown by the difference between the first two bars in each dataset in Figure 3.6, is within experimental error for all bugs. For bug 2, which overwrites the default path environment variable, AutoBash is able to identify that the bug affects CVS and re-runs CVS predicates. As with the CVS experiments, without causality support, the number of predicates that need to be tested to find a solution tends to increase with the number of solutions tried, and the total execution time is affected by the subtlety of the injected bug. For most bugs, causality analysis improves solution search time by 34–49%.

An interesting observation highlighted by our cross-compilation evaluation is that predicate testing time for different applications can vary greatly. For instance, the cross-compilation predicates run much faster than Apache's. With causality analysis we improve predicate testing time by 67–99% because AutoBash did not re-run any Apache predicates as they were not causally dependent on any of the cross-compilation solutions.



This figure shows the number of predicates executed by AutoBash while finding a solution for the 10 Apache bugs in Table 3.2. Five trials were run for each bug; however, the number of predicates executed was identical in each trial.

Figure 3.9: Number of predicates executed for Apache benchmark

3.5.2.3 Apache Web server

Figures 3.8 and 3.9 show results for the ten Apache bugs in Table 3.2. The time to find a solution without causality tracking scales roughly linearly with the number of bugs because Apache has a large amount of configuration state. Thus, the bugs we injected are mostly subtle; a misconfiguration often affects only a single predicate or two. The solution for bug 1 changes the permission of /home/USER, where all our predicates are located. Therefore, the output set of the solution for bug 1 intersects with the input sets of all 19 predicates. So, bug 1 does not benefit from causality analysis as all 19 predicates need to be run. However, bug 10 shows the largest benefit — its total execution time decreases by 58% and predicate testing decreased by 95% because causality analysis reveals that only one predicate needs to be re-run.

3.5.3 State delta

We ran two interactive applications to see how well AutoBash understands the state delta. For each task, AutoBash requests Speculator to track the application's output set and queries Speculator for the output set. AutoBash then performs the same optimization described in Section 3.3.4 to generate a state delta. For each

application, we present the task we performed, its state delta, and the numbers of system call using `strace` to perform the same task.

3.5.3.1 Configuration menu: make menuconfig

The first task is to configure a Linux kernel source using `make menuconfig`. We ran `make menuconfig` in a Linux kernel source tree, changed a kernel compilation option and saved the changes. Table 3.4 lists the state delta for this task. From the state delta captured by AutoBash, we can conclude that "make menuconfig" created several object files, modified a header file according to the configuration file and changed the configuration file itself. There are a total of 16 directory entry modifications, 1 directory created, and 13 files modified.

We performed the same task and ran `strace` to capture the system calls made. There are around 16,065 system calls made. Some system calls change the system state during the task but do not make permanent system state changes. The number of system calls made is much larger than the state delta, thus it is much harder to reason about the changes made to system state.

3.5.3.2 Editor: vi

The second task is to create and edit a new file using `vi` editor. We used `vi` to open a new file and edited the file. Table 3.5 shows the state delta. Similarly, we ran `strace` for the same task, and 481 system calls were made. The number of system calls is dependent on the duration `vi` is opened as the `vi` process repeatedly executes the select system call to read input from the standard input.

3.5.4 Case study

Configuration problems involving multiple applications can be more difficult to solve. We investigate the effectiveness of AutoBash in solving such problems by developing a misconfiguration scenario involving both CVS and the Apache Web server. In our scenario, the user sets up a CVS repository to manage the source tree

Number	Directory entry modifications
1	/home/yysu/src/autobash/linux-2.4.21-15.EL/include/asm added
2	/home/yysu/src/autobash/linux-2.4.21-15.EL/include/linux/.tmpconfig.h added
3	/home/yysu/src/autobash/linux-2.4.21-15.EL/include/linux/modules added
4	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/checklist.o added
5	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/inputbox.o added
6	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/lxdialog added
7	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/lxdialog.o added
8	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/menubox.o added
9	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/msgbox.o added
10	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/textbox.o added
11	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/util.o added
12	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/yesno.o added
13	/home/yysu/src/autobash/linux-2.4.21-15.EL/.config.old deleted
14	/home/yysu/src/autobash/linux-2.4.21-15.EL/.tmpconfig added
15	/home/yysu/src/autobash/linux-2.4.21-15.EL/.tmpconfig.h added
16	/home/yysu/src/autobash/linux-2.4.21-15.EL/autoconf.h deleted
Number	Directory modifications
1	/home/yysu/src/autobash/linux-2.4.21-15.EL/include/linux/modules created
Number	File modifications
1	/home/yysu/src/autobash/linux-2.4.21-15.EL/include/asm created
2	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/checklist.o created
3	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/menubox.o created
4	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/textbox.o created
5	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/yesno.o created
6	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/inputbox.o created
7	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/util.o created
8	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/lxdialog.o created
9	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/msgbox.o created
10	/home/yysu/src/autobash/linux-2.4.21-15.EL/scripts/lxdialog/lxdialog created
11	/boot/kernel.h modified
12	/home/yysu/src/autobash/linux-2.4.21-15.EL/.config created
13	/home/yysu/src/autobash/linux-2.4.21-15.EL/include/linux/autoconf.h created

Table 3.4: State delta for `make menuconfig`

Number	Directory entry modifications
1	/test/autobash/autobash/group added
2	/test/autobash/autobash/group~ deleted
Number	File modification
1	/test/autobash/autobash/group created
2	216285 deleted

Table 3.5: State delta for `vi`

for a website. The configuration bug is that the user did not turn off CVS's keyword substitution feature so that whenever a developer checks in a CGI Perl script, CVS automatically substitutes keywords such as Id in the script with the username of the user who checks in the file. Later, when the user checks out the website and executes the CGI Perl script, he will encounter an incorrectly formatted HTML page.

This bug can be resolved by running AutoBash in replay mode. We hypothesize an existing predicate in a standard predicate set shipped with Apache that works as follows: an Apache user checks in a Perl script to a CVS repository on the machine. The predicate, which runs as root, checks out the Perl script into a CGI scripts directory, `wget`s the results of executing that CGI Perl script, and `diff`s the output with the expected output.

We ran AutoBash in replay mode with the solutions for bugs described in Table 3.2 and predicates in Table 3.3. During initial testing, AutoBash finds that CVS predicate 5 fails and the remaining predicates succeed. Therefore, AutoBash tries solutions associated with CVS first and finds the solution that fixed CVS's bug 3 solves the problem. Even though the misconfiguration manifests as an Apache problem, AutoBash is able to identify that it is actually due to a misconfiguration in CVS.

3.6 Discussion

One important design chose we made is to leverage operating system level checkpoint/rollback to try potential solutions and execute predicates to test. Virtual ma-

chines can also provide checkpoint and rollback capability for both transient state (memory) and persistent state. The advantage of choosing virtual machines is that AutoBash can be deployed to different operating systems without kernel modifications. Since a virtual machine is a layer below the operating system, it also enables us to debug kernel-related configuration problems similar to Chronus. The disadvantages of using virtual machines is the "semantic gap." Virtual machine rollback restores transient state, including all running processes, to a prior state. This means that users would lose other useful work that does not have a causal relationship with configuration activities. OS-level checkpoint and rollback tracks the information flow among processes and can rollback only processes that have observed or been affected by configuration activities. Also, due to the semantic gap between the virtual machine monitor and its host operating system, introspection techniques [47] are required to understand the causal relationships between the executions of solutions and predicates.

3.7 Summary

In this chapter, we demonstrate how we can make flexible software systems easier to manage through automated configuration management. AutoBash is a set of tools that leverage operating system support for speculative execution and causality tracking to automate many of the time-consuming tasks that occur when users deal with the complexity of modern computer systems. Our results show that OS support can reduce both the time and user effort needed to fix configuration errors.

CHAPTER 4

Inferring Predicates and Solutions from User Traces

The previous chapter showed that AutoBash makes configuration management easier if AutoBash has a set of predicates to test system correctness and a set of solutions that modify system state. When predicates and solutions do not exist, users need to spend a lot of effort to manually generate them. This chapter addresses that problem by automatically generating predicates and solutions.

4.1 Introduction

In the previous chapter, we manually wrote predicates for three applications to evaluate AutoBash's ability to find solutions for them. We found the process of manually writing predicates tedious and time consuming. Writing predicates requires domain knowledge about the application and is more likely to be done by expert users. Also, predicates created by experts might not be applicable to systems that are heavily customized. This chapter tries to reduce the time and effort spent on manually writing predicates by answering the following question: Can predicates be automatically generated?

Test cases check if certain requirements are satisfied in a program and can be used as predicates in AutoBash. If given the source code or specification of a software program, we could adopt automated test generation systems to generate predi-

cates [18, 27, 31, 42, 54, 64, 74]. However, source code is not always available for every application and specifications are even less likely to be available to the end users. In this work, we assume that source code or specifications are not available and treat a software program as a black-box.

We take a different approach by generating predicates from user observation. Users often need to troubleshoot misconfigurations. Consider the typical actions a user takes to fix a configuration problem: the user would examine the system state, search on the Internet or query colleagues for potential solutions, then apply the most likely solution, and test if the solution works. Our observation is that users often test whether the problem is solved during the troubleshooting process. If we can isolate the actions invoked by the user for testing, we can use them as predicates.

Identifying suitable actions as predicates is only the first step. AutoBash relies on a predicate to return true (or evaluate to true) when the system it is testing is correct, and otherwise returns false. Therefore, after generating predicates from user actions, we then need to determine if a predicate should return true or false.

Before we describe our solution to automatically generate predicates, we list the assumptions we make about user actions. Broadly speaking, user actions can be either commands typed in a shell or events entered in a window system, such as mouse clicks and keystrokes. In this work, we limit actions to only commands the user types in a shell because predicates composed of commands are easier to understand than predicates composed of window events.

To find commands that can be used as predicates, we observed two properties of these commands that test system state. First, users test the system state multiple times while fixing a configuration problem. Troubleshooting misconfiguration is an iterative process and the user often has to try many solutions to fix the problem. In order to test whether a solution has succeeded in fixing a problem, users reuse the same set of commands. Therefore, commands that are repeatedly executed can indicate a predicate. Second, users often rely on *output* from testing commands to determine if a solution works. The output we examine includes exit codes, screen output, and the output set for a command as defined in Section 3.3.2.1. For example,

if a user tests whether a local Web server is serving data correctly by using `wget` to retrieve a specific Web page, one observable output the user relies on is the screen output. If `wget` displays the progress of downloading the page, the user knows that the Web server is up and running. If `wget` displays HTTP error messages, the user knows that the Web server is not working correctly. Commands such as these with observable output are good candidates for predicates.

To evaluate the effectiveness of the above approach, we first conducted a user study to collect traces of users fixing configuration problems for CVS and the Apache Web server. We then employed the above approach to generate predicates from traces collected during the user study. The results showed that we were able to generate correct predicates for at least half of the traces for both CVS and Apache traces. We also show that we can identify solutions by aggregating traces that fixed the configuration problems.

We first describe an overview of our approach to generate predicates in Section 4.2 and present a user study in Section 4.3. We discuss our first approach to inferring predicates in Section 4.4 and describe why that approach did not work. We then present how we reached our final approach in Section 4.5 and provide details of our final approach in Section 4.6. Finally, we present our evaluation results in Section 4.7.

4.2 Overview of our approach

Our goal is to infer predicates from user traces so they can be used by AutoBash. We first discuss the user traces we used in this work and then describe details of our approach to generate predicates. Finally, we present a summary of our evaluation results.

We decided to collect our own traces of users fixing specific software configuration problems, as we did not find any existing traces that suited our needs. We divided user trace collection into two phases. The first phase was to record the series of commands a user typed when fixing a configuration problem, and the second phase was to use Speculator [62] to collect causal information for them. The reason behind this two-

phase process is that user studies are very time consuming to conduct, and simplifying them as much as possible while still achieving the same goal is recommended. The most essential input we need from the participants is how they troubleshoot and fix misconfigurations, such as what files they examine and how they test if the problem is solved. Causal information can be collected by re-running the commands a participant has typed after the study, and, hence, does not need to be collected during the study.

In the second phase, we replayed the commands each participant typed using AutoBash's observation mode to collect the causal input and output sets (defined in Section 3.3.2) for each command. The causal input set is the set of kernel objects a command comes to depend on during its execution. The causal output set is the set of kernel object a command has causal effects on during its execution. We also modified AutoBash's observation mode to collect the exit code and screen output (including both standard output and error) for each command.

To find commands that can be used as predicates from user traces, we implemented a *base algorithm* that searches for repeated commands with different observable output. To be more specific, the base algorithm uses three *output features* to represent the observable output for a command: the exit code, the error messages contained in the screen output, and the output set. The base algorithm marks repeated commands that differ in at least two output features as predicates. Differences in these output features suggest that the command was executed before and after the system state transitioned from an unhealthy state to a healthy one. To determine if a predicate should return true or false, the base algorithm assumes the last execution of a predicate to return true. For each execution of the same predicate, if it has at least two different output features from the last execution, that predicate returns false. Otherwise, the predicate returns true.

To verify the effectiveness of the base algorithm, we ran the base algorithm on the user traces we collected. We noticed that the base algorithm is very effective in finding predicates that contain a single command. However, some predicates require *preconditions* to be executed first. Preconditions are a set of commands that need to be executed to change the system state before a predicate can be used. For

example, if a predicate contains a `cvs checkout test_module` to demonstrate a CVS configuration problem, it is not usable unless the previous successful `cvs import test_module` is included as a precondition. This is because the predicate would always fail if the `cvs import test_module` is not executed first.

We solved the problem of finding preconditions for a predicate using causal information between commands. We call our solution the *precondition heuristic*. The precondition heuristic searches for commands that have causal effects on all executions of a predicate. It uses two rules to determine if a command is a precondition for the predicate or a solution. First, the heuristic marks commands that affect both successful and failed predicates as preconditions. Second, the heuristic marks commands that have causal effects only on successful predicates but that occur chronologically after all failed predicates as solutions. The insight is that solutions are the actions that cause a predicate to return a different value.

To see how well the precondition heuristic can find solutions, we ran the precondition heuristic on the user traces. We noticed that when the base algorithm generates predicates that do not test the specific configuration problem, the solution found by the precondition heuristic is not a correct solution. To better find the solution for a specific problem, we developed a *solution-ranking heuristic*. The solution-ranking heuristic aggregates solutions found across multiple successful user traces and ranks each solution according to its popularity. A solution that occurs in more successful traces is more popular and therefore ranked higher.

Finally, to evaluate the effectiveness of our approach, we combined the base algorithm and the precondition heuristic to generate predicates from the user traces collected in the user study. We were able to generate correct predicates for four out of ten traces of users fixing CVS configuration problems and six to eight correct predicates out of eleven traces of users fixing Apache configuration problems. Our approach works better for Apache configuration problems because single-command predicates are sufficient to test the Apache Web server. The evaluation results also show that the correct solution is ranked the highest for all configuration problems.

4.3 User study methodology

In this section, we describe the user study we conducted to collect user traces. We describe the user study setup and procedure, the tasks given to the participants, and the participants in our study. Finally, we describe the data we collected.

4.3.1 Experimental setup and procedure

We conducted our user study on a Dell Precision 370 desktop computer with a 3.00 GHz Pentium 4 processor and 2 GB of memory. The computer runs a Red Hat Enterprise 3 Linux kernel version 2.4.21. We ran VMware Workstation 6.2 to instantiate virtual machines for each experiment. We used Freecorder Toolbar [8] to record users' audio during each experiment.

This user study consisted of twelve participants. Each participant was given six tasks. Each task lasted up to fifteen minutes and required a participant to fix a configuration problem. We broke these six tasks into two sessions, each with three tasks. In each session, one task was used for training participants and the other two tasks were used for the study.

Each task ran in a virtual machine. We first created a base virtual machine image with RedHat Enterprise 3 Linux kernel version 2.4.21 installed. We then installed all default software packages included in the installation CDs to the base virtual machine image and created necessary users and groups. We disabled external network access for each virtual machine, as Speculator only allows network access within the same machine. Since no network access was available for a virtual machine, we provided participants with a laptop computer to browse the Internet for information. We created a virtual machine image for each task from the base VM image and injected each with a configuration problem. For each task, each participant was provided with a machine running a copy of the VM image with the injected problem.

Participants interacted with the virtual machine through our modified `autobash` shell. This is a modified `bash` shell similar to AutoBash's observation mode, described in Section 3.3.3, but does not invoke Speculator to record causal information or exe-

cute any predicates. It records user-typed commands, screen output, and commands to a file. The exact procedure, documentation, and survey used for this study are shown in Appendix A.

4.3.1.1 Training process

Upon arrival, each participant was given a consent form and a brief introduction to the user study. Each participant was asked to fill out a survey assessing their experience in configuration management and given a handout that described the scenario in the study. The scenario was as follows: the user is a new hire in an IT department and is given several configuration problems to solve. Their goal was to fix these configuration problems so that the configuration met the requirements described in the handout.

During the training process, the root cause of the misconfiguration was given to the user and the user only needed to determine and apply a solution to the problem. After fixing each problem, the participant was asked to label each command into one of the three categories: fixing, testing, or other.

4.3.1.2 Testing process

During the testing process, each participant was given two configuration problems. No information was provided about the configuration problem itself. Participants were allowed to use any software packages available in the virtual machine and fix the configuration problem as they saw fit. Participants were given a time limit of fifteen minutes for each task. Similar to the training process, participants were asked to label each command after finishing a task.

4.3.2 Misconfiguration tasks

We chose the CVS version control system and the Apache Web server for our study and selected two configuration problems from Table 3.2 in Chapter 3. We selected configuration problems 1 and 2 for the CVS version control system and

problems 1 and 2 for the Apache Web server. We did not quantify the difficulty level for configuration problems in Table 3.2, but we consider these four problems to be the easiest. We deliberately chose more straightforward problems because our participants possessed a wide variety of experience levels. We hoped to have some traces of participants successfully fixing problems. In our first design of the user study, we selected more subtle configuration problems and invited a few volunteers to try to solve them. Our volunteers commented that these problems were so subtle that even the most experienced users we could find might not be able to fix them. Therefore, we selected more straightforward configuration problems for our final user study.

We used configuration problems not listed in Table 3.2 for the training process. For the CVS training task, we created a script that added an unauthorized user to the CVS group for the training task in the CVS session. For the Apache training task, we manually commented out a line in the configuration that instructs Apache to permit CGI execution. These two problems are straightforward to fix.

4.3.3 Participants

Users with differing experience levels in system administration all face configuration management tasks. We modeled this by recruiting participants that ranged from system administrators for on-campus IT departments and research groups to graduate students with basic Unix and shell knowledge.

A total of twelve users with varying skills participated in our study: ten graduate students and two operations staff members, all from the University of Michigan. To understand the experiences in the participant pool, we used the self-assessment survey to divide participants into three categories: novice, intermediate, and expert. Table 4.1 is a summary of our participants. This categorization was also used in the study conducted by Nagaraja *et al.* [60]. A user is considered novice if he knows how to use that application, an intermediate if he has configured that application before, and an expert if he has diagnosed and fixed misconfigurations several times for that

User	CVS version control			Apache Web server		
	Experience level	Prob 1 Fixed?	Prob 2 Fixed?	Experience level	Prob 1 Fixed?	Prob 2 Fixed?
A	Expert	N/A	Y	Expert	Y	Y
B	Novice	Y	Y	Intermediate	N	Y
C	Intermediate	Y	Y	Novice	Y	Y
D	Expert	Y	Y	Expert	Y	Y
E	Beginner	N	N	Expert	Y	N
F	Intermediate	Y	Y	Expert	Y	Y
G	Novice	Y	N/A	Beginner	N	N
H	Intermediate	Y	Y	Expert	Y	Y
I	Intermediate	Y	Y	Expert	Y	Y
J	Novice	Y	Y	Expert	Y	Y
K	Expert	Y	N	Intermediate	N	Y
L	Intermediate	Y	Y	Novice	N	Y
Total fixed		10	9		8	10

This table shows the summary of the distribution of the level of user experience and the number of problems solved. We categorize users into novice, intermediate, and expert levels.

Table 4.1: Participant summary

application. User E's experience in CVS and user G's experience in Apache Web server did not meet the novice definition and were therefore labeled as "Beginner" in Table 4.1. We excluded user E's CVS traces and User G's Apache traces from the evaluation in Section 4.7.

Table 4.1 shows that almost all experienced users fixed both configuration problems with two exceptions: user E did not fix Apache Web server configuration problem 2 and user K did not fix CVS configuration problem 2. Both were due to users not reading the handout carefully: User E tried to fix a configuration problem not specified as a requirement in the handout, while user K did not see that two authorized

users should be in the CVS group and only tested for one. Two cases are noted as not applicable (user A's CVS configuration problem 1 and user G's CVS configuration problem 2) because we we had copied a wrong virtual machine image for one, and the user declined to do it for the other.

Overall, we were satisfied with the number of traces we captured that fixed the problem. We had expected that some participants would not fix all problems and had five traces (one CVS problem 2 trace and four Apache problem traces) that did not solve the problem presented.

4.3.4 Data collected

We collected four different kinds of data for each trace so that we would have sufficient information to replay user traces after the study. The data we collected were:

- Commands: We collected commands that participants typed, including the executable and the arguments. This information is similar to output displayed by `history` in `bash` shell.
- Virtual machine recording: We turned on VMware's Record/Replay to record and to capture the complete execution behavior. When we later manually re-typed commands that a user has typed in to collect causal information, we could first use VMware's recording to replay the exact and complete execution behavior of the user's trace. This allows us to see what a user entered into an interactive application, such as an editor.
- User-specified labels: At the end of each task, we asked users to specify for each command if it was used for testing, fixing, or other purposes. We collected these user-specified labels for each trace. We did not use this data in our final solution but had considered using it when we started our study.
- Self-assessment surveys: We collected the surveys completed by participants to categorize the experience level of each participant.

4.4 Our first approach to inferring predicates

When we first started, we unsuccessfully took an approach that tried to estimate the commands each participant stated they used for testing. Our failed approach was to develop heuristics that could identify *all* commands labeled as "testing" by participants. Our belief was that these user-specified labels were the most accurate predicates because users knew best how they tested the system. One example of a heuristic we used was to search for repeated commands that have screen output but an empty output set. This heuristic can find commands a participant used to examine system state, for example, `ls -l` the CVS repository to examine its user, group, and file permissions. Another example heuristic was to search for repeated commands that had different input sets or one that had its input set affected by another command. This heuristic tried to use these repeated commands that might have different output as predicates.

The main problem with this approach was that for many predicates, we did not know whether they should return true or false. For example, we did not know how to determine if a `ls` command that examines the CVS repository should return true or false. Predicates for which we could not determine an appropriate return value would not be useful to AutoBash. Also, user labels are often inconsistent.

4.5 Re-examine the predicate definition

After realizing this approach would not work, we first re-examined the predicate definition to understand why the approach failed. In Chapter 3, we defined predicates as configuration activities that test system correctness and a predicate should deterministically evaluate to true if the system is correct and false otherwise. Predicates include both a set of commands that can test system state and a return value, true or false, that indicates if the system state is correct or not. The main lesson learned from the failed approach is that many commands can be used for testing, but success or failure is not always easily determined. For example, it is difficult to determine if

`ls -l` should evaluate to true or false without semantic knowledge of the directory being listed.

To find an approach whose results are easy to determine, we first examined how users fixed configuration problems. Based on our observation during the user study, users often took an *action-based approach*, a *state-based approach*, or a combination of both to troubleshoot a configuration problem. An action-based approach means that the user interacts with the misconfigured application to learn the behavior of the application. For example, when troubleshooting a CVS problem, the user would `cvs import` a new module or `cvs checkout` a module. The user learned about the misconfiguration by examining the error messages and progress-reporting messages displayed on the screen by CVS. A state-based approach means that the user passively examines the relevant state the application depends on. For example, when troubleshooting the same CVS problem, the user would see if the CVS repository directory exists, what the CVS repository file permissions are, the user and group information, etc.

Commands used in both action-based and state-based approaches can be used for testing. We observe that if we use commands used in an action-based approach to be predicates, it is often easier to determine what their return value should be. When the system state is incorrect, processes that execute these commands often have different return values or display error messages. On the other hand, commands used in a state-based approach are often more difficult to evaluate because determining correctness requires semantic information or knowledge about the problem itself to explain about observable output.

To better explain why it is easier to determine the return values of commands used in an action-based approach, we classified commands according to two axes:

- **Can it be used as a predicate?** Users often use many commands in the process of fixing a configuration problem. Broadly speaking, commands are used for three different purposes: testing the system state, fixing a problem, and facilitating the troubleshooting process. Commands that are used to test the system state can be used as predicates. Both commands used in state-

	Easy to determine result	Difficult to determine result
Used as predicates	`wget`[1] `cvs`[1]	`ls`[2] `groups`[2]
Not used as predicates	`chmod` a file's permission `usermod` a user's group	edit configuration files clear the terminal screen

[1] The `wget` and `cvs` commands return different exit values indicating success or failure.

[2] The `ls` and `groups` commands print different screen output that reflect the current state, but they return the same exit codes and both have an empty output set.

This table shows examples of various commands that fall into one of the four possible combinations. This thesis focuses on finding commands that are both used for testing and can easily determine the result without application-specific semantic information (the shaded cell).

Table 4.2: Command taxonomy

based and action-based approach fall into this category and can be used as predicates. Commands that are used to fix the configuration problem cannot be used as predicates. If these commands are used as predicates, the system state would always be correct because in the process of testing system state, the predicates fix the configuration problem. Commands that are not used for fixing or testing are used for facilitating. Examples of facilitating commands include `clear`, which clears the terminal screen, and `man`, which displays manual pages for commands. These commands cannot be used as predicates because they do not give relevant information about the system state.

- **Is it easy to determine if a predicate returns true or false?** Commands used in an action-based approach often have some good indicators and produce output that make it easy to determine their results. For example, if a command returns a zero exit code on success and non-zero exit code on failure, the exit code is a good indicator that different return values exist for this command. It is often difficult to determine the returning result of commands used in a state-based approach because they often display unstructured screen output and do not have good indicators, such as exit codes. One example is the command

`groups`, which prints the groups the user is in. When a user executes `groups` to test if a user is in the right group, the user knows what the expected output should be. If the user executes `groups` once before applying a solution and once after to test if the user is in the right group, classifying the screen output is difficult without semantic knowledge. Another example is the command `ls`. If the user executes `ls` to examine the permission of a specific file, it is very difficult to know which file the user is looking for and what the expected permission for that file is.

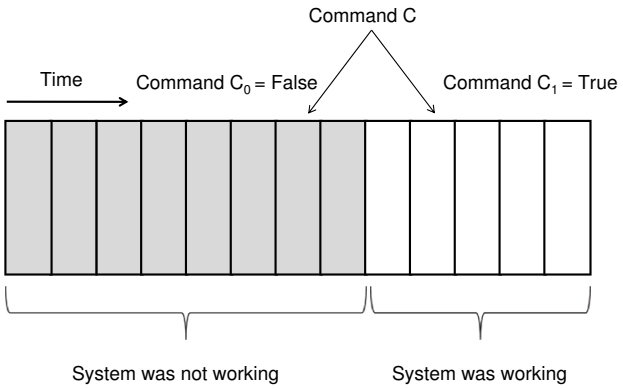
Table 4.2 shows the four possible combinations of these two classifications along with examples of commands that satisfy each combination. Our goal is to identify commands that fall into the shaded cell, and these commands are mostly used in an action-based approach.

4.6 Generating predicates and solutions from user traces

This section describes the algorithm and heuristics we implemented to generate predicates with results that are easy to determine. We first describe the base algorithm, then give details of the heuristics we used to improve the base algorithm.

4.6.1 Base algorithm

We started out by assuming a predicate contains only one command. Our *base algorithm* searches for repeated commands within a trace that differ in at least two out of three output features: exit code, error messages contained in the screen output, and the output set. The rationale is that if a repeated command has different output features, as shown in Figure 4.1, one of them could return true and the other could return false. Therefore, repeated commands with different output features are good predicate candidates. This is similar to how Chronus [82] used probes to search for the time when the system state transitions from a working to a non-working state.



This figure shows a command C executed twice; the first time is shown as C_0, and the second is shown as C_1. If C_0 and C_1 have different output features, C_0 would return false and C_1 would return true. C is a good predicate for AutoBash as it has different output features as evidence that it is executed in a non-working and a working state.

Figure 4.1: The rationale behind the base algorithm

We used the repeated commands as our probes and the different output features as evidence that the system state transitions from a non-working to a working one.

The base algorithm uses the following three output features:

- **Exit code is zero or non-zero.** When a shell creates a process to execute a command, the return value of the process when it exits is returned to the shell and is often called the exit code. Conventionally, a command returns a non-zero exit code to indicate failure and zero to indicate success. Mickens *et al.* [56] used exit codes to mark user traces as "good" or "bad" traces and discovered that application developers do not always follow this convention.

- **If the screen output contains an error message.** Human beings are good at understanding screen output that gives feedback on the progress or result of executing a command. However, the screen output may contain unstructured text that is difficult for computers to analyze. Our hypothesis is that we only need to search for certain positive or negative keywords on the screen output to determine if a repeated command has different output. Searching for specific words is much simpler than trying to understand arbitrary screen output from the application, and it is often sufficient to know whether the two executions of

a repeated command are different.

To search for keywords, we first need to intercept screen output displayed to the user. We modified AutoBash's observation mode to make a *typescript* of the terminal session into a log file. We implemented this using Unix's pseudo-terminal interface, and this implementation is transparent to the user. When AutoBash's observation mode (a modified `bash` shell) is in the initialization phase, it first finds an available pseudo terminal and forks a child process. The parent process provides the input and output interface with which the user interacts while the child process handles executing commands the user types. The pseudo terminal is a pair of character devices (master and slave) that provides a bi-directional communication channel. Any terminal input entered into the master end is automatically provided to the slave end. Any terminal output written to the slave end can be read by the process connected to the master end. We modified the parent process to read from standard input and write to a log file. The child process redirects its standard input, output, and error to the slave end of the pseudo terminal. This implementation is very similar to `script`.

For each command, the base algorithm reads from the log file for the screen output and searches for any error messages. Currently, the base algorithm searches for the system error messages (from errno.h) and the word "error." Two commands are considered to have different error message features if one contains an error message in its screen output and the other does not. We did not search for "positive" words as we did not find any viable candidates for such a word.

- **If the output sets are the same.** We collect the output set for each command in the same way as AutoBash's observation mode. We provide a brief description here for completeness. For each command, the AutoBash shell process first forks a child process and invokes Speculator to execute the command speculatively and track its output set. Speculator keeps track of the output set, which contains

kernel objects that come to depend on the process executing the command. When AutoBash finishes executing a command, it queries Speculator for the command's output set. For each execution of a repeated command, the base algorithm compares if the output set of one execution is the same as the output set of the last execution of the command.

When comparing commands, we noticed that strict string comparison may not be the best way to compare. Strict string comparison has two problems: the first is it does not know *who* is executing the command. User identity is important when managing configurations because a command executed as different user identities may return different results. We solved this problem by keeping track of a subset of environment variables, including the current working directory and user. Most shell programs, including `bash`, already keep track of environment variables, so we duplicated this functionality. When comparing commands, we first checked their user identities and considered them as not repeated if their user identities differed.

The second problem with using strict string comparison is that it does not take into account *environment variables* on which a process depends. The working directory in which a process is executed determines if the process has permission to create a file in that directory or list a directory. For example, the execution of `ls` depends on whether the process has permission to read the current working directory. Also, a process might depend on an environment variable specified by the user to find the default location of shared libraries or a configuration file. For example, `cvs` depends on the environment variable CVSROOT to point to the CVS repository and CVS_RSH to specify the protocol to log in to a remote CVS server.

We solved the second problem in a manner similar to the first problem. When comparing two commands, we also compare the working directory and the user-specified environment variables. Comparing the working directory might sometimes be too conservative. For example, creating a file in a user's home directory and in /tmp are both permitted. We chose to be conservative when comparing environment variables, as we prefer missing a few correct predicates over generating incorrect predicates. We do not have sufficient experience to know which environment variables should be com-

pared, but based on our evaluation in Section 4.7, comparing user identities, working directory, and user-specified environment variables generated correct predicates for at least 40% of the total traces we evaluated.

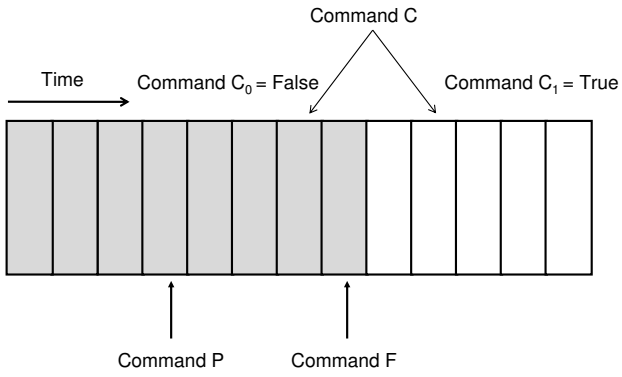
4.6.2 Determining the result for a predicate

After the base algorithm generates predicates, we then need to solve the problem of determining if a predicate should return true or false. We developed a *result-assigning heuristic* that takes as input the output features of all executions of a predicate and compares them. The result-assigning heuristic assumes that the last execution of a predicate *always* returns true. The reason we use this heuristic is that a predicate that returns a non-zero exit code and generates an error message should sometimes return true. Consider a configuration problem in which an unauthorized user is given too many permissions. The predicate that uses the excess permission to access data that should be unauthorized to access should return false when its exit code is zero.

For each predicate, the result-assigning heuristic starts by assigning success to the last execution and compares the last execution with each previous execution of the predicate. If at least two out of three output features are the same as the last execution, the heuristic assigns true as its return result. Otherwise, it assigns false.

4.6.3 Finding preconditions

The base algorithm is very effective in identifying single-command predicates that test the system state. However, we noticed that in some cases a single-command predicate relies on some prior commands to change the system state before they can work correctly. We call these prior commands on which a predicate depends *preconditions*. A single-command predicate that relies on preconditions cannot be used by AutoBash because it would always return the same result. For example, if a user fixed a CVS problem starting from an empty repository and the base algorithm identified `cvs checkout` to be a predicate, this predicate would work only if the command `cvs import` had first been executed. The `cvs import` is a precondition



This figure demonstrates how the precondition heuristic works. Assume that the base algorithm determines that C_0 returns false and C_1 returns true. If both a command P and a command F have causal effects on C_1 only, the precondition heuristic determines that the command P is the precondition for command C and the command F is the solution, as command F might be the solution that causes the state transitioning that causes C_0 and C_1 to return different results.

Figure 4.2: The reasoning behind the precondition heuristic

for that predicate because, without it, the predicate will always return false.

To find preconditions, we first search for commands that have causal effects on a predicate. In addition to including prior commands that have causal effects on predicates, we also include prior commands that add or remove environment variables. Since the base algorithm compares environment variables when comparing commands, adding or removing environment variables is considered to have an effect on commands. For example, `export CVSROOT=/home/cvsroot` is considered to have a causal effect on all later commands. This is more conservative than strictly needed, as environment variables could be added but not read by some or all later commands executed in the same shell.

Not all prior commands with causal effects on predicates are preconditions. If the user has successfully fixed the configuration problem, the solution command should have a causal effect on the last execution of the predicate. Including a solution as a precondition would cause a predicate to always return true, rendering the predicate ineffective.

To correctly identify preconditions, we developed a *precondition heuristic* that can differentiate between preconditions and solutions. The precondition heuristic first

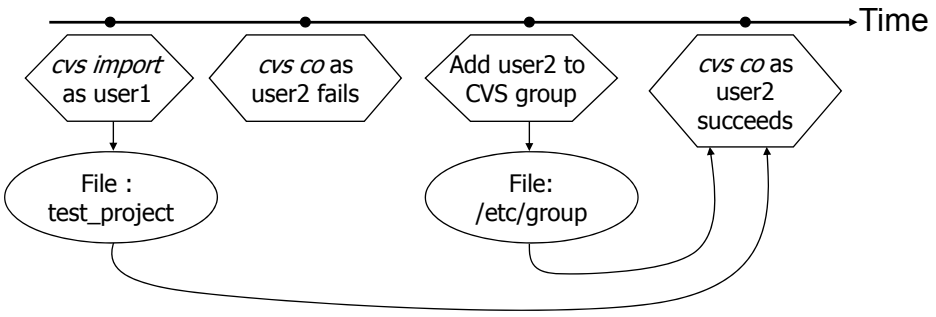


Figure 4.3: An example to demonstrate the precondition heuristic

finds all commands that causally affect all executions of a predicate. Within these commands, the heuristic uses two rules to determine if a command is a precondition or a solution. First, a command that has causal effects on both successful and failed predicates is a precondition. Second, a command that only has causal effects on successful predicates and is executed chronologically after all failed predicates is a solution.

Figure 4.2 shows the reasoning that allows the second rule to distinguish between a solution and a precondition. The reasoning is that users will usually first verify preconditions happen before finding out the root cause of a misconfiguration. Commands that occurred after failed predicates and before successful ones are solutions that cause predicates to have different output features.

If a precondition is executed after failed predicates, it has the same causal effect as a solution from the point of view of our heuristic. Thus, it is difficult to whether it is a precondition or a solution without information about the command semantics.

Consider an example in which both user1 and user2 are authorized to access a CVS repository, but only user 1 is in the CVS group. Figure 4.3 shows the four commands the user executes and the causal relationship between commands. First, the base algorithm would determine that "`cvs checkout` as user2" is a predicate. The first predicate execution is considered to return false and the second one is considered to return true. Both "`cvs import` as user1" and "add user2 to CVS group" commands have causal effects on the second predicate execution. The precondition uses the

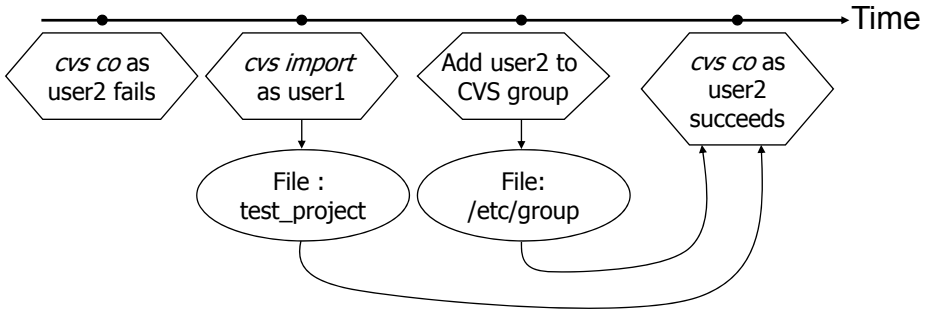


Figure 4.4: An example to demonstrate when the precondition heuristic does not work

chronological order to determine that "`cvs import` as user1" is a precondition and "add user2 to CVS group" is a solution.

However, consider the same example but the order of "`cvs import` as user1"and "`cvs checkout` as user2" are reversed as shown in Figure 4.4. Both "`cvs import` as user1" and "add user2 to CVS group" have causal effects on the successful predicate and were executed after the failed predicate. Without semantic information, such as `cvs` ordering constrains, the precondition heuristic would mark them both as part of the solution. We note that this is less likely to happen if the user already knows that the CVS repository is empty.

4.6.4 Inferring solutions from user traces

Solutions found by the precondition heuristic are not always the correct solution for the specific configuration problem the user is trying to solve. When predicates are not correctly identified by the base algorithm, the resulting solution found often does not fix the problem the user is trying to solve. To better identify the correct solution for a specific problem, we developed a *solution-ranking heuristic* that aggregates solutions found from multiple traces. In this heuristic, we assume that we have multiple traces of users successfully fixing similar configuration problems. This heuristic ranks each solution based on its *popularity.* A solution is more popular if more users apply

it to successfully fix the configuration problems.

One way to measure popularity of a solution is by the number of traces in which a solution occurs. However, a strict string comparison does not yield accurate results because many variants of a command can be used to successfully solve a problem. For example, when solving the Apache configuration problem caused by the Web server process not having search permission (the execution bit) on a user's home directory /home/testuser, solutions executed in the user study included `chmod 755 /home/testuser`, `chmod 755 testuser/`, and `chmod og+rx testuser`. These solutions all give the directory /home/testuser search permission, but they are different commands.

To better measure the popularity of a solution, we compare solutions by their *state deltas*. The state delta was first introduced in Section 3.3.4 to capture the difference in the state of the system caused by the execution of a command. For example, the state delta for command `chmod a+r test.pl` includes only the file permissions for test.pl changing from its original permission to a new permission. The solution-ranking heuristic first groups solutions based on state deltas so that all solutions having the same state delta are in the same group. It then sorts these groups by the size. The group with most solutions mapped to that state delta is ranked highest.

Here is an example of how the solution-ranking heuristic ranks the three solutions: `chmod 755 /home/testuser`, `chmod 755 testuser/`, and `chmod -R 777 testuser/`. The heuristic groups the first two commands into one group because they both have the same state delta, changing the permission of the directory /home/testuser to 755. It puts the third solution into a separate group. Thus, the solution with the same state delta as `chmod 755 /home/testuser` has group size two and is ranked higher than the other group with only `chmod -R 777 testuser/`.

4.7 Evaluation

Our evaluation answers the following questions:

- How well can we automatically generate predicates from user traces?

User	CVS problem 1 CVS repository not initialized			CVS problem 2 Authorized user not in CVS group		
	# of Pred	Correct?	Total # of commands	# of Pred	Correct?	Total # of commands
A	—	—	—	1	correct	44
B	1	correct	105	1	correct	44
C	0	SBA	57	0	NRC	46
D	0	SBA	49	0	SBA	26
F	1	correct	22	1	correct	30
G	0	NRC	61	—	—	—
H	0	NRC	58	0	NRC	74
I	0	NRC	18	1	correct	18
J	0	NRC	65	0	SBA	72
K	1	correct	55	0	DNF	24
L	1	correct	40	0	SBA	24

There are three major reasons for no predicate identified: (1) NRC means that the user did not use a repeated command to test the configuration problem. (2) DNF means that the user did not fix the configuration problem. (3) SBA means that the user used a state-based approach to fix the problem.

Table 4.3: Summary of predicates generated for CVS traces

- How well does the solution-ranking heuristic rank the solutions?

4.7.1 Methodology

We implemented the base algorithm and heuristics described in Section 4.6 using python and ran them on the traces collected from our user study. We generated predicates using both the base algorithm and the precondition heuristic.

We then took the solutions inferred by the precondition heuristic from successful traces and used the solution ranking heuristic to rank them. Table 4.9, Table 4.10, Table 4.11, and Table 4.12 show the solution ranking results.

User	Apache problem 1 Insufficient permission for user's home			Apache problem 2 Insufficient permission for CGI script		
	# of Pred	Correct?	Total # of commands	# of Pred	Correct?	Total # of commands
A	1	correct	45	1	correct	45
B	0	DNF	39	0	NRC	39
C	2	correct correct	41	1	correct	41
D	1	correct	68	1	correct	68
E	0	NRC	35	1	wrong: NOF	35
F	0	NRC	33	1	correct	33
H	1	correct	32	1	correct	32
I	1	correct	22	1	correct	22
J	0	NRC	40	1	correct	40
K	1	wrong: NOF	67	1	correct	67
L	0	DNF	55	0	NRC	55

There are two reasons for no predicate identified: (1) NRC means that the user did not use a repeated command to test the configuration problem. (2) DNF means that the user did not fix the configuration problem. NOF means that an incorrect predicate due to noise in output features, such as the file showed on the screen contained an error message.

Table 4.4: Summary of predicates generated for Apache traces

4.7.2 Quality of predicates generated

Table 4.3 and Table 4.4 show a summary for each predicate generated from each trace. For each trace, we list the number of predicates generated, the correctness of each predicate, and the total number of commands entered. We use an acronym that describes the reason why no predicate is generated or why a predicate is incorrect.

For both CVS problems, we generated correct predicates for four out of ten traces. We generated six correct predicates out of eleven traces (however, only nine traces actually fixed the problem) for Apache problem 1 and eight of out eleven traces for Apache problem 2. Overall, our predicate results for Apache problems are better

User	Predicate	
B	Precondition	`export CVSROOT="/home/cvsroot"` as testuser
	Predicate	`cvs import -m "Msg" yoyo/test_project yoyo start` as testuser
	Solution	`cvs -d /home/cvsroot init` as cvsroot
F	Precondition	`export CVSROOT=/home/cvsroot` as testuser
	Predicate	`cvs import test_project` as testuser
	Solution	`cvs -d /home/cvsroot init` as cvsroot
K	Predicate	`cvs -d /home/cvsroot import cvsroot` as testuser
	Solution	`cvs -d /home/cvsroot init` as cvsroot
L	Predicate	`cvs -d "/home/cvsroot" import test_project` as root
	Solution	`cvs -d /home/cvsroot init` as root

This table lists the predicates correctly identified from traces for CVS problem 1 (CVS repository not initialized). Each predicate contains one or more steps, and the last step is a repeated command. The first step of a predicate is listed as Precondition, and the last step is listed as Predicate.

Table 4.5: Predicates correctly generated for CVS problem 1 traces

because the predicates for Apache are mostly single-command ones. We next examine the predicate generated for each trace and explain in detail about when no predicates are generated or why some predicates are incorrect.

4.7.2.1 Predicates generated from CVS problem 1 traces

CVS problem 1 is due to the CVS repository not being properly initialized, and the solution is to initialize it using `cvs init`. The initial state included an empty directory, /home/cvsroot, intended to be the CVS repository. The initial state also included two authorized users, testuser and cvsuser, who have permission to access the CVS repository and one unauthorized user, noncvsuser, who does not have permission to access the repository. A test_project directory was created in the testuser's home directory.

Table 4.5 lists the predicates correctly generated for each trace and the solution

identified by the precondition heuristic. We break each predicate into two parts: *Predicate*, the repeated command identified by the base algorithm and the command that determines if this predicate returns true or false, and *Precondition*, the set of commands that must be executed before the predicate. We also list the *solution* found by the precondition heuristic. All correct predicates involve the user importing a module into the CVS repository. The base algorithm can find these predicates because they have different output features when executed before and after the solution is applied.

No predicate was found for six traces becauses users either used a state-based approach (users C and D) or did not use repeated commands (users G, H, I, and J). Both users C and D discovered that the CVS repository was not initialized by examining the repository directory. Users G, H, I, and J used slightly different commands to test the system state before and after fixing the configuration problem. For example, user H's two `cvs import` commands have different CVS import comments. This could be solved if we were to incorporate semantic information about the `cvs` command.

4.7.2.2 Predicates generated from CVS problem 2 traces

In CVS configuration problem 2, both testuser and cvsuser are authorized to access the CVS repository but only testuser is in the CVS group. The solution to this problem is to add cvsuser to the CVS group so that the user can access the CVS repository. The initial state included an empty CVS repository and a test_project directory created for testuser.

Table 4.6 shows the predicates correctly generated for CVS problem 2 traces and the solution identified by the precondition heuristic. All correct predicates involve cvsuser trying to check out a module from the CVS repository and need the precondition heuristic to find the first `cvs import`. Even though each user might use a different module name, the base algorithm and precondition heuristic can find the commands that import and check out the module.

No predicate was generated for six traces. We found three main reasons for this.

User	Predicate	
A	Preconditions	`cvs import test_project head start` as testuser
		`cvs co test_project` as testuser
		`export CVSROOT=/home/cvsroot` as cvsuser
	Predicate	`cvs co test_project` as cvsuser
	Solution	`vi /etc/group` as root
B	Precondition	`cvs -d /home/cvsroot import yoyo/test_project` as testuser
	Predicate	`cvs -d /home/cvsroot checkout yoyo/test_project` as cvsuser
	Solution	`usermod -G cvsgroup cvsuser` as root
F	Preconditions	`cvs import test_project` as testuser
		`cvs co test_project as testuser`
		`export CVSROOT=/home/cvsroot` as cvsuser
	Predicate	`cvs co test_project` as cvsuser
	Solution	`vim group` as root
I	Precondition	`cvs -d /home/cvsroot import test_project` as testuser
	Predicate	`cvs -d /home/cvsroot co test_project` as cvsuser
	Solution	`vi /etc/group`

This table lists the predicates correctly identified for CVS problem 2 traces (an authorized user, cvsuser, not added to CVS group). Each predicate contains one or more steps and the last step is a repeated command. The last step of a predicate is listed as Predicate and the first step is listed as Precondition.

Table 4.6: Predicates correctly generated for CVS problem 2 traces

The first reason is that the user did not use repeated commands to test (users C and H). User C used a `cvs co` as cvsuser to test if cvsuser had permission to access the CVS repository. The differences between the two uses of this command was that one specified /home/cvsroot as the repository and the other specified /home/cvsroot/. User H used the same command line but with different environment variables specified. The base algorithm is more conservative and compares *all* user-specified environment variables. Second, no predicate was found for traces in which the user did

User	Predicate	
A	Preconditions	`wget http://localhost` as root
		`chmod 777 index.html` as root
		`chmod 777 public_html/` as root
	Predicate	`wget http://localhost/~testuser/index.html` as root
	Solution	`chmod 777 testuser` as root
C-1	Predicate	`wget http://localhost/~testuser/` as root
	Solution	`chmod o+rx /home/testuser` as root
C-2	Predicate	`apachectl stop` as root
	Solution	`vim /etc/httpd/conf/httpd.conf` as root
		`chmod o+rx /home/testuser` as root
D	Preconditions	`wget localhost` as testuser
		`chmod -R 777 public_html/` as testuser
	Predicate	`wget localhost/~testuser` as testuser
	Solution	`chmod -R 777 testuser/` as testuser
H	Preconditions	`mkdir scratch` as root
		`wget http://localhost` as root
		`rm index.html` as root
	Predicate	`wget http://localhost/~testuser` as root
	Solution	`chmod 755 /home/testuser/` as root
I	Precondition	`wget http://localhost` as root
	Predicate	`wget http://localhost/~testuser` as root
	Solution	`chmod 755 /home/testuser/` as root

Table 4.7: Predicates correctly generated for Apache problem 1 traces

not fix the problem (user K). Since the user did not fix the problem, the repeated command does not have different output features. Finally, no predicate was found for user D's and L's traces because these users both used a state-based approach to fix the problem. User D executed `groups` and User L examined /etc/group to discover that cvsuser was not in the CVS group.

4.7.2.3 Predicates generated from Apache problem 1 traces

The Apache configuration problem 1 is caused by the Apache Web process not having search permission to testuser's home directory, and the solution is to give search permission (set the world execution bit) of testuser's home directory. The initial state included a public_html directory in testuser's home directory.

Table 4.7 shows the correct predicate identified for each trace. Almost all predicates are downloading the testuser's home page. Most preconditions found by the precondition heuristic are not required for the predicate to work but also do not affect the correctness of the predicates. Predicate C-2, `apachectl stop`, did not seem to be a predicate at first sight, so we examined why it was generated. We found that predicate C-2 successfully detected an error in the Apache configuration file introduced by the user. `apachectl` is a script that controls the Apache process and it does not stop the Apache process if an error is detected in the configuration file.

No predicate was generated for five traces for similar reasons seen in CVS traces. Users B and L did not fix the problem and hence no predicate was identified for their traces. Users E and F executed a command in different directories, so the base algorithm considered that command as not repeated. User F executed a `wget` command as root in /root and /home directory. The base algorithm is too conservative as the current directory does not affect the `wget` command.

One incorrect predicate was generated for user K's trace. User K did not fix the problem, so we first expected no predicate to be generated; however, the base algorithm generated one predicate that was incorrect due to noise in the output features. The predicate edits the Apache configuration file using `emacs` and is identified by the base algorithm because one execution of this command has a different error message feature and output set feature. The error message output heuristic was triggered because the word "error" was contained in one part of the configuration file and was sometimes displayed. The output sets varied between executions depending on whether the user edited the configuration file. We label this "noise in output feature."

User	Predicate	
A	Predicate	`wget http://localhost/cgi-bin/test.pl` as root
	Solution	`chmod 755 test.pl` as root
C	Predicate	`wget http://localhost/cgi-bin/test.pl` as root
	Solution	`chmod go+r test.pl` as root
D	Preconditions	`wget localhost/cgi-bin/test.pl` as testuser
		`vi /var/www/cgi-bin/test.pl` as root
	Predicate	`wget localhost/cgi-bin/test.pl` as root
	Solution	`vi /var/www/cgi-bin/test.pl` as root
		`chmod 777 /var/www/cgi-bin/test.pl` as root
F	Predicate	`wget http://localhost/cgi-bin/test.pl` as root
	Solution	`chmod 755 test.pl` as root
H	Precondition	`mkdir scratch` as root
	Predicate	`wget http://localhost/cgi-bin/test.pl` as root
	Solution	`chmod 755 /var/www/cgi-bin/test.pl` as root
I	Predicate	`wget http://localhost/cgi-bin/test.pl` as root
	Solution	`chmod 755 /var/www/cgi-bin/test.pl` as root
J	Predicate	`wget 127.0.0.1/cgi-bin/test.pl` as root
	Solution	`chmod +r test.pl` as root
K	Predicate	`wget http://localhost/cgi-bin/test.pl` as root
	Solution	`chmod a+r test.pl` as root

Table 4.8: Predicates correctly generated for Apache problem 2 traces

4.7.2.4 Predicates generated from Apache problem 2 traces

Apache problem 2 is due to the Apache Web process not having read permission for the CGI Perl script and therefore not being able to execute it. The solution is to give it read permission for that CGI script. The initial state included a CGI Perl script at Apache's default CGI directory, /var/www/cgi-bin.

Eight correct predicates were generated out of eleven successful traces. Table 4.8 shows these correct predicates. All correct predicates involve retrieving the CGI

Solution	Number of traces
`cvs -d /home/cvsroot init` as cvsroot	3
`cvs -d /home/cvsroot init` as root	1

CVS problem 1 is that the repository is not initialized and the correct solution is to initialize the repository using `cvs init`. Solutions are ranked by the number of traces that use it.

Table 4.9: Solutions ranked for CVS problem 1

script. Some traces include preconditions that are not technically required, but these preconditions do not affect the correctness of the predicates.

No predicates was generated for user B's and user L's traces because they executed `wget` commands in different directories. The working directory did not affect the correctness in these two cases but could cause `wget` to behave incorrectly when the process executing `wget` does not have sufficient permissions on the working directory.

Only user E's predicate was incorrect. User E did not fix the problem, but the base algorithm generated one predicate. User E used `links` [3] to connect to the Apache Web server. `links` generates files with random names, so each invocation of `links` has a different output set. We consider this noise in the output set because output sets are always different.

4.7.3 Solution ranking results

We took the solutions found by the precondition heuristic from all successful traces and used the solution-ranking heuristic to rank them. Almost all predicates generated by the base algorithm can be used to test the specific configuration problem, so most solutions were correct ones.

4.7.3.1 Solutions ranked for CVS problems

Table 4.9 shows the solution ranking results for CVS problem 1. Four solutions were generated, and all of them were correct solutions, initializing the CVS repository with a `cvs` command. The solution ranking heuristic ranked the `cvs init` as cvsroot

Solution	Number of traces
`vi /etc/group` as root	3
`usermod -G cvsgroup cvsuser` as root	1

CVS problem 2 is caused by an authorized user, cvsuser, not in the CVS group, and the correct solution is to add cvsuser into the CVS group. Each solution is ranked by the number of traces that apply the solution.

Table 4.10: Solutions ranked for CVS problem 2

Solution	Number of traces
commands with the same state delta a	2
`chmod 755 /home/testuser` as root	
`chmod 777 testuser` as root	1
`chmod o+rx /home/testuser` as root	1
`chmod -R 777 testuser/` as testuser	1
`vim /etc/httpd/conf/httpd.conf` as root	1

This problem is caused by the Apache Web process not having enough permission to search for /home/testuser, and the solution is to give /home/directory search permission (execution bit) for others. Each solution is ranked by the number of traces that applies the solution.

Table 4.11: Solutions ranked for Apache problem 1

higher than as root because more users applied it to fix the problem.

Table 4.10 shows the solution ranking results for CVS problem 2. The correct solution, adding cvsuser to CVS group, can be done by either editing the /etc/group file or using `usermod` to modify a user's group information. These two solutions have different output sets as `vi` writes to a log file to record the user's actions. The solution ranking heuristic ranked the editing group file higher than `usermod` because it occurred in more traces that fixed the problem.

4.7.3.2 Solutions ranked for Apache problems

Table 4.11 shows the solutions ranked by the solution ranking heuristic for Apache problem 1. The problem is that the Apache process does not have search permission

Solution	Number of traces
commands with the same state delta as	6
`chmod 755 /var/www/cgi-bin/test.pl` as root	
`chmod 777 /var/www/cgi-bin/test.pl` as root	1
`chmod +r test.pl` as root	1
`vi /var/www/cgi-bin/test.pl` as root	1

This problem is caused by the Apache Web process not having read permission for the CGI script, and the solution is to give read permission to the CGI script. Each solution is ranked by the number of traces that uses the solution.

Table 4.12: Solutions ranked for Apache problem 2

(the execution bit) for the /home/testuser directory, and the solution is to give that directory global search permission. The solution `vim /etc/httpd/conf/httpd.conf` that solved a user-introduced error in the configuration file was ranked the lowest as only that user fixes his own configuration error with this command.

Table 4.12 shows the solution ranking results for Apache problem 2. This configuration problem was due to the Apache process not having read permission for the CGI script, test.pl, and the solution is to give read permission to that file. Most users fixed the configuration problem by giving read permission, and this solution is also ranked the highest.

4.8 Discussion

When designing solutions to automatically generate predicates, our main goal is to minimize user involvement and generate high-quality predicates. We examined three alternative approaches before we decided on our final one. These approaches may produce more accurate predicates but would require more user-effort.

4.8.1 Alternative approaches to infer predicates

The first approach we considered was to have the user explicitly turn on and off recording when she started and finished testing the system. This required users only to keep a mental note of turning the recording on and off. If the user forgot to turn on or off the recording, she had to examine the user trace to determine where the recording tool should have been turned off.

The second approach we considered was to have AutoBash observation mode record all user interactions with the system and write them to a file. The user then would be given the file and be required to identify which commands were used as predicates. This approach could have recorded the most complete information as we would not have to worry the user forgetting to turn recording on and off. It might have been the most accurate as the user knew best what she had used for testing. However, we used this approach in our user study and found that this approach is very tedious for the user. Users often do not remember exactly what they were doing when presented with a static display of commands and screen output.

The last approach we considered was to ask user-specified rules to determine when commands were to be used as predicates. For example, a rule can specify which executable is used only as predicates. The main problem for this approach was that writing correct rules are time-consuming and difficult. Also, not all executable had a distinct role of either predicate or solution.

4.8.2 Using other output features for inferring predicates

In our approach, we used three output features (exit code, error message displayed on screen, and output set) to determine if two executions of a command should indicate different testing results. We chose these three output features because they are easy to collect and have shown to be effective. Some other potential useful output features include writing to an error log or applying more sophisticated techniques to understand screen output. Using output to an error log can be an effective output feature as many applications only writing to an error log when an error occurs. Tan

et al. [75] applied natural language processing to understand when source code and comments are inconsistent. In our approach, we only used standard error messages but could apply natural language processing to analyze screen output and understand application-specific error messages. As the purpose of error messages is to give users a hint about what problem an application occurs, it may be possible to write a template that can model error messages.

4.9 Summary

Predicates play an important role for automated configuration management tools, such as AutoBash and Chronus. However, writing predicates by hand is tedious, time-consuming, and requires expert knowledge. This work solves the problem of manually writing predicates by automatically inferring predicates and solutions from traces of users fixing configuration problems.

Our heuristics find repeated commands with different output features to identify predicates and used output features to determine if a predicate should return true or false. To evaluate the effectiveness of our heuristics, we conducted a user study and collected traces from real users fixing configuration problems. In the evaluation, we ran our heuristics to automatically generate predicates and solutions from the traces collected in the user study. Our heuristics can generate correct predicates for four out of ten CVS traces and for more than half of the Apache traces.

CHAPTER 5

Related Work

This thesis describes two research problems. The first problem we addressed is running demanding applications on mobile computing devices. To the best of our knowledge, our solution, Slingshot, is the first system to dynamically instantiate replicas of stateful applications in order to improve the performance of small, resource-poor mobile computers. Our work draws upon several areas of prior work, including virtual machine, and process migration, cyber foraging, fault-tolerant computing, and remote execution.

The second problem we addressed is how to make managing software configurations easier. We first built AutoBash, a set of tools that automate many parts of managing configurations. To the best of our knowledge, AutoBash is the first project to leverage operating-system-level causality tracking and speculative execution to improve configuration management.

AutoBash depends on predicates to test system state and the current approach is to rely on expert users to hand craft predicates that can test specific problems. Manually writing predicates is tedious, time-consuming, and not scalable with the number of applications. This thesis addresses this problem by automatically inferring predicates and solutions from user traces.

5.1 Slingshot: related work

Cyber foraging, the opportunistic use of surrogates to augment the capabilities of mobile computers, has been proposed [10] to solve the same usability problem of running demanding applications on mobile computing devices we discussed in this thesis. Slingshot is an instance of cyber foraging. Previous work in Spectra [34] examined how a cyber foraging system could locate the best server and application partitioning to use given dynamic resource constraints. In contrast, Slingshot takes this selection as a given and provides a mechanism for utilizing surrogate resources. Balan [13] and Goyal [39] have also proposed cyber foraging infrastructure. Compared to these systems, the major capability added by Slingshot is the ability to execute stateful services on surrogate computers. Data staging [35] and fluid replication [49] use surrogates to improve the performance of distributed file systems. They share common goals with Slingshot such as minimization of latency and ease of management—however, Slingshot applies these principles to computation rather than storage.

Slingshot's replication strategy is a form of primary-backup fault tolerance in that the replica on the home server allows the system to tolerate a fail-stop failure (halt on failure) of any number of second-class replicas. Our approach is most reminiscent of Hypervisor [19], which used deterministic replay to provide fault tolerance between virtual machines. In contrast to systems such as Hypervisor and ReVirt [33] which enforce determinism at the instruction level, Slingshot enforces determinism at the application level. This choice was driven by our desire to use a robust commercial virtual machine (VMware) without modification. Our approach to enforcing determinism was inspired by Rodrigues' BASE [71], which provides Byzantine fault tolerance by wrapping non-deterministic software with a layer that enforces deterministic behavior. A similar approach was used in Brown's operator undo [20].

After Chen and Noble [24] first suggested that virtual machine migration could be an effective mechanism for process migration, several research groups have built working prototypes. Our research focus is not on the migration mechanism itself, but rather on how it can be used to service the needs of small, mobile clients. We

use the Fauxide and Vulpes components from Intel's Internet Suspend/Resume [51] to intercept disk I/O requests made by virtual machines. The difference between ISR and Slingshot is that ISR executes a user's computing environment on a single terminal at a time. In contrast, Slingshot decomposes a user's environment into distinct services and replicates services on multiple computers. Slingshot hides the perceived latency of migration and surrogate failures, while letting a user execute applications anywhere a wireless connection exists.

Sapuntzakis [72] uses virtual machine migration, but focuses on users who compute at fixed locations, rather than the mobile users that Slingshot targets. Slingshot uses several of the optimizations suggested by Sapuntzakis, including ballooning and content-addressable storage. These optimizations have also been suggested by Waldspurger [79] and Tolia [78], respectively.

The applications we have investigated so far have been easy to partition because they were designed for client-server computing. Potentially, Slingshot could use one of several methods that automatically partition applications. For instance, Coign [45] partitions DCOM applications into client and server components. Globus [36], Condor [17], and Legion [40] dynamically place functionality, but target grid rather than mobile computing. Balan [12] has also provided a solution to quickly retarget applications to run on cyber foraging infrastructures. Chong *et al.* [26] proposed a way to automatically divide Web applications into client and server parts that can both secure sensitive data and minimize network communication.

Baratto's MobiDesk [15] is similar to our VNC application in that it virtualizes a remote desktop for mobile clients. However, MobiDesk migrates its desktop service between well-connected machines in a cluster in order to minimize downtime during server maintenance or upgrades. Slingshot uses replication rather than migration, and utilizes the computational resources of surrogates located at wireless hotspots. A MobiDesk-like cluster could serve as the ideal home server for Slingshot applications. Conversely, although Baratto shows considerable improvement over VNC in remote display performance, his results indicate that network latency still degrades interactive performance. Thus, surrogates could improve MobiDesk performance for mobile

clients.

5.2 AutoBash: related work

Chronus [82] also looked at the use of checkpoint and rollback for configuration management. Chronus uses a virtual machine monitor to implement rollback at the granularity of the entire computer system. The VM implementation allows Chronus to diagnose kernel bugs, but would also make it much harder to extract the causal information used by AutoBash to guide its search. Like AutoBash, Chronus uses user-defined predicates to test the behavior of the system. Chronus attacks a more limited problem: finding the point in time where a previously-working system ceased to operate correctly. AutoBash tries more generally to allow one system to learn from others by speculatively applying and testing fixes that have worked elsewhere for similar problems. Chronus shares our use of rollback to eliminate predicate effects, as does the work of Joshi *et al.* [47] in the IntroVirt project.

We have used Speculator [62] to implement checkpoint and rollback. Rx [66] and Pulse [53] apply operating system speculation to different domains: transparent recovery from non-deterministic application failure and deadlock detection, respectively.

Brown and Patterson's Operator Undo [21] uses a form of checkpoint and rollback to allow administrators to fix mail configuration errors. However, their approach requires application modification, whereas AutoBash functionality requires no application-level support.

PeerPressure [80] and its predecessor, Strider [81], share the same overarching goal as AutoBash: making configuration management easier. However, AutoBash takes a fundamentally different approach than PeerPressure or Strider; AutoBash reasons about *actions* rather than *state*. AutoBash therefore uses causal tracking and analysis to understand how user actions affect predicates that test system correctness, while Strider and PeerPressure apply statistical methods to reason about similarities between configuration state on different machines. The AutoBash approach works

well in environments such as Linux where configuration state may be hard to find because it is scattered throughout the file system, rather than coalesced in a central registry. Like AutoBash, Strider and PeerPressure observe causality to determine the system state that causally affects buggy applications. However, unlike AutoBash, these tools do not follow causal links across processes. So, if one process observes state and causally affects another process that exhibits a bug, PeerPressure cannot trace the appropriate causal chain back to the misconfigured state. Ultimately, the problem of configuration management seems complex enough that it may be best to combine multiple approaches such as AutoBash causal analysis and PeerPressure state analysis.

Many prior systems reason about causal interactions. For instance, King's BackTracker [50] traces causal interactions to determine what state has been changed during an intrusion. Aguilera *et al.* [6] use causal tracing of RPCs to debug performance problems. Causeway [23] allows applications to inject metadata that follows causal paths for distributed applications. PASS [59] uses causality to annotate files with provenance that describes their causal inputs: our input sets try to capture similar information, but limit the scope of information collected to specific periods of time.

Clarify [41] improves error reporting by monitoring software execution to generate a behavior profile when an error occurs. It then applies a classifier to match the bug profile to erroneous execution reports previously submitted by other users. One can regard Clarify as focusing on causality within process execution, whereas AutoBash monitors causal interactions external to a process. AutoBash may benefit from using similar classification and machine learning techniques to those employed by Clarify.

AutoBash's replay mode currently uses a simple approach, Hamming distance calculated over a vector of predicate results, to determine which solutions to try first. Other approaches, drawing from machine learning and information retrieval, could potentially do a better job of identifying bugs and their corresponding solutions. For example, Cohen *et al.* [29] showed that an approach that uses statistical methods to capture relationships between low-level performance metrics and high-level behaviors

outperformed an approach that used only the low-level metrics in tasks such as root-cause analysis and behavioral clustering. Attariyan and Flinn [9] aggregates the input sets for standard predicates to rank solutions and showed that correct solutions were ranked highest even when predicates were not written to test specific configuration problems.

5.3 Inferring predicates and solutions from user traces: related work

Nagaraja *et al.* [60] observed that operator mistakes are an important factor in the unavailability of online services and conducted experiments with human operators. Their experiments asked human operators to perform maintenance tasks and troubleshoot misconfigurations for their three-tiered Internet service. Their experiments are similar to our user study but since they used a three-tiered Internet service and we focus on configuration management for desktops and stand-alone servers we did not re-use their traces.

The software testing research community faces the same problem — manually writing test cases for software testing is tedious and time-consuming. Test automation, including automatic test generation and test oracles, is most related to this work. Automatic test case generation often requires a specification [18, 27, 31, 42, 54, 64] for the program being tested. Unit test cases can be generated by symbolic execution combined with concrete execution [74]. We did not use this technique to automatically generate predicates because writing a specification requires detailed knowledge of the program, and our solution does not require one. Many types of specification-based tests can generate test cases that have a thorough coverage but our approach can only infer predicates that a user executed. Our approach can improve its coverage by observing many users solving different configuration problems for the same software program.

The second part of test automation is a *test oracle* [69] that determines whether

a system behaves correctly during test execution. Most specification-based testing requires the specification to also describe what the expected output of a test case. Researchers also have shown approaches to automate test oracles for reactive systems from specifications [69, 68] and for GUI applications using formal models [55]. Our work did not require specifications or formal models but assumed that the last execution of a predicate is the expected result and used that as our oracle.

Strider [81] compares the registry of a misconfigured machine to a prior healthy snapshot to find bad registry entries and queries a database of known misconfiguration for the correct content. To eliminate the step of manually identifing a healthy state, PeerPressure [80] compares the registry of a misconfigured machine with those from a set of sample machines and applies statistical methods to rank a set of suspects. Both systems find solutions by using the content of a reference computer (a past healthy state or the most popular value from the sample machines) while we find solutions using causal dependency information and rank them by popularity among successful traces.

Snitch [56] collected configuration traces using an always-on tracing environment and used an *outcome marker* to explicitly identify faulty application runs. They used exit codes as the outcome marker, while we used exit codes, error messages on the screen, and output sets to identify if two executions of a repeated command exhibit distinctive execution behavior.

CHAPTER 6

Conclusion

This chapter concludes this thesis and includes the contributions this thesis makes as well as a summary.

6.1 Contributions

This thesis makes contributions in two major areas. The first is its conceptual contributions which include the novel ideas generated by this work. The second area is its two artifacts: Slingshot and AutoBash.

6.1.1 Conceptual contributions

After the idea of cyber foraging was proposed by the mobile computing community, researchers have worked on improving different aspects of cyber foraging. The first key novel idea of this thesis is exploring replication-based cyber foraging. Replicating applications has the advantage of improving both application response time and reliability when surrogate computers fail.

OS-level speculative execution has been used to improve performance for distributed systems and security checks. This thesis is the first to explore using OS-level speculative execution to automate managing configurations. Causal information tracking has been applied to intrusion detection, deadlock detection, and performance debugging. This thesis is the first to explore causal information between solutions

and predicates to improve one key component of configuration management.

Automatically generating test cases to automate software testing has been an active research problem in the software engineering community. Instead of having developers write specifications or models, this thesis explores automatically generating predicates from user input. This thesis used traces from users fixing configuration problems and applied heuristics to infer predicates.

6.1.2 Artifacts

To validate this thesis, I have developed two artifacts: Slingshot and AutoBash. Slingshot is the first cyber-foraging infrastructure that replicates applications on multiple computers to improve response time and data safety. Slingshot also uses virtual machines to encapsulate application-specific state to make replicating application to different physical machines easy. To manage replicas on surrogate computers and the home machines, Slingshot uses the checkpoint-and-replay technique often seen in the fault tolerance literature.

AutoBash is the first automated configuration management tools to use OS-level speculative execution to find solutions for configuration problems. AutoBash's observation mode is a modified `bash` shell that automatically runs predicates to test system state after each command and provides rollback capability. AutoBash's replay mode is a tool that speculatively executes solutions until it finds a solution that all predicates would return true. The replay mode also tracks causal information between predicates and solutions to reduce the time for regression testing.

6.2 Closing remarks

This thesis addresses two research problems. The first problem is how to run demanding applications on mobile computers. To solve this problem, we designed and built Slingshot, a replicated cyber foraging infrastructure. Slingshot replicates demanding applications of a mobile computer to run on both nearby surrogate computers and a home computer that the user owns; the surrogate computer can provide

swift response time and the home computer can be a safe repository.

The second problem this thesis examined is how to make configuration management for software systems easier. We designed and built AutoBash that makes it easier to manage software systems. AutoBash leverages OS-level speculative execution to automatically test system state and find solutions to fix configuration problems. AutoBash also uses causality tracking to improve the time to find a solution. We also extend AutoBash by developing heuristics that can infer predicates from user fixing configuration problems.

APPENDIX

APPENDIX A

User study document

A.1 Informed consent

Study name Inferring Software Configuration Predicates from User Traces

Contact person The primary investigator for this project – to whom questions and problems should be directed – is Ya-Yunn Su, email: yysu@umich.edu. The faculty advisor is Jason Flinn, email: jflinn@umich.edu

Description of research A common problem facing a computer user is troubleshooting software misconfigurations. Previously, we built a tool that can automatically search for solutions for misconfigurations if the user provides us with potential solutions and a set of predicates that can check if the misconfiguration is resolved. In this study, we would like to record users fixing configuration problems and apply different algorithms to the traces after wards to infer what predicates the user applied.

Description of human subject involvement You will be asked to solve six configuration problems and all your keyboard strokes will be recorded. We ask you to complete a short background survey at the beginning and end of the study. Also, we would like to interview you at the end of the study to understand your experiences in troubleshooting misconfigurations.

Length of human subject participation We ask that you come in to our laboratory in the Computer Science and Engineering building on North campus. We estimate that the total time commitment for this study is 1–2 hours.

Risks There are no foreseeable physical, psychological or social risks to you with this study. Since our study will be conducted on our laboratory machine, no personal information on your computer is disclosed. We use an existing commercial product to record your interactions (keystrokes) on our machine but you can turn the recording off at any time.

Expected benefits to subjects or to others Although you may not receive direct benefit from your participation, others may ultimately benefit from the knowledge obtained in this study.

Costs to subject resulting from participation in the study There are no monetary costs to you.

Payments to subject for participation in the study You will be compensated with a $20 Amazon.com gift certificate for participation in this study. You will receive $10 Amazon.com gift certificate if you stay at least one hour but do not successfully complete the study. Payment amounts for other situations will be at the sole discretion of the Principal Investigator. You will need to fill out a payment form to receive the payment.

Confidentiality You will not be identified in any reports on this study. Records will be kept confidential to the extent provided by federal, state, and local law. However, the Institutional Review Board, the sponsor of the study (i.e. NIH, FDA, etc.), or university and government officials responsible for monitoring this study may inspect these records.

Contact information Should you have any questions regarding this study, please contact Ya-Yunn Su at yysu@umich.edu or (734)763-5107 or Jason Flinn at jflinn@umich.edu or (734)936-5983.

IRB contact information Should you have questions regarding your rights as a research participant, please contact the Institutional Review Board, 540 E. Liberty Street, Suite 202, Ann Arbor, MI 48104-2210, (734) 936-0933, email: irbhsbs@umich.edu.

Your participation in this project is voluntary. Even after you sign the informed

consent document, you may decide to leave the study at any time without penalty or loss of benefits to which you may otherwise be entitled.

You may skip or refuse to answer any survey question without affecting your study compensation or academic standing/record.

One copy of this document will be kept together with the research records of this study. Also, you will be given a copy to keep.

I have read of the information given above. The investigator has offered to answer any questions I may have concerning the study. I hereby consent to participate in the study.

Printed Name Consenting signature

Audio recording of subjects

As part of this study, we wish to interview you and ask questions about your experience and would like to audiotape this interview. Please sign below if you are willing to have this interview recorded. You may still participate in this study if you are not willing to have the interview recorded.

Signature Date

A.2 User study hand out

A.2.1 General Instructions

You are new to an IT group in a small software company and are asked to assist your colleague to troubleshoot two misconfigurations for two applications: a version control system and a Web server. For each application, there will be more detailed instructions on what the requirements for that application are. You will troubleshoot

each misconfiguration in a separate virtual machine, and each virtual machine will not have network access. A laptop computer will be available to you to look up information on the Internet.

Here is the procedure you will take to troubleshoot a misconfiguration.

1. Before you start troubleshooting a misconfiguration for an application, read the instructions associated with that application.

2. The investigator will start `VMware Workstation` in console mode.

3. Start troubleshooting:

   - You will be given 15 minutes for each troubleshooting task. This time limit is to ensure the progress of this study.
   - Enter all commands into the virtual machine the investigator just started.
   - Think aloud during each task. For example, state what you are looking for, what you are testing, etc. The investigator will play a passive observer role, transcribe what you say, and also record audio to ensure the completeness of the information captured.

4. Enter `exit` when you complete the troubleshooting task.

5. Open the file `/test/tmp/commands` which contains all the commands you typed and their screen output. Specify the category (more details about category below) to which each command belongs.

A.2.2 Command Categories

We divide commands in the troubleshooting process into three categories: the first category is "Testing (T) if the configuration problem is solved." Commands in this category are any commands the user applies to test if the system is working. Examples of testing commands are pinging localhost, checking if a user has the right file permission to a file, etc. Based on the results of testing, the user determines the

root cause of the misconfiguration and takes action to remove the root cause. These actions are in the second category: "Fixing (F) the configuration problem." These are the commands that change the computer state, such as editing a configuration file or setting a environment variable, that can remove the root cause. The last category is "Others (O)," which include commands that cannot be easily categorized into "Testing or Fixing." Commands in this category can be commands that are typed wrong, or looking up manual pages, etc.

A.2.3 Pre-usage Survey

A.2.3.1 Familiarity with CVS version control

How would you describe your familiarity with the CVS version control system. Please select all that apply.

- I have basic knowledge of CVS version control system. ☐ Yes ☐ No.
- I use CVS regularly. ☐ Yes ☐ No.
- If the answer to the previous question is "Yes," please answer how often you use CVS.

  ☐ Once a week ☐ Once a month ☐ Once every three months ☐ Other. Please specify:

  _____

  _____

  _____

- I have set up a CVS repository before. ☐ Yes ☐ No.
- If the answer to the previous question is "Yes," please specify the number of times you have set up a CVS repository in the past year. _____
- I have troubleshot CVS repository configuration problems before. ☐ Yes ☐ No.

- If the answer to the previous question is "Yes," please specify the details of the configuration problems that you solved.

  _____

  _____

  _____

A.2.3.2 Familiarity with Apache Web server configuration

How would you describe your familiarity with the Apache Web server. Please select all that apply.

- I have set up an Apache Web server before. ☐ Yes ☐ No.
- If your answer to the previous question is "Yes," please specify what features you have set up.

  The Apache Web server I set up can ☐ display a user's homepages. ☐ execute CGI scripts. ☐ handle secure communication. ☐ handle PHP scripts. ☐ handle virtual hosts. ☐ Other, please specify

  _____

  _____

  _____

- I have troubleshot Apache Web server configuration problems before. ☐ Yes ☐ No.
- If your answer to the previous question is "Yes," please specify the details of the configuration problems that you solved.

  _____

  _____

  _____

- I understand how the Apache Web server executes CGI scripts. ☐ Yes ☐ No

A.2.3.3 General questions

What was the most challenging/difficult configuration task you have ever performed? The term "configuration task" includes but is not limited to OS and software installation, (re)configuration, troubleshooting misconfiguration, and upgrade.

Why do you consider that task challenging/difficult?

A.2.4 TASK: Troubleshooting CVS Version Control Misconfigurations

A.2.4.1 Instruction

You just join an IT department in a small software company and your manager asks the IT department to set up a source version control system to manage source code.

Your manager does not know much about setting up and configuring a version control system but knows what he would like the system to achieve. He would like the version control system to be able to:

- Allow each software developer to check in and out projects he or other users in the group have previously checked in.
- Restrict file access of the repository to only users in the development team.

Your colleague decided to use CVS version control system and has done some work in setting up a repository. He gives you the task of testing the CVS repository *locally* (that is, on the same machine and not via a network) and troubleshooting any misconfigurations you come along.

Here is the work your colleague has done

1. Read the CVS user manual.
2. Created a new user `cvsroot`, which has `cvsgroup` as its login group and `/home/cvsroot` as its home directory.
3. Initialized `/home/cvsroot` to be the CVS repository by executing `cvs -d /home/cvsroot init`.
4. Obtained a list of software developers who can have access to the CVS repository (`cvsuser` and yourself, `testuser`) and a list of other users who should not have access (`noncvsuser`).
5. Created a test directory at `/home/testuser/test_project` and a file `/home/testuser/test_project/testfile`.

Your colleague has *not* imported any project (or modules) into the CVS repository, therefore, there are no projects (or modules) in the CVS repository.

A.2.4.2 Cheat Sheet

Here is some useful information for you to finish your task.

- The CVS user manual is located at `http://ximbiot.com/cvs/manual/stable`. The "Overview," "The Repository," and "Starting a project with CVS" sections are especially useful.
- The CVS command manual is located at `http://ximbiot.com/cvs/manual/stable`'s Section "Guide to CVS commands" and "Quick reference to CVS commands."

A.2.4.3 Useful information

Since you are limited to console mode, here is some useful information.

- You are logged in as `testuser` (password is testuser) and the password for `root` account is also testuser.
- You may start by creating test cases that demonstrate the configuration problem and use them to guide your troubleshooting process.
- Read the requirements for each application as your goal is to troubleshoot each application so that it meets the requirements.
- Use `su` to switch between two different users. For example, type `su` and enter the root password (testuser) to become root. Type `exit` to become `testuser` again. Since you do not have other users' passwords, you can first become root and become that user.
- Use `chmod` to change directory or file permissions.
- Use `chown` to change the owner of a file or directory.

A.2.4.4 TASK: Training Session for CVS

In this training session, you are given a sample CVS version control system misconfiguration to troubleshoot and will become familiarized with basic CVS commands. Remember that your goal is to make sure the CVS repository is configured in a way that meets your manager's requirements:

- Allow each software developer to check in and out projects he or other users in the group have previously checked in.
- Restrict file access of the repository to only users in the development team.

The configuration problem here is that the CVS repository is not initialized: your colleague did not execute `cvs -d /home/cvsroot init`.

1. The investigator starts a virtual machine.
2. Start troubleshooting:

- Think aloud during each task. For example, state what you are looking for, what you are testing, etc.
- Enter all commands into the virtual machine the investigator just started.
- Enter `exit` when you complete the troubleshooting task.

3. After you finish troubleshooting this configuration problem, open the file `/test/tmp/commands`, which numbers all commands you have typed and their screen output. For each command, specify the category of that command:

   - F: Fixing the problem.
   - UF: Undoing a prior command used for fixing.
   - T: Testing if the problem is solved. Also specify 'S' if the screen output indicates the testing succeeds and 'F' if the screen output indicates that the testing fails.
   - UT: Undoing a prior command used for testing.
   - O: Others, any command that you cannot easily categorize into any of the above.

A.2.4.5 TASK: Troubleshooting Session for CVS

Now we will begin the real troubleshooting sessions for CVS. You will follow the same steps as in the training session and have the same goal as in the training session. However, the investigator will *not* tell you what the misconfiguration is and there will be *only one* configuration problem for each virtual machine.

Configuration Problem 1

1. The investigator starts a virtual machine.

2. Start troubleshooting:

   - Think aloud during each task. For example, state what you are looking for, what you are testing, etc.

- Enter all commands into the virtual machine the investigator just started.
- Enter `exit` when you complete the troubleshooting task.

3. After you finish troubleshooting this configuration problem, open the file `/test/tmp/commands`, which numbers all commands you have typed and their screen output. For each command, specify the category of that command:
   - F: Fixing the problem.
   - UF: Undoing a prior command used for fixing.
   - T: Testing if the problem is solved. Also specify 'S' if the screen output indicates the testing succeeds and 'F' if the screen output indicates that the testing fails.
   - UT: Undoing a prior command used for testing.
   - O: Others, any command that you cannot easily categorize into any of the above.

Configuration Problem 2

1. The investigator starts a virtual machine.
2. Start troubleshooting:
   - Think aloud during each task. For example, state what you are looking for, what you are testing, etc.
   - Enter all commands into the virtual machine the investigator just started.
   - Enter `exit` when you complete the troubleshooting task.
3. After you finish troubleshooting this configuration problem, open the file `/test/tmp/commands`, which numbers all commands you have typed and their screen output. For each command, specify the category of that command:
   - F: Fixing the problem.
   - UF: Undoing a prior command used for fixing.

- T: Testing if the problem is solved. Also specify 'S' if the screen output indicates the testing succeeds and 'F' if the screen output indicates that the testing fails.
- UT: Undoing a prior command used for testing.
- O: Others, any command that you cannot easily categorize into any of the above.

A.2.5 TASK: Troubleshooting Apache Web Server Misconfigurations

You are new to an IT department in a small software company and your manager asked the IT group to set up a Web server for the company.

Your manager does not know much about setting up or configuring a Web server but knows what he would like the Web server to achieve. He would like the Web server to be able to:

- Display the company Web page.
- Display each user's home page.
- Generate dynamic content for the company.

Your colleague decided to choose the Apache Web server for the company's Web server. He breaks the Apache Web server setup into three stages and has done some work in each stage. He told you to test the Apache Web server *locally* and troubleshoot any configuration problems you come along.

Stage 1

Set up the Apache Web server to display the company Web page.

Your colleague planned to use the Apache Web server's default page as "the company's Web page" for testing. Here is the work your colleague has done.

1. Downloaded the Apache Web server version 2.0.50 rpm package from `http://www.apache.org`.

2. Installed the rpm package using Linux package management tool.

3. Tested by starting the Apache Web server and retrieving the Apache Web server default Web page from `http://localhost`.

Stage 2 Configured the Apache Web server to display each user's home page.

Your colleague planned to use your (`testuser`) home page to test. Here is the work he has done to display each user's home page:

1. Referred to `http://httpd.apache.org/docs/2.0/urlmapping.html`.

2. Created a `/home/testuser/public_html` directory and an `index.html`.

Stage 3

Configured the Apache Web server to generate dynamic content.

Your colleague chose CGI (common gateway interface) to generate dynamic content and planned to use a `test.pl` Perl script to test. Here is the work that he has done.

1. Referred to `http://httpd.apache.org/docs/2.0/howto/cgi.html` on how to generate dynamic content with CGI.

2. Created a sample CGI script `test.pl` and put that script at `/var/www/cgi-bin`. The expected output of executing this CGI script is returning the environment variables.

3. Edited `httpd.conf` to specify the directory `/var/www/cgi-bin` contains only CGI scripts.

A.2.5.1 Cheat Sheet

Here is some useful information for you to finish your task.

- The Apache Web server manual is located at `http://httpd.apache.org/docs/2.0/`.

- Run `apachectl start` to start the Apache Web server.
  Run `apachectl stop` to cleanly shut down the Apache Web server.
- The Apache Web server's access and error logs are located at `/etc/httpd/logs/`.
- The Apache Web server's default configuration file `httpd.conf` is located at `/etc/httpd/conf/`
- The Apache Web server's default document root is located at `/var/www/htdocs`
- The Apache Web server's default CGI script directory is located at `/var/www/cgi-bin`

A.2.5.2 Useful information:

Since you are limited to console mode, here is some useful information.

- You are logged in as `testuser` (password is testuser) and the password for `root` account is also testuser.
- You may start by creating test cases that demonstrate the configuration problem and use them to guide your troubleshooting process.
- Read the requirements for each application as your goal is to troubleshoot each application so that it meets the requirements.
- Use `su` to switch between two different users. For example, type `su` and enter the root password (testuser) to become root. Type `exit` to become `testuser` again.
- Use `wget` to retrieve a Web page from a Web server.
- Use `chmod` to change directory or file permissions.
- Use `chown` to change the owner of a file or directory.

A.2.5.3 TASK: Training Session for Apache Web Server

In this training session, you are given a sample Apache Web server misconfiguration to troubleshoot and become familiarized with basic Apache Web server commands. The configuration problem here is that the default CGI directory (`/usr/local/apache2/cgi-bin`) not configured in the Apache Web server's configuration file and CGI scripts in that directory cannot be executed.

1. The investigator starts a virtual machine.
2. Start troubleshooting:
    - Think aloud during each task. For example, state what you are looking for, what you are testing, etc.
    - Enter all commands into the virtual machine the investigator just started.
    - Enter `exit` when you complete the troubleshooting task.
3. After you finish troubleshooting this configuration problem, open the file `/test/tmp/commands`, which numbers all commands you have typed and their screen output. For each command, specify the category of that command:
    - F: Fixing the problem.
    - UF: Undoing a prior command used for fixing.
    - T: Testing if the problem is solved. Also specify 'S' if the screen output indicates the testing succeeds and 'F' if the screen output indicates that the testing fails.
    - UT: Undoing a prior command used for testing.
    - O: Others, any command that you cannot easily categorize into any of the above.

A.2.5.4 TASK: Troubleshooting Session for Apache Web Server

Now we begin the real troubleshooting sessions for the Apache Web server. You will follow the same steps as in the training session but the investigator will *not* tell

you what the misconfiguration is. There will be *only one* configuration problem for each virtual machine.

Configuration Problem 1

1. The investigator starts a virtual machine.
2. Start troubleshooting:
   - Think aloud during each task. For example, state what you are looking for, what you are testing, etc.
   - Enter all commands into the virtual machine the investigator just started.
   - Enter `exit` when you complete the troubleshooting task.
3. After you finish troubleshooting this configuration problem, open the file `/test/tmp/commands`, which numbers all commands you have typed and their screen output. For each command, specify the category of that command:
   - F: Fixing the problem.
   - UF: Undoing a prior command used for fixing.
   - T: Testing if the problem is solved. Also specify 'S' if the screen output indicates the testing succeeds and 'F' if the screen output indicates that the testing fails.
   - UT: Undoing a prior command used for testing.
   - O: Others, any command that you cannot easily categorize into any of the above.

Configuration Problem 2

1. The investigator starts a virtual machine.
2. Start troubleshooting:
   - Think aloud during each task. For example, state what you are looking for, what you are testing, etc.

- Enter all commands into the virtual machine the investigator just started.
- Enter `exit` when you complete the troubleshooting task.

3. After you finish troubleshooting this configuration problem, open the file `/test/tmp/commands`, which numbers all commands you have typed and their screen output. For each command, specify the category of that command:
   - F: Fixing the problem.
   - UF: Undoing a prior command used for fixing.
   - T: Testing if the problem is solved. Also specify 'S' if the screen output indicates the testing succeeds and 'F' if the screen output indicates that the testing fails.
   - UT: Undoing a prior command used for testing.
   - O: Others, any command that you cannot easily categorize into any of the above.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] *CVS - Concurrent Versions System*. http://cvs.nongnu.org/.

[2] *GCC, the GNU Compiler Collection*. http://gcc.gnu.org/.

[3] *Links home page*. http://www.jikos.cz/~mikulas/links/.

[4] *RealVNC*. http://www.realvnc.com/.

[5] Microsoftw next-generation secure computing base - technical faq. Tech. rep., Microsoft Corporate, 2003.

[6] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 74–89.

[7] The Apache Software Foundation. *Apache HTTP Server Project*. http://httpd.apache.org/.

[8] Applian Technologies Inc. *Freecorder Toolbar - Free Sound Recorder*. http://applian.com/sound-recorder/.

[9] Attariyan, M., and Flinn, J. Using causality to diagnose configuration bugs. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX '08)* (Boston, MA, June 2008).

[10] Balan, R., Flinn, J., Satyanarayanan, M., Sinnamohideen, S., and Yang, H.-I. The case for cyber foraging. In *the 10th ACM SIGOPS European Workshop* (Saint-Emilion, France, September 2002).

[11] Balan, R. K. *Simplifying Cyber Foraging*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2006.

[12] Balan, R. K., Gergle, D., Satyanarayanan, M., and Herbsleb, J. Simplifying cyber foraging fro mobile devices. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services* (San Juan, Puerto Rico, June 2007), pp. 272–285.

[13] Balan, R. K., Satyanarayanan, M., Park, S., and Okoshi, T. Tactics-based remote execution for mobile computing. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services* (San Francisco, CA, May 2003), pp. 273–286.

[14] Barak, A., and Wheeler, R. Mosix: An integrated multiprocessor unix. In *Proceedings of the 1989 USENIX Annual Technical Conference* (San Diego, CA, January 1989), pp. 101–112.

[15] Baratto, R. A., Potter, S., Su, G., and Nieh, J. MobiDesk: Mobile virtual desktop computing. In *Proceedings of the 10th Annual Conference on Mobile Computing and Networking* (Philadelphia, PA, Sept/Oct 2004), pp. 1–16.

[16] Barham, P., Dragovic, B., Fraser, K., Hand, S., and Harris, T. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 164–177.

[17] Basney, J., and Livny, M. Improving goodput by co-scheduling CPU and network capacity. *International Journal of High Performance Computing Applications 13*, 3 (Fall 1999).

[18] Boyapati, C., Khurshid, S., and Marinov, D. Korat: Automated testing based on java predicates, 2002.

[19] Bressoud, T. C., and Schneider, F. B. Hypervisor-based fault-tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 1–11.

[20] Brown, A. B., and Patterson, D. A. Rewind, repair, replay: Three R's to dependability. In *the 10th ACM SIGOPS European Workshop* (St. Emilion, France, September 2002).

[21] Brown, A. B., and Patterson, D. A. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Technical Conference* (San Antonio, TX, June 2003).

[22] Carson, M. *Adaptation and Protocol Testing thorugh Network Emulation*. NIST, http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm.

[23] Chanda, A., Elmeleegy, K., Cox, A. L., and Zwaenepoel, W. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)* (Santa Fe, NM, June 2005).

[24] Chen, P., and Noble, B. When Virtual is Better Than Real. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems* (Schloss Elmau, Germany, May 2001).

[25] Cheriton, D. The v distributed system. *Communications of ACM 31*, 3 (March 1988), 314 – 333.

[26] Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L., and Zheng, X. Secure web applications via automatic partitioning. In *SOSP07* (Stevenson, WA, October 2007).

[27] Chow, T. S. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering 4*, 3 (1978), 178–187.

[28] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. In *The Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)* (Boston, MA, May 2005).

[29] Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 105–118.

[30] Cox, L. P., Murray, C. D., and Noble, B. D. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 285–298.

[31] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. Model-based testing in practice, May 1999.

[32] Douglis, F., and Ousterhout, J. Transparent process migration: Design alternatives and the sprite implementation. *Software Practice and Experience 21*, 7 (August 1991).

[33] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 211–224.

[34] Flinn, J., Narayanan, D., and Satyanarayanan, M. Self-tuned remote execution for pervasive computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Schloss Elmau, Germany, May 2001), pp. 61–66.

[35] Flinn, J., Sinnamohideen, S., Tolia, N., and Satyanarayanan, M. Data staging for untrusted surrogates. In *Proceedings of the 2nd USENIX Conference on File and Storage Technology* (San Francisco, CA, March/April 2003), pp. 15–28.

[36] Foster, I., Kesselman, C., Nick, J., and Tuecke, S. Grid services for distributed system integration. *Computer 35*, 6 (2002).

[37] Gambiroza, V., Sadeghi, B., and Knightly, E. W. End-to-end performance and fairness in multihop wireless backhaul networks. In *Proceedings of the 10th International Conference on Mobile Computing and Networking* (Philadelphia, PA, 2004), pp. 287–301.

[38] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 193–206.

[39] Goyal, S., and Carter, J. A lightweight secure cyber foraging infrastructure for resource-constrained devices. In *Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications* (Lake Windermere, England, December 2004).

[40] Grimshaw, A. S., and Wulf, W. A. Legion: Flexible support for wide-area computing. In *Proceedings of the 7th ACM SIGOPS European Workshop* (1996).

[41] Ha, J., Rossbach, C. J., Davis, J. V., Roy, I., Ramadan, H. E., Porter, D. E., Chen, D. L., and Witchel, E. Improved error reporting for software that uses black-box components. In *Proceedings of the Conference on Programming Language Design and Implementation 2007* (San Diego, CA, 2007).

[42] Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 3, 231–274.

[43] Holland, D. A., Josephson, W., Magoutis, K., Seltzer, M., Stein, C., and Lim, A. Research issues in no-futz computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Schloss Elmau, Germany, May 2001), pp. 106–110.

[44] Huang, Q., Wang, H. J., and Borisov, N. Privacy-preserving friends troubleshooting network. In *Proceedings of the 12th Network and Distributed System Security Symposium* (San Diego, CA, February 2005), pp. 245–257.

[45] Hunt, G. C., and Scott, M. L. The Coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (New Orleans, LA, February 1999), pp. 187–200.

[46] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., P.Hontalas, Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Weidel, J., Younger, H., and Bellenot, S. Time Warp operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, TX, November 1987), pp. 77–93.

[47] Joshi, A., King, S. T., Dunlap, G. W., and Chen, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 91–104.

[48] Kephart, J., and Chess, D. M. The vision of autonomic computing. *Computer 36*, 1 (2003), 45–52.

[49] Kim, M., Cox, L. P., and Noble, B. D. Safety, visibility, and performance in a wide-area file system. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (January 2002).

[50] King, S. T., and Chen, P. M. Backtracking intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 223–236.

[51] Kozuch, M., and Satyanarayanan, M. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications* (Callicoon, NY, June 2002).

[52] Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 121–136.

[53] Li, T., Ellis, C. S., Lebeck, A. R., and Sorin, D. J. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Annual Technical Conference* (31–44, April 2005).

[54] Marinov, D., and Khurshid, S. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)* (San Diego, CA, November 2001).

[55] Memon, A. M., Pollack, M. E., and Soffa, M. L. Automated test oracles for guis. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

[56] Mickens, J., Szummer, M., and Narayanan, D. Snitch: Interactive decision trees for troubleshooting misconfigurations. In *In Proceedings of the 2007 Workshop on Tackling Computer Systems Problems with Machine Learning Techniques* (Cambridge, MA, April 2007).

[57] Microsoft Corporation. *Universal Plug and Play Forum*, June 1999. http://www.upnp.org.

[58] Mummert, L., Ebling, M., and Satyanarayanan, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995).

[59] Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference* (Boston, MA, May/June 2006), pp. 43–56.

[60] Nagaraja, K., Oliveria, F., Bianchini, R., Martin, R., and Nguyen, T. Understanding and dealing with operator mistakes in internet services. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 61–76.

[61] Nath, P., Kozuch, M., O'Hallaron, D., Harkes, J., Satyanarayanan, M., Tolia, N., and Toups, M. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX '06)* (Boston, MA, June 2006).

[62] Nightingale, E. B., Chen, P. M., and Flinn, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.

[63] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 2006), pp. 1–14.

[64] Offutt, J., and Abdurazik, A. Generating tests from uml specifications. In *Second International Conference on the Unified Modeling Language (UML99)* (Ocrober 1999).

[65] Osman, S., Subhraveti, D., Su, G., and Nieh, J. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 361–376.

[66] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 235–248.

[67] Quinlan, S., and Dorward, S. Venti: a new approach to archival storag. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Monterey, CA, January 2002), pp. 89–102.

[68] Richardson, D. J. TAOS: Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)* (Seattle, WA), pp. 138–153.

[69] Richardson, D. J., Aha, S. L., and O'Malley, T. O. Specification-based test oracles for reactive systems.

[70] Robert Bradford, Evangelos Kotsovinos, A. F. H. S. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 2007 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (San Diego, CA, June 2007), pp. 169–179.

[71] Rodrigues, R., Castro, M., and Liskov, B. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001), pp. 15–28.

[72] Sapuntzakis, C. P., Chandra, R., Pfaff, B., Chow, J., Lam, M. S., and Rosenblum, M. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 377–390.

[73] Satyanarayanan, M. Pervasive computing: vision and challenges. *IEEE Personal Communications 8*, 4 (August 2001).

[74] Sen, K., Marinov, D., and Agha, G. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)* (2005), pp. 263–272.

[75] Tan, L., Yuan, D., Krishna, G., and Zhou, Y. /* iComment: Bugs or bad comments? */. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, October 2007).

[76] TCG TPM Specification Version 1.2 Revision 94. Tech. rep., Trusted Computing Group, March 2006.

[77] Tolia, N., Harkes, J., Kozuch, M., and Satyanarayanan, M. Integrating portable and distributed storage. In *Proceedings of the 3rd Annual USENIX Conference on File and Storage Technologies* (San Francisco, CA, March/April 2004).

[78] Tolia, N., Kozuch, M., Satyanarayanan, M., Karp, B., Bressoud, T., and Perrig, A. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference* (May 2003), pp. 127–140.

[79] Waldspurger, C. A. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 181–194.

[80] Wang, H. J., Platt, J. C., Chen, Y., Zhang, R., and Wang, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 245–257.

[81] Wang, Y.-M., Verbowski, C., Dunagan, J., Chen, Y., Wang, H. J., Yuan, C., and Zhang, Z. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of Usenix Large Installation Systems Administration Conference* (October 2003), pp. 159–172.

[82] Whitaker, A., Cox, R. S., and Gribble, S. D. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 77–90.

[83] Whitaker, A., Shaw, M., and Gribble, S. D. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 195–209.

[84] Wizzard Software Corporate. *IBM ViaVoice Speech Recognition SDK*. http://www.wizzardsoftware.com/ibm_viavoice_sdk.php.