Cabala: A Speculative Execution Framework to make Linux Services Fault Tolerant

by

Kavya Joshi

S.B., Computer Science and Engineering,
Massachusetts Institute of Technology, 2012

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 2013

Author: [illegible] ....
Department of Electrical Engineering and Computer Science
May 24, 2013

Certified by: ....
Prof. Robert Morris, Thesis Supervisor
May 24, 2013

Accepted by: [illegible] ....
Prof. Dennis M. Freeman,
Chairman, Masters of Engineering Thesis Committee

Cabala: A Speculative Execution Framework to make Linux Services Fault Tolerant

by

Kavya Joshi

Submitted to the Department of Electrical Engineering
and Computer Science
on May 24, 2013, in partial fulfillment of the
requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer
Science

Abstract

Cabala is a speculative execution framework that enables server programs in Linux to be fault tolerant. The errors it targets are run-time errors that are due to program bugs, and which cause program exit; in particular, it provides resilience to errors triggered by requests. The Cabala framework is implemented as a user-space library, and enforces per-request checkpoint/rollback semantics to provide fault-tolerance. It provides facilities for checkpointing the state of the server program just before request processing starts, executing the processing in an isolated environment, detecting tolerated run-time errors before they cause program exit, and committing or discarding the modifications made during processing. Cabala addresses the fault tolerance needs of server programs in the context of supported errors. It enables the services provided by a server program to be highly available and ensures that a program which is correct and satisfies Cabala's requirements remains correct despite tolerated errors; it also ensures that the system's state remains consistent. In addition, Cabala is easy to use; the only changes required to the server program's source code are the inclusion of the Cabala library and the addition of three library function calls. Cabala was evaluated with two Linux server programs, the Apache2 web server and the DHCP4 DHCP server; it detected the tolerated errors and correctly recovered the server programs in both cases.

Thesis Supervisor: Prof. Robert Morris
Title: Professor, Department of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Robert Morris, who was my thesis supervisor and sounding board too. I appreciate his guidance in this thesis, and especially value all the great discussions we've had.

I would also like to thank my support system: my mom and sister, Krittika; and my friends, in particular Alexandre Milouchev and Sanja Popovic who kept me going throughout.

Lastly, I'd like to dedicate this thesis and the journey it ends to the one person I cannot thank enough - my dad.

Contents

List of Tables

Chapter 1

Introduction

This thesis introduces a recovery scheme for server programs in Linux, targeted at run-time errors activated by requests. The Cabala library provides a speculative execution framework for server programs that enforces per-request checkpoint/rollback semantics to guarantee high service availability and data consistency.

1.1 Motivation and Proposed Solution

A server is a software application that provides a "service" to client applications [ ]. Web servers that deliver web content and mail servers that receive and deliver email messages are two routinely used examples of server programs among the many that provide different services. With the growing number of applications that depend on servers, the fault tolerance of server programs is an increasingly important concern [ ]; specifically, server programs must ensure that the service is available when needed by client applications and that it is correct, even in the face of errors.

Errors that affect a server program may result from bugs in the hardware, operating system software or program itself; an error may cause the server program to exit or hang, return responses in an untimely manner, or return incorrect responses - termed crash failure, timing failure and server response failure respectively. The subset of errors we provide tolerance to are errors that result from bugs in the server program and which cause program exit.

Non-trivial programs like server programs have bugs despite testing since not all bugs are uncovered during compilation and testing. These bugs, called latent bugs, are the result of logic flaws and programming errors. Since they are not transient, a simple program restart does not eliminate the errors. Replication techniques for fault tolerance do not provide resilience to them either since all copies of the program would contain the bugs. The standard approach to latent bugs is therefore to simply rectify the bug after it surfaces. This approach suffers from two shortcomings: first, the service provided by the server program is unavailable till the bug is fixed and second, the bug-fix rectifies the particular error-causing bug; other latent bugs in the program may be activated in the future and will require individual correction then. An additional drawback is that after the corrected server program is restarted, state recovery may be necessary depending on whether the server is stateless or stateful. A stateless server treats each request as an independent transaction which is unrelated to any previous request. Since information or state pertaining to a request is not retained across multiple requests, the server can simply be restarted if it exits in the middle of processing a request. On the other hand, a stateful server keeps in-memory state across requests and hence, the server's prior-crash state must be reconstructed on restart. Existing state recovery schemes are inefficient, further disadvantaging the standard approach. Our approach provides a less bug-specific and more efficient alternate to make server programs resilient to latent bugs. The problem of providing fault tolerance to latent bugs is compounded if they result in timing or server response failures when activated, since these failures are harder to detect than a crash failure. For this reason, we restrict the set of errors we detect and recover from to those due to program bugs and which would cause a server crash failure, specifically abnormal program termination. Examples of such errors are division-by-zero errors, null pointer dereferences and assertion failures. Our recovery scheme is most relevant to stateful servers.

Our scheme makes three fundamental assumptions. First, that request processing for a request is self-contained and atomic i.e. each request is processed independently of other requests and if the server fails midway through request processing for a request, all changes made so far during processing must be undone to ensure the system is in a consistent state. The per-request atomicity assumption is in line with the request processing requirements of protocols implemented by stateful servers like Lightweight Directory Access Protocol (LDAP) in LDAP servers [ ] and File Transfer Protocol (FTP) in FTP servers [ ], and is therefore reasonable. The second assumption is that a request triggers the program bug that causes server failure. The error-causing request may be malicious or simply malformed and its input may be invalid input that triggers a bug in the input-validation code, or valid input that triggers a bug elsewhere in the request processing code; in any case, a bad request is responsible for the error surfacing. The third assumption made is that tolerated errors can be detected by the operating system and are reported to the server program; this holds true for the errors in the set of errors we address.

Our recovery scheme is implemented as a speculative execution framework for server programs, provided as a user-space library called Cabala. The Cabala library enables a server program to process each request speculatively until an error occurs or processing completes successfully. In case of an error, Cabala restores the program and system state to that right before request processing started. If processing completes successfully, the state at the end of the speculative execution is committed to become the new program and system state. The start and end of request processing are defined by the server program; error detection and recovery are performed transparently by Cabala.

Cabala enforces speculative request processing for each request by creating an isolated environment right before processing starts. Request processing then proceeds in this isolated environment, which is a private copy of the server program's memory and the system state at the time of environment creation. If an error is reported during request processing, Cabala discards the isolated environment and restores the original program and system state, which are not modified by the processing. On the other hand, if request processing completes successfully, Cabala commits the isolated environment which contains all the state modifications made during processing; this operation replaces the program and system state that is visible to the rest of the system with that contained in the isolated environment. Our speculative execution framework thus provides per-request checkpoint/rollback semantics; the server program is checkpointed before the start of a request, and is effectively rolled-back to the checkpoint if request processing for the request fails. The failed request is not re-executed.

Cabala's design is closely related to that of two existing systems: libckpt and Speculator. Libckpt [ ] is a user-level checkpoint/rollback library that transparently checkpoints Unix applications to provide recovery from processor failure. Upon a processor failure, the user would restart the checkpointed application with a special flag to resume execution from the most recent checkpoint. Speculator [ ] is a Linux kernel extension that provides support for speculative process execution in order to improve the performance of distributed file systems. Speculator implements a lightweight checkpoint/rollback mechanism in the Linux kernel, provides isolation of speculative state from non-speculatively executing, or non-speculative, processes and tracks casual dependencies between related speculative processes. Like libckpt, Cabala is a user-space library. We chose to implement our recovery scheme in user-space rather than in the kernel for reasons of simplicity of implementation, the low overhead of inclusion in applications and to leverage application-specific knowledge. The trade-off is that, like in libckpt, checkpointing is not completely transparent to the application. A further similarity with libckpt is that the recovery scheme restores the program to its state at the last checkpoint, from which it then executes. The speculative execution model in Cabala is largely inspired by that in Speculator. Like Speculator, we implement checkpoint/rollback and guarantee speculative state is never exposed to non-speculative processes. Unlike Speculator though, we do not track speculative process dependencies; thus, in our system, a process may perform only a restricted set of operations while executing speculatively.

The implementation of Cabala's support for speculative execution was the main challenge in our undertaking; specifically, the isolated environment for speculative execution includes a shadow file system to which all file system operations are applied, and ensuring that all cases of all operations are handled correctly was tricky. The implementation of the other components, like checkpointing, recovery and commit, was more straightforward.

Cabala addresses the fault tolerance needs of server programs in the context of supported errors. It enables the services provided by a server program to be highly available and ensures that a program which is correct and satisfies Cabala's requirements remains correct despite tolerated errors; it also ensures that the system's state remains consistent. In addition, Cabala is easy to use; the only changes required to the server program's source code are the inclusion of the Cabala library and the addition of three library function calls. Cabala's predominant limitation is the server program's restricted operation during request processing, since processing occurs during speculative execution; the fault tolerance benefits provided remain valuable nonetheless.

1.2 Thesis Contributions

The primary contribution of this thesis is a mechanism for making server programs in Linux fault tolerant to the set of errors that result from program bugs, and which cause program exit; in particular, Cabala has the property of making server programs resilient to error-causing requests. The fault tolerance provided by Cabala is especially useful to stateful servers. We therefore evaluate it with a stateful server like the DHCP4 DHCP server, in addition to the stateless Apache2 web server; Cabala detects and correctly recovers from the tolerated errors in both cases.

Cabala is, to our knowledge, the only system that implements per-request checkpoint/rollback to provide fault tolerance; it is also the first to implement speculative execution to provide fault tolerance.

Cabala provides an easy to use mechanism for fault tolerance; to make a server program fault tolerant to the tolerated errors, only four additional lines must be added to its source code, which is a vastly simpler solution than most existing options for fault tolerance.

1.3 Outline

The rest of the thesis is organized as follows: Chapter 2 presents an overview of the related work. Chapter 3 describes the design of Cabala; Chapter 4 details its implementation. We present an evaluation of Cabala in Chapter 5, and Chapter 6 concludes.

Chapter 2

Related Work

This chapter presents related work and compares each to Cabala.

Cabala is a user-space library that provides a speculative execution framework to make server programs in Linux fault tolerant. Cabala provides tolerance to run-time errors that result from bugs in the server program, and which cause program exit; specifically errors that are triggered by requests. We first present related work in the field of software fault tolerance and then the related work in speculative execution.

2.1 Software Fault Tolerance

Cabala is closely related to libckpt [ ], a user-space library that implements checkpoint/rollback for Unix applications. Libckpt transparently checkpoints applications to provide recovery from processor failure; upon a processor failure, the user would restart the checkpointed application with a special flag to resume its execution from the most recent checkpoint. Unlike Cabala, libckpt does not detect any type of run-time error since it is intended for recovery from processor failure. Although it would be possible to use libckpt to recover from a run-time error that has been detected using a separate mechanism, recovery is not transparent to the application since it requires manually restarting the program. Furthermore, since libckpt assumes a transient fault model, the error-causing request would be re-executed and the server program would fail in the same way again.

The Chorus micro-kernel operating system [ ] also provides a checkpoint/rollback mechanism for applications. Chorus facilitates the persistent checkpointing of programs, and if a program fails, detects and restarts the failed program. Unlike libckpt, recovery in Chorus is transparent to the application but like libckpt, Chorus does not address the target fault model either.

An alternate strategy to checkpoint/rollback for fault tolerance is redundant execution. Systems like Tandem's NonStop operating system [ ] and the UNIX-based Targon/32 operating system [ ] employ a process-pair model to achieve tolerance to hardware failure like processor failure, and Heisenbugs which are transient software bugs. In the process-pair model, a server program is run as two processes on two different processors; one process, called the primary process, receives clients' requests and executes them. The primary process in NonStop forwards requests to the back-up process as well, which executes them independently. In Targon/32, the primary process instead periodically sends a state snapshot to the back-up process, which updates its state correspondingly. In both versions of the process-pair model, the back-up process takes over as the new primary if the primary process or the processor it runs on fails. The back-up process first re-executes all client requests since the last message received from the primary, before it begins to accept client requests as the new primary. For this reason, neither NonStop or Targon/32 provide tolerance to the errors Cabala targets. After a server program's primary process fails, the back-up process would re-execute the error-causing request during its promotion to primary; the request triggers the same latent bug in the back-up, since the back-up runs the same program as the failed primary, and the back-up process would fail as well.

Primary site is another redundant-execution approach to fault tolerance; a modified version is implemented by Huang and Kintala in [ ], and used in several AT&T telecommunications network management products. In the primary site approach, the service to be made fault tolerant is replicated at many nodes or machines, one of which is designated as the primary site and the others as back-up sites. All client requests for the service are sent to the primary site, which executes them. The primary periodically checkpoints its state to the back-ups as well. The primary site approach provides fault tolerance to hardware errors, transient software (operating system, application) errors and power failure, since if the primary fails for any of these reasons, one of the back-ups becomes the new primary. In Huang and Kintala's system, primary failure due to either application failure or system failure (i.e. hardware or operating system failure) is detected and recovered from. Like the process-pair model, the primary site approach fails to address the target fault model since it replicates the server program and repeats the error-causing request, which results in server failure at the back-up as well.

Replicated State Machines (RSM) involve replicating the service on different machines as well. All client requests are sent to the primary, which chooses the order of processing and executes the requests locally, in that order. The primary forwards the requests to the back-ups or replicas, which also execute the requests locally, and respond to the primary. Bressoud and Schneider implement a hypervisor-based RSM scheme [ ] to provide application fault tolerance to processor failure. In their scheme, the primary runs in a virtual machine on one processor and the back-ups run in independent virtual machines on different processors. The hypervisors coordinate to ensure that all I/O inputs and interrupts are delivered to the back-ups at exactly the same execution points that the primary received them at. Primary processor failure is detected by the back-ups and after a set time period, one of the back-ups becomes the new primary. The hypervisor-based fault tolerance system is able to detect termination of the server program's primary process. In the interim before a back-up takes over as the new primary, the back-up does not re-execute instructions; it skips them. Thus, the hypervisor-based system provides fault tolerance to the errors Cabala targets. The VMware vSphere Fault Tolerance line employs an improved version of the same scheme, which drastically reduces the performance overhead. However, the key disadvantage compared to Cabala is still the degree of server slow-down caused.

Horus [ ] is a toolkit that removes the complexity of implementing replication techniques for fault tolerance from application development. The Horus subsystem, which can run in either user-space or the kernel, implements a reliable group communication protocol and atomic broadcast for applications. It thus enables applications to leverage replication for fault tolerance. Although Horus makes it easier for server programs to become fault tolerant to hardware failure and transient software failure, it does not help them become tolerant to our target fault model. The server program is replicated at the back-ups which execute all requests, including the error-causing request; thus, the back-ups would fail in the same way as the primary.

Multi-version techniques for fault tolerance involve running different versions of the program or blocks of the program to be made tolerant so as to reduce the likelihood of the same program bug being triggered in the primary and back-ups. In N-version programming, N functionally equivalent yet independently developed versions of the program are run on N different machines, and the final output to be returned is determined by supervisor software. The DEDIX distributed supervisor and testbed system [ ] is a UNIX-variant framework for executing N-version applications. The implementation overhead of providing fault tolerance through N-version programming is extremely high, although the target fault model would be addressed.

In the recovery blocks method [ ], a critical code block is coupled with an acceptance test and alternate implementations of the block to form a "recovery block". If the acceptance test determines that the result produced by a code block is incorrect, the block is replaced by one of its alternates and re-executed. Recovery blocks are intended for logical program errors that can be arbitrated by an acceptance test; errors that cause server program exit would not be detected.

There are other system-specific fault tolerance techniques. The IBM MVS operating system [ ], for example, implements a multi-level recovery scheme which attempts a pre-specified hierarchy of recovery routines for a faulty program, before its abnormal termination occurs. The IBM MVS OS would address our target fault model for server fault tolerance; however, there is a high implementation overhead since the recovery routines would have to be implemented as part of the server program.

2.2 Speculative Execution

Speculative execution has previously been successfully applied in processor and compiler design [ ]. The goal of these systems however, is to improve performance rather than provide fault tolerance.

The speculative execution model in Cabala is largely inspired by Speculator [ ], a Linux kernel extension that provides support for speculative process execution, in order to improve the performance of distributed file systems. Speculator implements a lightweight checkpoint/rollback mechanism in the Linux kernel. It also ensures that speculative state is never externalized or directly observed by non-speculative processes, and allows multiple processes to share speculative state by tracking causal dependencies propagated through inter-process communication. Speculator improves distributed file system performance by masking remote-I/O latency; the file system predicts the result of a remote operation, checkpoints the state of the calling process using Speculator and continues its execution speculatively i.e. based on the predicted result. If the prediction is correct, the checkpoint is discarded; if not, the calling process is restored to the checkpoint by Speculator, and the operation is retried.

Speculator uses speculative execution for value prediction in file system operations and does not aim to provide fault tolerance.

Chapter 3

Design

This chapter describes the design of Cabala. We first present an overview of the design and the fault model it handles. We then define the requirements imposed by the design, before describing the design in detail. We conclude with a discussion of its limitations.

3.1 Overview

The Cabala library implements a speculative execution framework for request processing in server programs in order to make the programs fault tolerant. The framework provides facilities for checkpointing the state of the server program just before request processing starts, executing the processing in an isolated environment, catching tolerated run-time errors before they cause program exit, and committing or discarding the modifications made during processing. The design of the library can be explained in terms of these four functions, which are introduced here and developed in the later sections of the chapter:

Checkpointing the state of the server program involves saving a snapshot of the program's memory and storing program state information such as the memory map, the program break, the current working directory and the file descriptors; the checkpointed state is the state the server is rolled-back to if an error occurs during request processing. Cabala implements Copy-On-Write (COW) checkpointing for the program's memory, an optimization well-known in the literature [ ] and detailed in section 3.4.1.

The speculative execution of the server program begins after checkpointing is complete; it ends at the successful completion of request processing or on encountering a tolerated error. During the period of speculative execution, none of the changes made by the server program should be externalized to non-speculative processes. To that end, Cabala creates an isolated environment right before request processing starts. The speculative processing then proceeds in the isolated environment, which is a private copy of the server program's memory and the system state at the time of environment creation. In particular, a private shadow file system for the server program is created, to which all its file system operations are applied; this is explained in section 3.4.3. In addition, system calls and library function calls that are not idempotent are either intercepted or disallowed, as explained in section 3.4.2

To detect tolerated errors during request processing, Cabala installs signal handlers for the respective signals. The handlers are also responsible for discarding speculative state i.e. the shadow file system and modified memory regions, and restoring the program to the state it was in just before request processing started i.e. the checkpointed state; this recovery procedure is described in section 3.4.4. After the program's state has been restored, it resumes execution from that state; it does not re-execute the error-causing request.

If request processing completes with no errors, the changes made to the program's memory and the shadow file system are committed i.e. made visible to the rest of the system; the commit process is described in section 3.4.5

3.2 Fault Model

Cabala's design provides fault tolerance to a specific set of of errors, namely, run-time errors that are due to server program bugs triggered by requests, and which cause server program exit. More precisely, the errors Cabala handles are errors that:

- Result from bugs in the server program:

The only errors Cabala provides resilience to are errors that are due to bugs in the server program. Cabala does not address hardware errors such as disk or processor failure, memory (RAM/ECC) errors, errors due to CPU overheating, or power failure. It also does not address errors due to operating system bugs.

Providing fault tolerance to this set of errors rather than those due to hardware or operating system bugs is valuable for three reasons: first, most failures are due to faulty software rather than hardware [ ]. Second, operating system bugs are less likely than other software bugs since operating system code is typically the most extensively tested and third, existing fault tolerance techniques do not address these errors sufficiently (see section 2.1)

- Cause immediate program exit:

  The only errors Cabala provides resilience to are those that cause immediate program exit since these errors are easy to detect. Cabala does not address errors that cause the server to return responses in an untimely manner or return incorrect responses. It also does not address errors that cause the program to crash, i.e. exit or hang, after a time delay.

- Are triggered by requests:

  Cabala handles errors that are due to bugs triggered by requests; the error-causing request may be malicious or simply malformed and its input may be invalid input that triggers a bug in the input-validation code, or valid input that triggers a bug elsewhere in the request processing code. Cabala handles request-triggered errors by design, since after the server program has been restored, it does not re-execute the error-causing request.

- Can be detected in user-space:

  Cabala only handles errors that are reported to the user-space program since error detection in Cabala's design occurs via error signals delivered to the signal handlers it installs.

  The set of errors that raise processor exceptions which are delivered to the

Table 3.1: Tolerated Errors in x86-Linux

Program Error	**Signal Generated**
Memory access error: null pointer dereference, array out-of-bounds	SIGSEGV
Arithmetic error: division-by-zero, overflow	SIGFPE
Illegal operation error: unknown/privileged instruction	SIGILL
Bad system call: bad argument to syscall	SIGSYS
Bus error: incorrect memory access alignment, non-existent physical address	SIGBUS
Assertion failure	SIGABRT

offending program as signals for handling is dependent on instruction set architecture (ISA) and operating system. The set of tolerated errors and the corresponding signals they generate in the x86-Linux architecture we use for our implementation of Cabala's design is listed in Table 3.1; examples are: divide-by-zero errors, null pointer dereferences and array out-of-bounds errors. Assertion failures raise processor exceptions that are delivered to the program irrespective of system architecture; thus, these are supported in our implementation too.

3.3 Requirements

Cabala's design imposes two requirements on the server program: first, the program must be structured as a single-threaded event-loop that processes requests; thus, Cabala does not support multi-threaded servers. Second, the program must declare the start and end of request processing; this is required because Cabala's design identifies a request as the unit of checkpoint/commit i.e. the recovery scheme in Cabala implements all-or-nothing semantics for the modifications made by request processing, which is per-request. With respect to requests, Cabala's design requires each request to be a self-contained unit; this is because Cabala's recovery scheme only discards the modifications made by the failed request processing and does not roll-back any other request processing that is dependent on those modifications.

3.4 The Cabala Library

Each component of Cabala's design, introduced in the overview (section 3.1), is explained in detail in this section.

3.4.1 Memory checkpointing

Cabala uses a Copy-On-Write (COW) scheme to checkpoint program memory. COW delays the creation of a memory page copy until the copy is first needed i.e. until the first write access to a writable page; it is an optimization to basic checkpointing, which creates copies of all pages at each checkpoint.

There are two components to our COW scheme: the page fault handler installed by Cabala and the checkpoint function called right before request processing starts. The program's writable memory regions are made non-writable by the function; a subsequent write access by the program therefore triggers a permissions page fault exception and execution control is transferred to the page fault handler. At this point, the handler creates a copy of the page which is saved away; the original page is marked writable, control returns to the program and the write occurs successfully.

3.4.2 System calls and library functions

The Cabala library provides custom implementations of two classes of system calls and glibc functions: first, those that modify program or system state and second, those that result in the creation of process dependencies on the speculatively executing process. The custom implementations for the former class track all modifications made during speculative execution so that they may be undone in case of error. The custom implementations for the latter simply disallow the actions in our initial version of the framework; the purpose of disallowing them is to prevent speculative changes

Table 3.2: Intercepted Syscalls and Library Function Calls:
Functions in the right-most column are the disallowed functions

mmap	read	fork
munmap	write	vfork
mprotect	link	clone
brk	unlink	pipe
sbrk	rename	kill
open	mkdir	IPC functions: msgsend, msgget etc.
creat	rmdir	socket functions: send, recv etc.
close	chdir	execve functions: exec, execv etc.

from having consequences on non-speculative execution since the latter cannot be undone.

The complete list of syscalls and library function calls Cabala intercepts is given in Table 3.2; examples of functions intercepted to track modifications are memory management functions such as mmap/munmap and sbrk, and file system functions such as open/close, read/write and link/unlink. Functions that are disallowed include functions for inter-process communication and network communication, and functions like fork and clone which create a child process.

To ensure our custom syscall/library function is invoked rather than the original, we use a standard Linux technique called function interposition.

3.4.3 Shadow file system

Cabala creates a private, per-request shadow file system for the server program, right before request processing starts; all file system modifications during processing are only applied to the shadow FS. Cabala uses an approach similar to COW to create this file system; the shadow FS is built up incrementally and recursively on non read-only (i.e. write or read/write) accesses to files, as explained below:

For each request, a file system rooted at "/var/tmp/[pid]" - henceforth called fakeroot, is created; initially, the fakeroot is a symbolic link to root - "/". On the first non read-only access to a file, all the directories in the pathname of the file are created under fakeroot. So for example, on a write-access to the file "/Users/kavya/thesis.tex", the following *directories* would be created: fakeroot, "fakeroot/Users" and "fakeroot/Users/kavya". All the other contents of the non-terminal directories would be created as symlinks to the original contents since they are not in the pathname. Similarly, a *copy* of the original file is created in the shadow file system as "fakeroot/Users/kavya/thesis.tex", whereas the other contents of the directory would merely be symlinks as well; this copy in the shadow FS is the file that is written to.

The purpose of populating the directory pseudo-copies created under fakeroot with symlinks for the contents not in the pathname is to provide a simple and correct implementation for directory reads (including the Linux command, ls), despite renames and deletions.

Cabala interposes on file system functions like open/close, read/write, link/unlink, rename, mkdir/rmdir etc. to create the shadow FS and route file system operations to it.

3.4.4 Recovery

Cabala installs signal handlers for the signals generated by the occurrence of tolerated errors. The handlers perform recovery. The goal of recovery is to undo all modifications made to program and system state during the request processing which caused the error. To achieve this, Cabala discards the shadow FS, which contains the file system modifications, and restores the program's memory to its checkpointed state, which effectively discards the modifications to memory; the program's memory is restored from the saved memory map and COW pages copies.

3.4.5 Commit process

At the successful completion of request processing for a request, Cabala commits the modifications made to the program and system state by the processing; this operation makes the modifications visible to future request processing and the rest of the system.

Due to Cabala's COW scheme, all modifications to the program's memory can be made during request processing and undone in case of error except memory region unmaps, which only occur at commit. Cabala's shadow file system is effectively copied over to the original file system to commit the file system modifications made during request processing.

3.5 Design Discussion

Cabala is designed to address the fault tolerance needs of server programs in the context of supported errors.

Cabala enables the services provided by a server program to be highly available since it detects an error as soon as it occurs and before the server program exits, by installing special signal handlers. Furthermore, our recovery procedure is efficient since we do not need to undo any operations other than memory allocations; we can simply discard the shadow file system and restore memory to its checkpointed state.

Cabala ensures that a program which is correct and satisfies Cabala's requirements remains correct despite tolerated errors; it also ensures that the system's state remains consistent. First, it checkpoints the server program's state and effectively the file system right before request processing starts. The checkpointed program and system state is error-free with respect to tolerated errors; our fault model does not include errors that cause state to be logically incorrect. If an error occurs during request processing, the program and system state are restored to the unmodified checkpointed state, which is known to be correct. Second, Cabala creates an isolated environment for the server program's speculative request processing to ensure that speculative state, which may be discarded, is not observed by other processes. Cabala also restricts how the program may interact with the rest of the system while executing speculatively. Specifically, the program is not allowed to perform any action that would cause other processes to be dependent on it; the purpose is to prevent speculative changes from having consequences on non-speculative execution since the latter cannot be undone, which would result in inconsistencies and incorrectness. For example, if during speculative request processing, the server sends a message to a non-speculative process which causes that process to take some action, although it is easy to restore the server to its pre-request state on error, we cannot undo the message-sending and receiving. Furthermore, it would be extremely difficult to track the chain of events set in motion by the message (like the action taken by the receiver) and undo them; but failing to do so would be incorrect since the server believes it has never sent the message. Thus, inter-process communication (IPC) including network communication, and functions that create a child process (like fork and clone) are disallowed by Cabala.

Lastly, Cabala is simple to implement and has a low implementation overhead for use in server programs since the library is implemented in user-space rather than in the kernel. To use Cabala, the only changes required to a server program's source code are the inclusion of the Cabala library and the addition of three library function calls.

3.6 Challenges and Limitations

3.6.1 Challenges

Conceiving the shadow file system was the main challenge in designing Cabala. Specifically, ensuring that all cases of all file system operations and sequences of operations are handled correctly by the design was tricky; for instance, Cabala maintains symlinks to files that have been not been opened for writing in order to correctly support directory reads after file deletions and renames.

3.6.2 Limitations

Our current design suffers from three key limitations:

1. Cabala's design does not allow a program that is executing speculatively to make syscalls or library function calls that would cause other processes to be

dependent on it. Thus, a server program may perform only a restricted set of operations during request processing.

2. Checkpointing is not completely transparent to the application since Cabala is a user-space library and the application is required to call the necessary Cabala functions for checkpoint/commit. In addition, as a user-space library, Cabala cannot support the execve family of system calls (in addition to the function calls disallowed to prevent process dependencies,) since they replace the program image which includes the Cabala library.

3. Shadow copies of files that are written to by the program during speculative execution replace the original files at the end of speculative execution i.e. at the commit; this overwrites all the changes made to the file by other programs during the period of speculative execution after the creation of the shadow copy. Although this was not an issue during our testing, since the server programs we tested Cabala with don't write to files that other programs write to as well (for example, the only files dhcpd writes to are its dhcpd.leases file and its log file), it could pose a problem in other cases.

Chapter 4

Implementation

This chapter details the implementation of the Cabala library. The library is implemented in C and consists of 2300 lines of code. Cabala currently supports Linux-x86 server applications and has been tested with the Apache2 web server, the Bind9 DNS name server and the DHCP4 DHCP server.

4.1 Overview

The implementation logic is split into four components: the checkpointing component which checkpoints the state of the server program before request processing starts (section 4.2), the shadow file system which is responsible for file system management during speculative execution (section 4.3), the recovery subsystem which executes recovery at the detection of tolerated errors (section 4.4) and the commit routine which runs at the successful completion of request processing (section 4.5). In addition, we re-implement the functions listed in Table 3.2 to provide the speculative execution semantics required by our design.

The initialization routine of the Cabala library allocates a pre-determined amount of memory for Cabala's data structures. It also installs the signal handlers required for recovery and performs function interposition of the syscalls and library functions we re-implement. cabala_init is the first Cabala library function call a server program makes, so as to set-up the framework for fault tolerance.

4.2 Checkpointing

Checkpointing the state of the server program involves saving a snapshot of the program's memory and storing program state information such as the memory map, the program break, the current working directory and the file descriptors. The server program is checkpointed by calling the cabala_checkpoint function right before request processing for a request starts.

4.2.1 Checkpointing program state information

Cabala stores the program's memory map and file descriptors by copying the /proc file system files that contain this information to certain locations; the /proc/[pid]/maps file is copied to /var/tmp/mapscopy/[pid] and similarly, the /proc/[pid]/fd and /proc/[pid]/fdinfo files are copied to /var/tmp/fdcopy/[pid] and /var/tmp/fdinfocopy/[pid] respectively. The current program break is obtained by calling glibc's sbrk; the value is stored in a global variable.

4.2.2 Memory checkpointing

Cabala implements the COW scheme described in section 3.4.1 to save snapshots of the program's memory regions.

The data structures implemented for the scheme are:

The vma_info struct : maps a memory region to the address of its snapshot.

The modified_vmas list : tracks the writable memory regions modified by the program during speculative execution.

The prot_vmas list : tracks the memory regions that have been mprotect-ed by the program during speculative execution.

The new_vmas list : tracks the memory regions that have been mmap-ed by the program during speculative execution.

The unmapped_vmas list : tracks the memory regions that have been unmapped (with the munmap function) by the program during speculative execution.

Cabala's checkpoint_memory function implements the mechanism to track writes to memory. The function parses the /proc/[pid]/maps file (where pid is the server program's pid) to obtain the address range and access permissions of memory regions currently mapped by the server program, and those that are writable are mprotect-ed to be non-writable while preserving their read and execute permissions.

The Cabala library installs a custom page fault handler in the cabala_init function to implement COW. In Linux, a page fault exception generates a SIGSEGV signal which is delivered to the offending program's SIGSEGV signal handler if one is installed. Thus, the user-level handler installed by Cabala is invoked at an illegal memory access. The handler first checks if the illegal memory access was a write to a COW address i.e. an address that is valid (mapped), and which was write-protected by the checkpoint_memory function. If this is the case, it creates a snapshot or read-only copy of the enclosing page, creates a vma_info struct that maps the memory range to the corresponding snapshot address, and appends the struct to the modified_vmas list. The original page is mprotect-ed to be writable and control returns to the program; the write to the original page then occurs successfully.

The memory allocated for Cabala's data structures must not be write-protected since they are used by the page fault handler. To that end, the cabala_init function allocates the memory needed by the Cabala library as a contiguous region of a pre-determined size and saves the address of the region allocated in a global variable; the checkpoint_memory function does not write-protect the memory region at the address in the variable. The function does not write-protect memory that constitutes the program's stack as well.

4.3 Shadow File System

Cabala creates a private shadow file system as part of the isolated environment created for the speculative execution of the server program. The per-request shadow FS is created right before request processing for a request starts, and all file system modifications during processing are only applied to the shadow FS.

The terminology used in this section is defined as follows: an original file is a file in the real file system and a shadow file is a file in the shadow file system.

4.3.1 Data structures

The following data structures are implemented to aid in file system operations during speculative execution:

The file_info struct maps a file to a list of shadow files. The file is identified by its device and inode number - henceforth termed the file id, and each shadow file is identified by its pathname.

The fd_pair struct maps an fd to a shadow fd. The fd is an fd opened before speculative execution; the shadow fd is an fd opened during speculative execution.

The inodes_copied list tracks original files that have been recreated in the shadow FS and shadow files that are created as a result of file creation operations during speculative execution.

The unlinked_files list tracks file pathnames unlinked by the program during speculative execution.

The mapped_fds list tracks pre-speculative execution fds that have been accessed during speculative execution.

4.3.2 File system operations

The shadow file system is built up incrementally and recursively on non read-only (i.e. write or read/write) accesses to files, as explained in section 3.4.3; Cabala interposes on the open function to implement this.

Cabala's implementation of open checks if a shadow file for the requested pathname exists; if it does, a file descriptor (fd) for the shadow file is returned. If not, it checks if the original file with the pathname exists and has not been deleted during this unit of speculative execution. If so, a corresponding copy is created in the shadow file system and an fd for the newly created shadow file is returned; if not, depending on the access flags argument passed to open, an error is thrown or a new file is created in the shadow file system and an fd for it is returned. In both the cases a shadow file is created, a file_info struct is appended to the copied_inodes list. If the shadow file is created as a copy of an original file, the struct maps the original file's file id to the shadow file's pathname; if the shadow file is created as a new file, the struct maps the shadow file's file id to the shadow file's pathname instead.

The creation of the shadow file in either case proceeds in a top-down, recursive check-and-create manner i.e. all directories in the pathname that do not have counterparts in the shadow file system are created first. Depending on the access flags, the shadow directories and the file may be created as symlinks to the original versions - as in the case of a read-only access, or they may be created as independent copies.

It is worth noting that the look-up we implement to determine whether a file has a counterpart in the shadow file system first invokes the stat syscall with the pathname the shadow file would have if it existed. If that fails, the look-up obtains the file id of the original file (by invoking stat with the pathname) and checks if the copied_inodes list has an entry for the file id; if it does, the entry contains the pathname of the file's corresponding shadow file. The look-up is implemented as such to ensure that the hard links to the original file that existed before the creation of the shadow FS map to the same shadow file. This is required to provide correct file system semantics with respect to hard links; for example, writes to a linked file appear in the other linked files as well.

File system functions that operate on file descriptors returned by open - like read, write and close, are re-implemented. In Cabala's implementation, they first check if the fd is one that was returned by an open call made during speculative execution. If this is the case, the standard file system function is invoked since Cabala's open function is guaranteed to have returned the right fd; an fd for the shadow file rather than the original was returned if the current access or any prior access to the file was a non read-only access, otherwise an fd to the original file was returned. If the fd was opened before speculative execution began, Cabala cannot perform the operation on the fd since it would be visible to the rest of the system. Therefore, close does not close the fd; it instead ensures the fd is a valid fd (using fcntl) and if so, returns success vacuously. Read and write check the mapped_fds list to obtain the corresponding shadow fd, if it exists; if it does, the standard read/write function is invoked with the shadow fd as the argument. If not, the pathname of the file represented by the fd is obtained from the program's checkpointed file descriptors information (i.e. by parsing the checkpointed fd and fdinfo files), and the file is opened; since the open invoked is Cabala's open, a shadow fd is returned. The state of the shadow fd is updated to that of the fd's using lseek, and an fd_pair struct that maps the fd to the updated shadow fd is appended to the mapped_fds list. The standard read/write function is then invoked with the shadow fd as the argument.

The creation of hard links during speculative execution is supported by intercepting calls to the link function. First, the shadow file of the file referred to by oldpath (i.e. the file to link to) is created if it does not exist. Next, the shadow file system version of the newpath pathname (i.e. the link pathname) is linked to the shadow file. Finally, the shadow link pathname is appended to the list of pathnames in the shadow file's file_info struct in the copied_inodes list, to ensure that the new link is accounted for at commit/recovery.

Unlinking requires re-implementation as well. A pathname of the file to be unlinked is created and appended to the unlinked_files list. If the original file exists, it is not unlinked; if it does not, i.e. the pathname was a new link/new file created during speculative execution, it is unlinked. The unlinked_files list is used by the open function to ensure that attempts to open a pathname that was unlinked during speculative execution will fail, and by the commit routine.

4.4 Error Detection and Recovery

4.4.1 Error detection

To detect the tolerated errors, Cabala installs signal handlers for the respective signals (see Table 3.1).

4.4.2 Recovery

The goal of recovery in Cabala is to undo all modifications made to program and system state by the request processing which caused the error.

The signal handlers installed by Cabala perform recovery:

1. The shadow file system is discarded. All file system modifications made during speculative execution are only contained in the shadow FS; thus, the file system is restored to its state before request processing for the error-causing request started.

2. The program's memory is restored to its checkpointed state to discard the modifications to memory. The saved memory map is parsed to obtain the original access permissions of the program's memory regions at the time of checkpoint. The non-writable regions are mprotect-ed with their original permissions to undo any mprotects or munmaps that may have occurred during speculative execution. For each writable region, the modified_vmas list is checked to see if it was modified during speculative execution; if so, the snapshot of the region, which was created by the COW scheme, is restored and the region is mprotect-ed with its original permissions. If not, the region is only mprotect-ed. To undo the memory allocations due to mmap, brk and sbrk calls during speculative execution, the memory regions tracked in the new_vmas list are munmap-ed.

After the program's state has been restored, it resumes execution from that state. Cabala uses sigsetjmp/siglongjmp to return execution control to the program with a special return value; the application does not re-execute the request on getting that return value.

4.5 Commit Routine

The commit routine is called by the server program after the successful completion of request processing for a request. The routine converts the speculative modifications made by the processing to non-speculative modifications i.e. at commit, the modifications are made visible to future request processing and the rest of the system.

The modifications made to the server program's memory during request processing are not all speculative; mmaps and mprotects in most cases directly manipulate the program's memory during request processing and their modifications are undone by recovery in case of error. Thus, the only modifications to memory that are deferred till commit are munmaps and mprotects in one case. To perform these modifications, the cabala_commit function first walks the modified_vmas list; for each memory region that does not have a snapshot, it checks the wrprotect bit. If the bit is not set, it mprotects the region with PROT_WRITE in addition to its original protections; none of the other memory regions in the list are mprotect-ed. cabala_commit then munmaps the regions tracked in the unmapped_vmas list.

The file system modifications during request processing are made to the shadow file system; the cabala_commit function applies them to the original file system in two steps: first, it walks the inodes_copied list; for each entry whose isnew flag is set, it creates a new file in the original file system, with the pathname corresponding to the shadow file's pathname, and copies the contents of the shadow file into the newly created file. The original files already exist for the other entries so the contents are copied over directly. In both cases, the pathnames in an entry's list of shadow files are the hard links to the file and therefore, the corresponding hard links to the original file are created as well. The cabala_commit function then walks the unlinked_files list and unlinks the pathnames (which are pathnames of original files) contained in it.

4.6 Challenges

The two main challenges in the implementation of Cabala were: handling the merging of memory regions in the /proc/[pid]/maps file and handling all cases of all file system operations correctly. The former complicates the implementation of the COW scheme and the other memory management functions, while the latter is tricky to get right.

4.7 Limitations

Our current implementation of Cabala suffers from the following limitations:

1. Checkpointed state (namely the program state information and the COW snapshots) and the Cabala library data structures are stored in memory rather than in persistent storage. We chose the in-memory approach for two reasons: first, simplicity of implementation and second, the slight performance benefit offered, which contributes to our goal of high service availability. The resulting limitations are that we do not tolerate faults that cause the loss of in-memory program state (for example, those that cause the program to be killed), and we cannot provide any fault tolerance guarantees if the memory itself is faulty.

2. We allocate a fixed amount of memory for the checkpointed state and the Cabala library data structures at the initialization of the framework. It is therefore possible to run out of space for these purposes; in our current implementation, we simply throw an error if that happens. The better design alternative is to implement our own memory allocator. We save this for future work in the interest of simplicity of the first version.

3. The Cabala library currently uses hard-coded values such as the program's root directory. Thus, our implementation does not support programs that have been chroot-ed before speculative execution begins; support for chroot-ed processes can be implemented using the /proc file system (specifically, /proc/[pid]/root) but we save this for future work.

Chapter 5

Evaluation

We tested Cabala with three Linux server programs to evaluate: one, whether it detects the tolerated errors and correctly recovers the server program; and two, whether it is easy to use for the server program. The methodology used in our evaluation and the results are presented in this chapter.

5.0.1 Methodology

We tested Cabala on an x86-Linux system, with the Apache2 web server (httpd), the Bind9 DNS name server (named), and the DHCP4 DHCP server (dhcpd); httpd is a stateless server, whereas named and dhcpd are both stateful servers. httpd is configured with the the mpm_prefork_module since Cabala does not support multi-threaded servers; for the same reason, named is configured with the –disable-threads option.

To test the Cabala library, we included the library in each server program's source-code, added calls to the cabala_init, cabala_checkpoint and cabala_commit functions in the appropriate locations, and rebuilt the servers. Each server program was run normally to test Cabala's checkpoint/commit facilities, and with a bug which was introduced to test Cabala's error detection and recovery facilities; we inserted a few lines of code to raise one of the tolerated errors in the request processing code-path.

5.0.2 Results

We evaluate Cabala on two criteria: one, whether it is able to detect the tolerated errors and recover the server program correctly; and two, whether it is easy to use. The tolerated errors are listed in Table 3.1

1. When the server program was run normally, without the error-causing lines, Cabala's checkpoint/commit worked correctly for both httpd and dhcpd. With the introduced bug as well, Cabala detected the error and recovered the program correctly for both httpd and dhcpd. Neither experiment worked with named; we attribute the failure to the incorrect placement of the Cabala function calls in the named source-code, which is complex.

2. The Cabala library is easy to use if the server program's source-code is well-understood; this was true for our experiments with httpd and dhcpd. The named source-code however, is complex and we believe we may have inserted the calls in the wrong locations. It is worth noting that the right placement of the Cabala function calls in the server program's source-code is crucial for Cabala to provide the desired fault tolerance.

Chapter 6

Conclusion

Cabala is a speculative execution framework that enables server programs in Linux to be fault tolerant to run-time errors that result from bugs in the program, and which cause program exit; in particular, Cabala provides resilience to errors triggered by requests.

Cabala addresses the fault tolerance needs of server programs in the context of supported errors. It enables the services provided by a server program to be highly available and ensures program correctness and system-wide consistency despite errors. In addition, Cabala is easy to use.

Cabala was evaluated with three Linux server programs; it detected the tolerated errors and correctly recovered the server program in two cases, the Apache2 web server and the DHCP4 DHCP server. It did not succeed in the experiment with the Bind9 DNS name server; we believe this is due to the incorrect placement of the Cabala function calls in the server program's source-code.

6.1 Future Work

We would like to improve Cabala in two ways.

Cabala currently does not track speculative process dependencies. As a result, the server program may perform only a restricted set of operations during request processing, since processing occurs during speculative execution. One way to track speculative process dependencies is to use the approach in Speculator, although this would involve making modifications to the Linux kernel.

Cabala currently does not implement a memory allocator. As a result, only a predetermined amount of memory can be allocated for the checkpointed state and the Cabala library data structures, and the allocation must occur at the initialization of the framework. It would be fairly straightforward to implement a standard user-space memory allocator for Cabala, which would prevent the risks of running out of space and having a large memory footprint.

Bibliography

[1] Y. Huang and C. Kintala, "Software implemented fault tolerance: Technologies and experience," in *FTCS*, vol. 23. IEEE COMPUTER SOCIETY PRESS, 1993, pp. 2–9.

[2] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[3] T. Howes, M. Smith, and G. S. Good, *Understanding and deploying LDAP directory services.* Addison-Wesley Professional, 2003.

[4] J. Postel and J. Reynolds, "File transfer protocol," 1985.

[5] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent checkpointing under unix.* University of Tennessee], Computer Science Department, 1994.

[6] E. B. Nightingale, P. M. Chen, and J. Flinn, "Speculative execution in a distributed file system," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 191–205.

[7] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard *et al.*, "Overview of the chorus distributed operating systems," in *Computing Systems.* Citeseer, 1991.

[8] J. F. Bartlett, "A nonstop kernel," in *ACM SIGOPS Operating Systems Review*, vol. 15, no. 5. ACM, 1981, pp. 22–29.

[9] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault tolerance under unix," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 1, pp. 1–24, 1989.

[10] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 80–107, 1996.

[11] R. Van Renesse, T. M. Hickey, and K. P. Birman, "Design and performance of horus: A lightweight group communications system," Cornell University, Tech. Rep., 1994.

[12] A. Avizienis, "The n-version approach to fault-tolerant software," *Software Engineering, IEEE Transactions on*, no. 12, pp. 1491–1501, 1985.

[13] B. Randell, “System structure for software fault tolerance,” *Software Engineering, IEEE Transactions on*, no. 2, pp. 220–232, 1975.

[14] S. Mourad and D. Andrews, “On the reliability of the ibm mvs/xa operating system,” *Software Engineering, IEEE Transactions on*, no. 10, pp. 1135–1139, 1987.

[15] F. Gabbay and A. Mendelson, *Speculative execution based on value prediction.* Citeseer, 1996.

[16] M. Baker and M. Sullivan, “The recovery box: Using fast recovery to provide high availability in the unix environment,” in *Proc. Summer 1992 USENIX Conference*, 1992, pp. 31–43.