AUTOMATIC RECOVERY FOR REQUEST ORIENTED SYSTEMS

BY

ANDREW DAVID LENHARTH

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor Vikram Adve, Chair
Professor Sarita Adve
Assistant Professor Samuel King
Professor Yuanyuan Zhou, University of California, San Diego

Abstract

Gracefully recovering from software and hardware faults is important to ensuring highly reliable and available systems. Operating systems have privileged access to all aspects of system operation, thus a fault related to them is able to affect the entire system. Existing approaches to operating system recovery either do not protect the entire system or require a completely new operating system design.

This dissertation presents a new approach to fault recovery in operating systems called *Recovery Domains*. This approach allows recovery from unanticipated faults in commodity operating systems. Recovery is organized around the concept of a dynamic request. Operating system entry points initiate requests to perform some action. System calls, for example, are a request by an application to the operating system. When a fault is detected, the recovery system rolls back the effects of the offending recovery domain while leaving the remainder of the system running. To ensure that the entire system (including the state of other concurrent kernel threads) remains consistent after the rollback, dependencies between domains are tracked as the system runs. When rolling back a faulting domain, any other domains that were dependent on the it, because of data-flow between the domains, are rolled back and restarted.

Recovery Domains do not make faults transparent. Request failures are reported to the requester. This visibility allows handling of faults which are

permanent: those faults which would reoccur if the request were retried. Recovery Domains also handle timing and transient faults.

Recovery Domains require compiler support to instrument the system. The necessary support is simple, but can cause unnecessarily large system overhead. This dissertation describes several performance improvements to Recovery Domains based on dynamic analysis of the system state and static analysis of memory regions, allocators, and locks. Runtime analysis of the inter-dependence of the active requests can allow reduced tracking of state changes. The recovery compiler can reason about memory regions and data structures protected by a lock to eliminate instrumentation on many operations to locked memory. "Fresh" heap objects, those objects which have been allocated and have not yet become visible to other requests and threads, require no instrumentation. These improvements to the recovery runtime and compiler provide substantial performance improvements over more simple implementations.

This dissertation describes the goals, approach, semantics, and programming model of Recovery Domains; the minimal implementation of the runtime and compiler; the static analysis and optimization at the compiler level and dynamic optimization to the runtime; and the porting of two significantly different versions of the Linux kernel to the recovery system. It evaluates the overhead, effectiveness, and coverage of recovery. Finally it describes the potential integration of a model fault detector with the Recovery Domains system.

To Vernita, for her love and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recovery and continued correct execution of software a system in the event of a fault has been sought for decades as a means to increase the reliability of these systems. As our dependence on software becomes more pervasive, software faults have a greater impact on our personal lives, our economies, our health, our safety, and our infrastructures. Reducing or nullifying the effects of failures is therefore quite desirable.

Operating systems form the heart of modern software stacks. In most areas, only a handful of commodity operating systems are in use. Desktop and server operating systems, for example, are commodities with estimates of the top three operating system families (Linux, Mac OS X, and Windows 2000 and later) covering around 96% of the market [98, 100]. Even in the specialized area of supercomputers, one commodity operating system, Linux, is found on 443 of the top 500 supercomputers [101]. The pervasiveness of a few operating systems gives common sets of bugs and vulnerabilities to many machines performing many varied functions and running many different applications. A fault that can be triggered in an operating system will affect a more diverse and larger set of users than a fault in a database or in presentation software.

Operating systems are also highly privileged, thus a breach in the OS becomes a breach in all applications running on the OS. Operating systems, due to their control of system resources and management of many security mea-

sures, have unfettered access to the state of anything running on them. This privileged state, while necessary, amplifies the effect and potential disruption of faults.

Operating systems provide a common and complex foundation for software systems. As such, they serve as a common failure point for a diverse range of systems. Fail they do [32]. Analysis of one mobile phone operating system shows that individual users experience an operating system failure every 11 days [19]. Operating system failures are blamed for problems ranging from the 2007 London Stock Exchange failure to regular crashes experienced by users [31].

The availability of computational power is steadily increasing, both in the low-end, consumer-system space, such as home routers, cell phones, and set-top boxes, as well as in the traditional desktop computing space. Some of this computational power can be used to improve the user's experience with the devices by improving reliability thus preventing frustrating crashes. Reliability is, in fact, already pushed by some vendors as a differentiating feature of their system (e.g. Apple, HP, Cisco, IBM, etc), showing a general market interest in more reliable systems.

Since operating systems are foundational to a system's stability and are integral to limiting the impact of faults in programs run on the system, the ability to recover from faults is especially important to creating and maintaining a reliable system.

The primary traditional approaches to recovering from faults in the operating system focus on drivers and, to a lesser extent, dynamically loaded extensions. Recovering from faults in drivers, such as in [78, 77, 95], is based on isolation, logging, and interposing on the communication between drivers and the rest of the kernel. Other systems, such as [69], use similar techniques

to protect the kernel against arbitrary dynamically loaded extensions.

Existing approaches do not address entire operating systems. Focusing on drivers and extensions leaves the majority of the code in an operating system unprotected from faults. In fact, errors in the core of commodity operating systems are especially difficult to recover from since such systems are inherently multi-threaded, handle resources shared between many clients (viz., user processes), have extensive asynchronous internal behavior, and directly interface to hardware.

This dissertation proposes an organizing principle for recovery called *Recovery Domains* that enables complex multi-threaded systems to recover from run-time faults in nearly all components. Recovery Domains have strong recovery semantics yet minimal and localized effects in the event that recovery is triggered. Recovery domains are easy for programmers to add to a system and require few changes to existing source code. The semantics are such that they can be easily analyzed and manipulated by a compiler or automated tool. Recovery Domains are a general mechanism capable of handling very demanding multi-threaded, state intensive, request-oriented software structures.

Recovery Domains do not require rewriting an operating system in a different style, such as as a microkernel, as some existing approaches do, such as [42, 68, 7, 6]. Nor do they only protect portions of the kernel, as in [95, 77, 69]. Further, they don't require a change in implementation language to one which is type-safe, as required by [42, 68]. Being able to protect an existing, entire kernel without significant changes in structure or language sets this work apart from exist approaches to operating system recovery.

As discussed in Chapter 6, Recovery Domains provide a high rate of fault recovery with only a couple of hundred lines of code changed in or added

to the original kernel. Recovery succeeded for an average of 33 consecutive injected random faults. Performance, with static and dynamic optimizations, is very good, often with less than 10% overhead. Low runtime overhead and low programmer overhead make Recovery Domains an attractive way to provide full-kernel fault recovery.

1.1 Research Contributions of this Dissertation

This work, broadly, contributes a *new approach to the recovery of request-oriented systems*. This approach is applied to commodity operating systems, potentially allowing significant improvements in reliability for real-world systems. The high-level contributions of this dissertation are:

- A new request-oriented recovery mechanism with simple, powerful, and easy to understand semantics.
- A simple to use interface for programmers to add recovery capabilities to their systems with little effort.
- A set of compiler transformations to introduce recovery into systems.
- An implementation of the recovery mechanism which supports the recovering of operating systems.
- An investigation of the effectiveness, overheads, and programmer complexity of introducing recovery into a commodity operating system, namely, Linux.
- The use of inter-procedural analysis of memory regions to reduce the overhead of recovery.

- The use of runtime analysis of the system to reduce the recovery overhead.

1.2 Dissertation Organization

Chapter 2 provides the motivation and high-level considerations in designing operating system recovery. Chapter 3 describes Recovery Domains, focusing on the fundamental approach, concepts, and structures. Chapter 4 describes a complete implementation of Recovery Domains, the experience of porting Linux 2.4.22. Chapter 5 describes the addition of several optimizations based on compiler analysis and runtime analysis and the experience porting Linux 2.6.27. Chapter 6 provides performance, survivability, and coverage results. Chapter 7 uses Secure Virtual Architecture's [21] memory safety tool as a model fault detector and shows how it would be modified to use Recovery Domains when a memory fault is detected. Chapter 8 places this work in the context of other recovery systems and recovery techniques. Finally, chapter 9 concludes the work.

Chapter 2

Operating System Recovery

Recovering from faults in operating systems presents many challenges. We discuss these before moving to enumerating the most desirable qualities of an ideal recovery system. To address those challenges and to meet those ideals, we present the unifying organization: requests.

2.1 Challenges of Operating System Recovery

Recovering from faults in operating systems has several unique challenges in addition to the challenges of recovering from faults in general software systems. The challenges of operating system recovery come from several sources:

- Their inherent multi-threaded nature;
- Their low-level operation;
- Their rich functionality and complexity; and
- The pragmatics of commodity system.

Modern operating systems are inherently multi-threaded. A recovery system for operating systems must be able to navigate the many threads potentially executing simultaneously on multiple processors and recover the system to a consistent state across all threads and processors.

Due to the low-level nature of the system which is being recovered, the recovery system itself must be self-standing; it must not rely on the operating system. Since the operating system must run on top of the recovery runtime, minimal dependencies should exist from the recovery system runtime to the operating system. Ideally the recovery system's data should not be corruptible by the OS. This can be achieved by having the OS running with such memory safety guarantees as would pro-actively prevent it from accessing the recovery system's runtime data. An example using of a memory fault detector which provides these guarantees is given in Chapter 7.

The operating system is also low-level due to it interacting with and controlling of hardware. Through these interactions, it interacts with the outside world. The recovery system must gracefully interact with the real world and other software. This primarily means that it should have a clear semantic on the effects of recovery on hardware device state.

A system to recover from operating-system-level faults needs to be able to handle the wide range of activities that an OS engages in on behalf of requests from its clients. An operating system has a large exposed surface available to malicious or buggy entities: the system call API, the network, and all hardware devices. Each of these, and others, has a distinct entry point into the kernel and interacts with kernel state in ways that are sometimes unique to their role. This large exposed surface comes from the many functions the operating system must perform, from processing network packets to interacting with block devices to handling timers to creating abstractions such as file-systems. This large surface imposes many paths into the kernel and many potential interactions between entities which a recovery system must be able to handle.

Operating systems, especially those in wide deployment, represent the results of significant engineering efforts. There seems to be little enthusiasm in the commercial realm to re-architect an existing operating system with either a new organization or in a new language which allows for easier recovery. Thus, a practical recovery system must fit within the common architectures for operating systems and make modest demands of the programmers with respect to changes in the operating system structure.

Code does not exist in a causally isolated bubble, nor is it common that all code running on a machine uses the recovery system. Hence the recovery system must deal with events, code, and operations that are outside the realm of recovery. It should allow, where external code is appropriately structured, the programmer to annotate external interfaces to allow proper restoration of a sane state to the external component.

Designing the semantics of recovery is in itself a major challenge. The semantics must be strong enough to allow automatic recovery; the semantics must require minimal design changes in the operating system; the semantics must be natural enough for programmers to easily comprehend; and the semantics must be possible to implement. These and other conditions are discussed in more detail in Chapter 2.2. The semantics dictate how threads interact, how requests interact, how requests are tracked across thread boundaries, how irreversible actions are dealt with, and in what state the system is left after recovery. Compelling levels of recovery require a recovery system which specifies in an easily understood way what the system will look like after a recovery. This allows the programmers to know how to have the system continue operating after recovery events. Semantics must be designed which can allow a programmer to succinctly specify recovery for an operating system kernel which is useful to the programmer and can handle the many corner

cases of complex, multi-threaded, legacy operating systems, while still allowing an implementation which can be understood. Recovery is not an event one wants to fail, so trust in the implementation is important; thus the complexity of semantics is at odds with the simplicity and, to some extent, the trustworthiness of an implementation.

Performance and correctness are also important and intertwined challenges of this work. The amount of performance degradation that is acceptable depends on how well the recovery mechanism works, how often the recovery system is employed, the size of the vulnerable surface of the target system, and the requirements for availability of the target system. That said, correctness is a must. The recovery system must not change the semantics of the program when it is operating fault-free. These two characteristics are intertwined. The recovery system must do work to enable a recovery to happen, and this work must happen during normal, fault-free execution. The challenge is to minimize the overhead incurred during fault-free execution to enable recovery. Care must be taken in the design and implementation of both the compiler transformation and the runtime to ensure correct and efficient execution and correct recovery. This includes not reordering memory accesses, respecting locks, and not deferring or batching recovery work if that would introduce a window of time during which recovery in another thread would cause undefined behavior.

2.2 The Ideal Recovery System

The ideal recovery system for recovery of faults in existing, common, deployed operating system has several enumerable properties. Among the most important are:

Automatic recovery: Recovery should be executed completely by the recovery system without intervention by the operating system, though some fault detector which triggers a recovery may be used.

For the entire system: Recovery should be possible from any part of the operating system, not just specific subsystems or drivers.

No porting effort: Use of the recovery system should require no porting effort by the operating system programmer to enable recovery.

For commodity systems: The recovery system should work for widely deployed existing operating systems.

Recover from detected errors of any type: Recovery should not be specialized to a particular type or class of error. It should be possible to recover from any detected fault.

Recovery not usable as a denial of service: Recovery runs the risk of being usable as an avenue for denial of service attacks by repeated recovery and re-execution of a fault. Recovery systems should be designed so that they cannot cause a denial of service.

Clear semantics: Programmers should know what will happen when recovery is triggered. The semantics should be simple, intuitive, and useful to programmers.

2.3 The System Model: Requests

To make an effective recovery system, it is first necessary to define a model of system structure. The key insight is that *a request-oriented view of recovery*

for a system allows novel solutions more powerful than previous work. Requests enter the system through well defined points. Processing of a request by the system may cross traditional sub-system and thread boundaries. Requests may issue requests to other parts of system in the course of processing an outside request.

We start with a request oriented system model which allows greater flexibility than previous work. Systems have been modeled in previous work in three major ways:

As a collection of state: In this model, the memory and open resources represent the running system and are periodically backed-up in a checkpoint; recovered precedes by using the checkpoint to restore the entire system state.

As a subset of state and sub-systems: In this model, state is tied to a subsystem, or, more generally, an object or set of objects and the code which operate on them. These sub-systems of the system are recovered independently by restarting them (and in some cases restoring some persistent state for that sub-system).

With no model: Systems lack a model but have an ad-hoc approach in which recovery happens by the programmer's error handlers (either supported in the language, e.g. exceptions, or by error values).

Request-orientation allow the recovery entity to more closely match the structure and logical operation of many systems. Requests subsume recovery of well engineered sub-systems; an entry into the sub-system is a request. Concurrent requests are separately recoverable, even within the same subsystem as it is the requests which are the entity of recovery, not the subsystem.

Request-oriented recovery also seems an intuitive match to a programmer's notion of system behavior, thus matching the recovery mechanism more closely to the system architecture while providing stronger guarantees than either exceptions or ad-hoc recovery. Applications make requests of an operating system: requests to open a file, requests to read or write through a file descriptor, requests to open a socket, and requests for memory pages. Many protocols are structured as requests. The SMTP definition [45] describes email transport as a series of commands and replies: "The server responds to each command with a reply"; this is a request oriented system where the requests are the commands. The HTTP definition [28] explicitly uses the language of requests. Similar arguments can be made for FTP [59], SSH [92, 90, 93, 91], CORBA [39], and many other network protocols. TCP [58] and IP [57] can both be seen as describing a request-based system between machines. In applications, libraries provide services that can be viewed as requests: `malloc` to allocate memory, `printf` to write to the console, `rand` to get a random number, and `qsort` to have an array sorted. Toolkits are a set of requests between the application and the toolkit. In Qt, the toolkit sends a "QEvent" to handlers in the application. This is a request to process some action, such as a mouse move. Applications send the toolkit requests to paint the screen, etc. Requests appear everywhere you look in the software stack, from protocol specification to interfacing with a hardware device.

2.4 Design Space

The design space of recovery is large. Recovery Domains pick out an aggressive, but still practical, point in this space.

The first design choice is whether faults are visible to the system. We choose to make faults visible to one thread, but invisible to the rest of the system. We do this to enable the handling of faults which are permanent. Reporting the fault to the requester allows us to avoid automatic re-execution of the fault. The rest of the system is see system state "as-if" the fault never occurred.

The second design choice is what recovery happens in the event of a fault. Options used by other systems range from restoring the full system to a previous point to simply transferring control. We restore the system state which is tainted by a fault, but leave the rest of the system alone. This allows some threads to make forward progress in the event of a fault.

In operating system recovery, an important design choice is what components of the system should be recoverable. Most approaches choose just drivers or dynamic extensions. While this can narrow the interfaces which the recovery system must deal with, it leaves parts of the system unprotected. We target the entire operating system with Recovery Domains. We want to recover from faults in as many locations as is feasible.

An important choice in design is what language to support or require. Use of type-safe languages can simplify recovery and are required by some recovery systems. We choose to support operating systems written in C (and any other compiled language) to enable recovery for most existing operating systems.

Finally, the design of recovery requires choosing the an architecture for the operating system. Many whole operating system recovery systems use micro-kernels. These often have cleaner interfaces, better isolation between components, and allow more fine-grained restart of components. We do not require a particular architecture. Although all the advantages with micro-

kernels in regard to recovery do apply to Recovery Domains, we do not depend on this. To have a practical system for commodity systems we support classic monolithic kernels.

Chapter 3

Recovery Domains

Recovery Domains, while not specific to operating systems, are designed to operate in systems with specific patterns. Recovery Domains are organized around the notion of a request as discussed in more detail in section 3.2. Within this over-arching framework, many substantively different implementations are possible.

This chapter explains the basis, goals, and semantics of recovery domains independent of a specific implementation. It discusses the types of domains, what they intend to cover, how execution is paired to domains, and how programmers are meant to interact with the system.

3.1 Concepts and Terms

This, and subsequent chapters, make use of the terms listed below. These terms will be explained in more detail in latter sections.

Recovery: The restoration of an erroneous state of the system to one which is acceptable.

Request: An operation performed on behalf of another entity. The other entity may be either software or hardware.

Requester: The entity issuing a request.

Request-oriented System: A system whose actions or computations are largely initiated by outside events and entities.

Fault Detector: Language or runtime features, compiler instrumentation, programmer inserted checks, or other sources of logic which detect the runtime violation of a desired constraint.

Recovery Domain: The runtime entity containing information about a request, including control-flow information, state changes, and data-flow between requests

Recovery Event: An invocation of the recovery system to perform a recovery, usually at the request of a fault detector.

Context-Independent Request: A request which does not depend on the state of the requester; the relevant and modified state is private to the request in that it is not accessed by the requester.

Recovery Execution: Execution under the control of the recovery system after a fault is triggered but before control is returned to the system.

Recoverable Domain: A domain is *recoverable* if the system can recover from an error during the execution of the domain.

Parent Domain: The domain that is executing when the entry point of a new domain, D, is reached and D begins execution. By definition, the parent domain will be in the same thread.

Ancestor Domains: The set of domains reachable through the parent relationship.

Sub-request: A request make in the course of processing another, existing request.

Commit: The transition of speculative state, state which has been changed based on conditional operations, to permanent state, state which is not dependent on conditional operations. A commit indicates that the condition is resolved and the operations dependent on that condition should have and did execute.

Output-commit: A state change which cannot be undone since it or its effects are visible to the outside environment, e.g. network activity, writes to storage, or changes to the display.

Error Virtualization: The use of a preexisting, programmer-defined type of error in place of an error defined by a fault detector which is not in the original system.

Dependency Tracking: Monitoring and recording data-flow of speculative values between requests to establish a partial commit ordering.

3.2 Recovery System Organization: Request-Oriented Systems

The key organizing principle of this recovery system is the concept of a request. A request, in this sense, is an operation in the software that performs some action on behalf of another entity, even if that other entity is the software itself. Making a request is an explicit invocation of the operation. Sources of requests may be internal to a specific piece of software or come from other software or hardware. Hardware tends to make requests through interrupts, though not exclusively. Software uses interrupts, function calls, and privilege-changing instruction (such as "Call Gates" on the Intel 386). Although, arguably, every instruction is a request to hardware and every line

of code is a request (or several requests) to change the state of the system, we focus on higher-level requests than these.

Recovery is tied to requests for the simple reason that they correspond to high-level semantics of many programs. Hence request-oriented recovery is both easy to reason about by programmers and does not require complex mapping by the compiler from existing code to recovery entities.

Request-oriented systems, those whose executions are largely driven by outside entities, are a common software organization structure. This structure arises fairly naturally due to either the nature of the software or modular engineering. Operating systems and server software, such as database servers and web servers, are often explicitly request oriented. Broad use of libraries cause most systems to have substantial request-oriented internal communication. Graphical user interfaces, for example, have at their heart an event loop that translates user requests (e.g. a mouse click) into a request of the application (e.g. saving a file). An HTTP server sends files to clients in response to requests from those clients.

Requests themselves may make requests in the course of processing. This arises from the notion of a module or sub-system. A sub-system provides an external interface used by the larger system to issue requests to the sub-system. These sub-requests may cross thread boundaries (such as putting a request on a work queue). Besides their important contribution in helping programmer to manage the complexity of large systems, sub-systems and their interfaces improve the granularity of recovery by reducing the scope of execution tainted by a fault.

In operating systems, requests come from three sources. The first source is system calls. These are calls into the operating system by applications requesting some operation be done. This is a quintessential example of a

request-oriented system. The second source of requests is interrupts. These are requests by hardware for attention. In the course of processing either of these request types, operating systems may make internal requests. This is the third source of requests. For example, an open system call will make several internal requests: looking up an i-node in a cache, allocating resources, checking permissions, etc. These internal requests come out of the structure of the operating system.

Server software has a model similar to operating systems, receiving its request from sockets instead of system calls. The exposed layer of many servers makes viewing the system from a request oriented standpoint straightforward.

General software, especially event driven software, often has substantial portions that are request-oriented. This type of software is not the focus of this dissertation, but may often be amenable to using Recovery Domains. Graphical applications have, either internally or in a library, an event loop that dispatches events, such as keystrokes or mouse clicks, to appropriate handlers. Events can be viewed as requests in such a system. Libraries create a natural request layer due to the modular design. Calls into most libraries are natural internal requests. This includes basic resource allocation and deallocation functions such as `malloc` and `free` as well as calls such as `stringdup` and `pthread_create`. Function calls and object method invocations, as an abstraction, map well, in most cases, to the notion of requests.

3.3 Requests Context

Requests exist within some context. This context primarily includes two factors. First, there may be a request which was already executing on the thread.

This previous request must be preserved and restored after the completion of the current request. It is not necessarily the case that the previously executing request initiated the new request. Interrupts initiate new requests on a thread which was already executing a different request. The second component of the context, then, is why the request started. A request started explicitly by another request is a sub-request. Requests from outside the system, such as an interrupt or a signal are not sub-requests, even if there was a previously executing request on that processor.

In the general case, sub-requests are dependent on the request which issued them. If a request ends, its execution is still speculative if the requester is speculative. However, a common structure, especially across sub-system and library boundaries, is to have requests which are not dependent on the state of the requester. These requests are to modules or sub-systems which encapsulate their state well. The prime example of this is `malloc`. `Malloc`, and functions like it, are isolated from the caller, revealing little or no information about their internal state. We call these requests context-independent requests.

Requests which represent an external entity forcibly transferring execution to a new request, as happens during an interrupt, are also context-independent. Although a request may have been executing, the source of the new request is an entity privileged enough to preempt the existing request and have its request processed before returning to the existing request.

It is often the case when context-independent requests are sub-requests, as it is in the case of `malloc`, that there are context-independent inverse actions. For `malloc`, it is `free`. This observation, which is not unique to this dissertation, allows higher-level recovery at the semantic level of the requests rather than at lower levels. This allows for more efficient and flexible imple-

mentations of recovery.

3.4 Recovery Domains: the Runtime Entities

The basic unit of recovery is a *recovery domain*. All code executes within some recovery domain. A recovery domain is an interval of execution demarcated by calls to the domain start and end operations. recovery domains track which state changes are necessary to undo when recovering as well as how data flows between requests. A recovery domain logically consists of:

Request entry point: This is the point at which the request starts and the point at which a new recovery domain is created. This demarcation is used as the point to which control returns in the event of a fault.

Request exit point: This is the point at which the request is complete. Control is returned to the requester and the recovery domain commits if it is able and allowed to do so.

Recovery action: The programmer supplies an action to perform after recovery is initiated. This action is how a fault which is recovered from is visible to the system. The most common action is to return an error code. Another useful option is to ignore the error (e.g. statistics updates may not affect correctness and can be ignored if they fail).

Inverse action: The programmer may supply an action which semantically undoes the effect of this request. A common example would be a `kmalloc` request supplying `kfree` as an inverse action or `vfs_open` supplying `vfs_close`. The recovery system may use the inverse action to undo a request during recovery rather than directly restoring the modified state.

Dependent domains: Domains which have data dependencies, either read-after-write or write-after-write, or are sub-requests of this domain are dependent on this domain. If a domain must recover, all dependent domains must be recovered and restarted. This ensures that data used by domains is produced only by domains which did not fault.

Each recovery domain executes within a single thread. Cross-domain and cross-thread dependencies are recorded. Dependencies are formed by monitoring data flow from state modification to accesses by recovery domains. recovery domains are not checkpoints: a rollback of one domain does not imply a rollback of all domains or all threads. As explained below, recovery domains may be nested to take advantage of existing error recovery paths and code that has a semantic inverse.

There are five types of recovery domains. Table 3.1 and Table 3.2 summarize the semantics of the five domain types. Because of the nature of the kernel entry points, it is not possible to statically determine the type of domain a request starts. Specifically, the domain type depends on both the declared behavior of the request and the context in which the request is issuer.

3.4.1 Comparison to Transactions

A recovery domain is transaction-like in that it provides failure atomicity. However, recovery domains are not a replacement for synchronization nor are they transactions. They do not provide any isolation; requests can interact in any legal way. No additional durability beyond what was already present in the system is provided by recovery domains. Some consistency is provided, but only to memory state; recovery domains cannot ensure consistency of device state or of communication with external systems. While a

Domain Types (Recoverable?)	**Semantics**	**Example Use**
Basic (Y)	Forms dependencies with any active domain as required, but committing is dependent on the commit of the parent.	Operations for which the kernel already handles failure, e.g., `open`
Reversible (Y)	Has a known semantic inverse operation that is sufficient for rollback, i.e., ignoring internal state changes.	Allocators; reference counts; *not* locks
Independent (Y)	Same as basic but for operations which do not have a parent.	System calls; interrupts handlers
Transparent (Y)	Upon Successful completion, state changes ignored durring rollback.	Operations whose state changes do not affect the correctness of the kernel; e.g. LRU numbering of pages, disk cache read-ahead policies
Unlogged (N)	Forms no dependencies, but tracks all writes to break dependencies in other domains.	Interrupt handlers and other device manipulation code

Table 3.1: Primary Types of Domains

	Normal	**Reversible**	**Independent**	**Transparent**	**Unlogged**
Reverts state changes	Y	Y	Y	Y	N
Recover dependent domains	Y	Y	Y	Y	N
Ignores dependencies on parents	N	Y	Y	Y	Y
On success, only log inverse	N	Y	N	N	N
On success, trys to commit	N	Y	Y	Y	N/A
Parent is aware of fault	Y	Y	Y	N	N
May be deferred by the programmer	Y	N	Y	Y	Y
Normally has a deferred inverse	N	Y	N	N	N

Table 3.2: Differences in Domain Types

transaction commits or aborts at completion, a recovery domain may exist after the request's completion if it depends on speculative state (state modified by other domains which have not committed yet). Since the processing of a request can traverse threads, recovery domains may compose across threads.

3.4.2 Basic Recovery Domains

Basic domains are the most common unit of recovery. Basic domains are intended for regions of execution that have existing error handling code at their exit. In the event of a fault, basic domains are rolled back independent of their parent and the request appears to fail in an expected way as specified by the programmer. Control is returned to the parent at the domain entry point, which can then handle the error as desired. Basic domains are the common form of nested domains (sub-requests) that execute during the processing of a request.

Basic domains participate fully in logging and interference tracking. Logically, these domains log all memory writes. This log contains all values which must be restored in the event of a fault in the domain. Memory reads are monitored to discover runtime data dependencies on other domains.

Basic domains fully participate in the interference tracking mechanism. All reads of speculative values, i.e. values written by domains which have not committed, form a dependency from the writer to the reader. Any value written by a basic domain is speculative until it commits, hence most readers of that value will form a dependency on the basic domain. The exceptions to this last rule are due to the nature of some classes of readers, not to the nature of basic domains.

Basic domains do not commit on normal completion. Instead, their log and dependencies are merged with those of their parent, thus passing on the

responsibility for committing state to their parent. This commit behavior is similar to that of "closed nested" transactions in the transactional memory literature [80] but, like all recovery domains, a key difference from transactions is that they are never rolled back during normal execution.

Basic domains are for sub-requests. If a basic domain is used as an entry-point into the kernel, the domain is considered an independent recovery domain which is discussed next.

3.4.3 Independent Recovery Domains

Independent domains handle requests which represent a new entry into the system. A system call is a common request which will normally be handled as an independent domain. These domains serve as a hard border between portions of the system (e.g. user space and kernel space). Not all entry-points are statically independent domains. A system call from an application represents a new request, say `sys_open`, whereas that same system call could be issued by the kernel in the processing of a different request, say `sys_exec`. Such a request is not an independent request when issued from another request because it is a sub-request; it is only part of the handling of an existing request.

Independent domains differ from basic domains primarily in how they interact with their ancestors and when they commit. Independent domains encapsulate local, context-free requests. This is to say that something declared as an independent domain is assumed to be free of dependencies on its calling context. It is not assumed that an independent domain is free of dependencies on domains executing in other threads. For example, a `alloc_page` in the Linux kernel is a basic allocator which is encapsulated with an independent request. A request by one thread to `alloc_page` may from dependencies on

a concurrently executing request to `free_pages` since both requests modify the same free-list. Hence an independent domain does not form dependencies on its thread-local ancestor domains, but does interact fully with the dependency mechanism with respect to all other threads.

Because it does not form dependencies on its ancestors, an independent domain may commit as soon as it completes if all its dependencies have committed. If an independent domain can not complete immediately, its caller forms a dependency on it. This means that if another thread on which the independent domain is dependent aborts, hence causing the independent domain to abort, the caller domain will also abort. In normal operation, this window is expected to be short, thus allowing the independent domain to commit quickly reducing the amount of speculative state tracked at runtime.

3.4.4 Semantically Reversible Recovery Domains

Reversible domains exist for operations for which a semantic inverse action can be defined, independent of the context in which it was called. This independence means, most importantly, that two such operations (on the same or different threads) *can be performed or rolled back independent of the other* (in the absence of an internal fault *within* one of the domains). This is referred to as the *context-independence* property.

For example, two calls to `kmalloc` (in Linux) may both update common internal allocator state, producing an apparent dependence, but at the level of the logical allocations, there is no dependence: either one can be performed or rolled back independent of the other. Furthermore, this rollback can be performed by any thread: it simply needs to have the address of the allocated memory. Many counters, including reference counting operations on heap objects, are also important cases of reversible domains. Like allocations, these

operations are extremely frequent, and create spurious dependencies between domains. They can be treated as reversible domains because (a) the inverse for an increment is simply a decrement, and vice versa, and (b) two operations on the same counter in two different domains (in the same or different threads) can be independently committed or rolled back.

When a reversible operation is complete, it can commit *ignoring any dependencies* on its parent. This optimization greatly reduces the number of interdependent domains that must be committed together or rolled back together on an error. For example, many concurrent allocations would become interdependent and cause their invoking threads to be dependent on each other, when in fact, at a semantic level, the context-independence property ensures that the operations can be committed independently of their parents.

Reversible domains perform all the state and dependency tracking of an independent and basic domain so *internal faults* within the domain can be detected and recovered from. On a fault, just the reversible domain and any dependent domain (such as multiple overlapping calls to an allocator) are rolled back. The recovery action, such as returning an error code, is executed.

However, on the successful completion of a reversible domain, the state changes by that domain are committed and the inverse action is logged with the parent. If the parent must rollback, this inverse action will be called to logically undo the operation of the reversible domain. So for example, a `malloc` will be logged and if the domain calling `malloc` aborts, `free` will be called on the allocated object. In principle, internal logging is *optional* and for well tested and trusted code could be turned off by the programmer to improve performance, although the recovery domain implementation presented here leaves it on and a domain type for this type of behavior is not defined.

```
1  void foo(void* ptr) {
   ...
   // Use ptr
   ...
   free(ptr);
6  ...
   // The freed memory should not be reused until the
   // free is known to not have been speculative
   ...
   if (bad_thing_happened)
11   rec_dom_abort(); // a recovery was triggered
   ...
   }
```

Figure 3.1: `free` releasing a resource which can be used by another thread

Since many reversible domains often involve the ownership or allocation of resources, the recovery system must ensure that a release operation (the inverse of an acquire) does not occur until the parent domain commits. This prevents reuse of speculative resources.

Let us assume that release actions are executed in place. Consider the allocation in Figure 3.1. If the `free` is executed, it will commit since it is an independent domain. The memory then may be allocated by a different thread. If the request faults and triggers recovery, then the memory which was freed will be in use and it cannot be rolled back safely. We wish to put the memory in the state it was at the start of the request, but we cannot because it is reused by another object. To solve this, we delay the the free until after the domain commits to ensure that the memory isn't reused by a different thread.

Thus inverse actions, when they appear in code, are delayed until the commit of the domain in which they appear. This is also the reason why lock acquires are *not* treated as reversible though they have an obvious inverse: deferring the inverse (the unlock) could cause deadlock if it is delayed until

after the acquisition of a different lock.

3.4.5 Transparent Recovery Domains

Transparent domains exist based on the observation that not everything matters nor has to be exact. Many places in a system, information is kept for statistic reporting. It is desirable that these operations be accurate, but if in the rare event of a fault they may deviate slightly from what they would in correct execution, no harm will result. For example, if statistics reporting from the network layer reports having processed 2^{30} packets rather than $2^{30}+1$ packets, the end-user is not likely to notice or care. If, however, packet counts used as TCP sequence numbers are not accurate, the protocol fails. This indicates that the first case is a candidate for a transparent domain, since small inaccuracies in the information is not critical while the second case are not a candidate for a transparent domain since accuracy is critical.

A second case transparent domains address is operations that are strictly used for performance optimization and have no impact on the semantics of the system. Tracking the LRU numbering of memory pages is one example: resetting all the numbers (except for any used for pinning memory) to arbitrary legal values will not affect kernel correctness. Other examples include caches and some memoization. For such domains, the state changes and their dependences do need to be logged to ensure recovery from errors *during execution of* such a domain. Upon successful completion of such a domain, however, any dependencies formed by the execution of the domain can be ignored, i.e., the domain can be committed immediately, since transparent domains leave the semantics of the system intact.

3.4.6 Unlogged Recovery Domains

The final type of domain is the unlogged domain. This domain is for extremely trusted code as well as for code which must not abort. An unlogged domain effectively commits each write as it is issued; it transitions from speculative to completed to committed after each write. This ensures that state changes caused by stores are never speculative. Because unlogged domains cannot abort, reads are handled specially. An unlogged domain cannot undo actions based on speculative values read from memory. To reflect this, reads are not tracked for dependency formation and reads cause a logical write of the value read back to the location. This ensures that the location is not considered speculative and will not be rolled back to a state before the unlogged domain read it.

For example, we want the system time to increase monotonically. Therefore, the request to update the system time after a timer interrupt is run in an unlogged domain. This is trusted code which writes a commonly read value. By using an unlogged domain, we ensure that time will never move backwards due to a rollback of the time update. We further prevent readers of the system time from forming a dependency on the interrupt handler, thus saving some dependency tracking.

3.5 Execution Modes

Execution takes place in one of two modes: normal mode and recovery mode. Most execution should take place in normal mode. Recovery mode is only entered in the event of a recovery event triggered by a fault detection mechanism. After the recovery system completes recovery, execution resumes in the normal mode.

3.5.1 Normal Execution

During execution in normal mode, all threads and requests execute without significant deviation from their execution in the absence of the Recovery Domains system. All memory operations are monitored and dependencies between domains are recorded in accordance with the types of domains involved. Recovery domains commit when they are able and never roll back.

In the absence of any fault, normal mode execution appears virtually identical to the code running without the recovery system. Execution deviates some amount due to the the time and memory needed to manage recovery domains and monitor state changes. This variation can potentially change the interleaving of operations between threads, but only to an interleaving that would have been allowed in the original system. Recovery domains do not change or eliminate locks, so any change to the order locks are acquired is a change allowed by the locking discipline.

3.5.2 Recovery Execution

Recovery is trigger by a fault detector when a fault occurs. These are explicit calls to the recovery system to initiate recovery. These may be inserted in the code by an instrumentation tool, inserted by the programmer, or in the runtime of a safety checking tool. There are no implicit recovery events in the semantics of the recovery system. All recovery must be initiated by the programmer or by an automated tool.

Recovery events are expected to be very rare, hence the semantics of recovery mode allow latitude in implementation. During recovery mode, the recovery system is allowed to suspend all threads while recovery is executing (though an implementation need not do this). Semantically re-

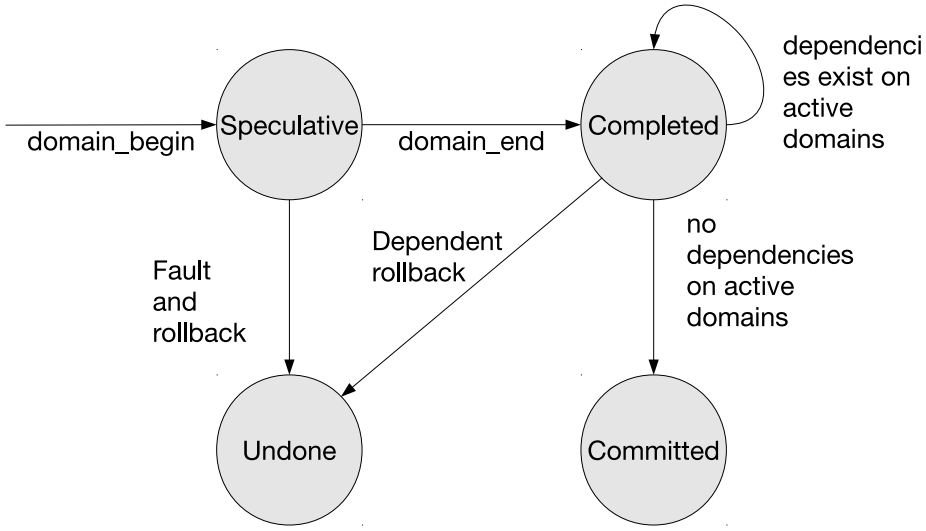

Figure 3.2: Commit protocol

versible requests are undone by executing inverse actions in reverse order per thread, but with no guarantee of orderings between inverse actions of different threads nor any guarantees of which thread the inverse actions will be performed. These loose orderings are allowable by the definition of context-independent requests.

Prior to inverse actions for semantically reversible requests being executed, the memory state is restored to appear as if the faulting request (and any dependent requests) never executed.

3.6 Committing

A recovery domain goes through as many as three states in order: Speculative, Completed, and Committed. These states and their transitions are illustrated in figure 3.2. All but unlogged domains begin in the Speculative state.

When a domain exits it transitions to the Completed state. When a domain in the Completed state has no more dependencies on uncommitted domains it transitions to the Committed state. Each transition is accompanied by a set of actions which will be described. Since dependencies can form chains ($A \rightarrow B \rightarrow C$), completed domains must look at the transitive closure of the dependence graph and check that they have no dependences on a Speculative domain to transition to the committed state.

When an unlogged parent transitions from Speculative to Completed state, it always enters the Committed state immediately; this step is explained below.

When a basic domain transitions from Speculative to Completed state, it merges with its parent domain such that in subsequent execution the two domains are synonymous. It does not logically exist after this point, but rather is part of its parent domain which remains in the Speculative state until such time as its parent domain commits.

When an independent, reversible, or transparent domain transitions from Speculative to Completed state, it removes all dependencies on its parents if any exist (allowed by the context independence property, explained in section 3.4.4). This should be rare, but can arise from calling conventions for passing structures which pass a pointer to a stack-allocated object. If a reversible domain has a logged parent, the domain adds a dependence edge from the parent to itself. The domain then attempts to enter the Committed state.

Attempting to enter the Committed state consists of checking that no dependencies exist on speculative domains in the transitive closure of the dependence graph. If no such dependencies exist, the domain transitions into the committed state. To do so, it performs a series of actions to remove itself from the system. First, it executes all logged delayed operations (resource

releases). It then marks all memory locations as non-speculative for which it is the most recent writer. It then removes all dependence edges to and from itself in the dependence graph. At this point no more references to the committed domain exist in the system and the domain's meta-data may be deleted. Note that for an unlogged domain, all these operations are no-ops as it will never have logged state changes; thus no domain will have dependencies to an unlogged domain. After a domain commits, each domain in the Completed state attempts to enter the Committed state (since the dependence graph has just had edges removed, more domains may be able to commit).

Note that the commit protocol is thread-agnostic. Even though dependencies may cross multiple threads, the protocol does not care about whether an incoming or outgoing dependence edge is from or to a different thread, or whether some delayed operations may involve updating shared state. Domains may be committed on any thread, even if that thread is different from the thread they originally executed on, and commits can happen while other threads continue to execute normally.

3.7 Predictable Error Semantics

Recovery Domains provide the following (predictable) error semantic to client code. To aid in explaining this semantic, as way of definition, we say domain *B* is directly *dependent* on domain *A* if domain *B* reads a memory location written by domain *A*.

- control flow in the faulting thread is returned to the start of the "faulting domain," i.e., the recovery domain within which the fault occurred;
- the memory and register state of the thread that executed the faulting domain are restored to those that would have occurred if the faulting

domain had not executed at all (i.e., there are no visible state changes beyond that point), unless the domain was *Unlogged*, as explained in Section 3.4.6;

- any other thread that executes a domain that is *dependent* on the faulting domain (directly or transitively) has its control flow and state rolled back to the start of the earliest such dependent domain, and its execution continues as if that domain itself encountered an error.

Under normal execution, there should be no visible change to the output of the system. In the absence of output-commits, a recovery event will trigger a rollback of just the domain affected to the entry point of the domain in the execution stack in which the domain is executing. Any dependent domains, as defined in 3.4, are rolled back to their entry point and restarted. All domains which are rolled back have their undo log walked and all memory changes undone and all inverse or compensating action executed (e.g. `free` if a `malloc` was called). Without output-commits, this allows parallel requests which share state with a faulting domain to be transparently isolated from the fault.

Output-commits are considered to immediately commit the domain in which they happen. In an operating system, if a domain issues commands to a hardware device, that domain commits, making it unrecoverable. This is generally true for user-space applications also, but in some cases inverse actions can be made for what would otherwise be an output-commit. Calling `malloc` has an inverse action, namely `free`, so it can be treated as undo-able rather than as an output-commit.

```
asmlinkage long sys_open(const char *filename,
                         int flags, int mode)
{
 long ret;

 /* Standard system call domain */
 REC_SYSCALL(REC_FLAGS_NORMAL);

 if (force_o_largefile())
  flags |= O_LARGEFILE;

 ret = do_sys_open(AT_FDCWD, filename,
                   flags, mode);

 /* end system call domain          */
 /* return -RRECOVERY on fault      */
 /* return ret on normal execution */
 REC_END(-RRECOVERY, ret);

 return ret;
}
```

Figure 3.3: Annotated open system call

3.8 Programming Model

Programmer involvement in this recovery system is minimal. At a minimum, programmers must identify requests. Figure 3.3 shows a prototypical annotated system call which is annotated to return an error value in the event of a fault. Once requests are delineated, the necessary work by the programmer is complete. However, further work can make recovery less invasive when triggered and lower the overhead under normal operation. Cross-thread requests should be identified as requests to enable complete recovery of a single logical request. Requests with an inverse action (e.g. `kmalloc` or `vfs_open`) should be identified and associated with an inverse action.

Porting an existing system to the recovery system can be incremental. Once the entry requests, those requests originating in the outside world, are identified, recovery will work. Continued refinement will lead to faster recovery and faster error-free execution as well as more precise error handling.

3.9 Examples

A system call serves as an excellent example of a recovery domain. A hypothetical system call is depicted as both succeeding and faulting in this section. The specifics of what system call it is or what the sub-requests are is ignored to focus on the behavior at the recovery domain level.

3.9.1 A Successful Request

Figure 3.4 depicts a hypothetical system call. Execution begins at the start of the request (1). The request makes a sub-request, to a generic helper library at (2) which starts a new protection domain (so that if the sub-request fails, the

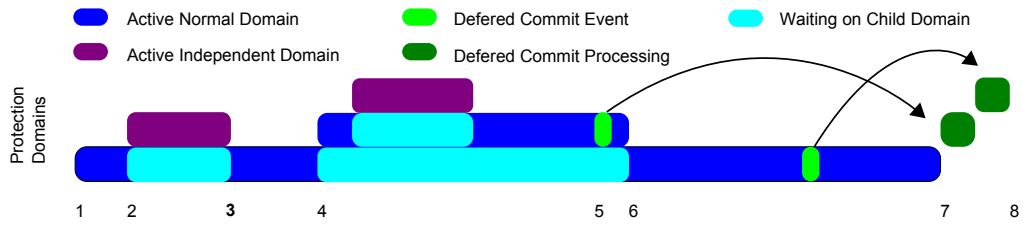

Figure 3.4: A hypothetical system call. 1: Start of a protection domain. 2: Start of an independent nested protection domain. 3: End of the nested protection domain; the fact of it's successful completion is logged in the parent domain. 4: The nested protection domain executes a child domain. 5: A deferred commit region is hit, and not executed. It is logged and deferred until commit time. 6: Nested domain ends, and becomes dependent on its parent for committing. 7: The protection commits. As part of this process, deferred regions are executed. 8: All deferred regions are complete and the system call is completely committed.

parent can clean up as specified by the programmer). Execution of the sub-request completes and it is committed since it is a domain which is marked as committing regardless of its context. The same happens at (4), but this sub-request makes a further sub-request. At (5), an action, such as a resource free, happens, which may not be visible to other threads until the request is known to commit. Thus it is logged and postponed. This prevents unintentional sharing of state between requests. The sub-request completes at (6). At (7) the main request completes and, after the runtime checks for safety conditions, commits. As part of the commit process the actions are deferred until commit is completed.

3.9.2 A Faulting Request

Figure 3.5 is similar to figure 3.4 until a fault occurs in the sub-request. At (5), the sub-request fails and the recovery mechanism takes over. The sub-request is rolled back, and a semantic inverse action for the sub-sub-request is executed to semantically roll it back. An error is returned to the main request

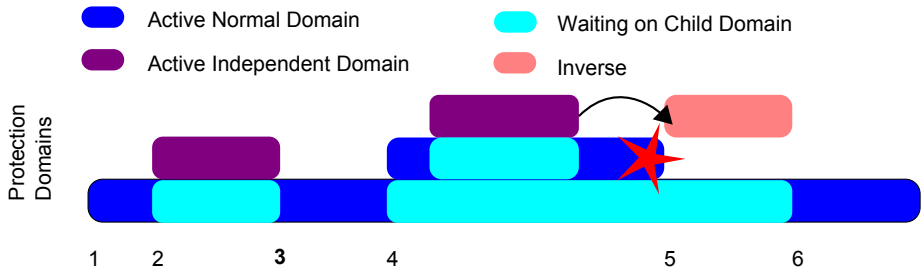

Figure 3.5: A hypothetical faulting system call. 1: Start of a protection domain. 2: Start of an independent nested protection domain. 3: Successful completion of the nested protection domain; this is logged in the parent domain. 4: A nested protection domain executes a child domain. 5: A fault occurs. Since the independent domain was reversible, its registered inverse is executed. 6. Rollback of memory state for the child domain is complete and all inverse operations have been executed. Control resumes on the error path at 4.

(as specified by the programmer of the sub-request) and the main request deals with the error in the manner it chooses, knowing that the effects of the sub-request are undone.

3.10 Fault Detectors

Recovery Domains do not detect faults or corrupted state in the target system. Recovery Domains only handle the recovery and restoration of the system once an outside entity, the fault detector, has indicated the need to recover. Since recovery is oriented around requests and does not necessitate rollback of the entire system, it is necessary to consider what constraints and requirements Recovery Domains place on fault detectors in order to orchestrate successful recovery.

The first requirement for recovering from a fault is that the fault be detected in the time frame of the request in which it occurred. Once a request is complete, it may commit, preventing recovery from a fault which occurs during its execution. Since recoverability may not survive the end of a request,

to recover from a fault and undo the corruption of system state caused by the fault, the fault detector must catch the fault in the execution time-frame of the request.

If a fault corrupts state in such a way that the fault detector will detect uses of the corrupted state and flag a fault, then the corrupted state will not spread to other requests, but only because the uses of the corrupt state cause recovery. In this case the original, unflagged corruption remains. To recover the initial corruption of state by a fault, it is important that a fault detector detect a fault in the execution context of the request in which the fault occurs.

It is highly desirable, though not necessary, that a fault detector be "eager". It should catch a fault before the fault can be expressed as corrupted system state. In general this means the flag a fault before there are dependent writes to memory. This prevents unintended executions in dependent requests. For example, preventing an incorrect write to a function pointer may prevent a different request from jumping to a incorrect location. As long as the original request eventually faults, the dependent request using the function pointer will be rolled back too. There is a window of vulnerability in which either the original request or the dependent request can initial an unrecoverable action, such as starting an unlogged domain or writing to a hardware device. It is therefore desirable that faults be caught as soon as possible.

3.11 Conclusion

The Recovery Domains system consists, at its heart, of runtime entities, namely recovery domains, and the interconnection of those entities. Recovery Domains are fashioned to match requests in a request-oriented system, a

common programming pattern found in operating systems and many other software systems. The interconnectedness of recovery domains comes from two sources: sub-requests form a parent-child relationship and memory dependencies form a commit-dependency relationship. These relationships govern when domains commit and what domains rollback in the event of a fault.

Recovery Domains provide simple semantics and a simple programming model. Placing a low burden on the programmer is intended to minimize the effort of porting an operating system kernel to the recovery system.

Chapter 4

Reference Design and Implementation

While Recovery Domains define a set of semantics and structure, they do not dictate an implementation. Many actualizations of the system could be imagined which maintain the semantics while differing wildly in design and implementation. This chapter provides a reference design and implementation. It discusses our porting a real operating system to use recovery and our experiences with the complete system.

An important component in a deployed system is the detection mechanism that signals faults. The recovery system design and implementation is independent of any detection mechanisms the programmer chooses to apply. Fault detectors are beyond the scope of this work, though we will describe a hypothetical combination of the memory safety detector in SVA [21] in chapter 7.

This reference design and implementation consists of two main components. The first is a set of compiler transforms to instrument the target system and transform programmer annotations into recovery domains. The second is a runtime which maintains all logs, tracks dependencies between domains, manages committing domains, and performs roll back at a recovery event.

This chapter walks through the runtime components including the logging system, the use of versioned memory, and how commits and aborts are processed. It then turns to the porting of Linux 2.4.22 and lessons learned from that port.

Parent pointer	previous recovery domain on the stack
Root pointer	request from the outside world
Peer pointer	linked list of nodes at constant tree depth
Child pointer	next level in tree
Dependence set	out-edges in the dependence graph
Undo log	old values and version numbers of written locations
Delayed action log	operations to perform on commit
Inverse log	operations to perform on rollback

Table 4.1: Recovery Domain Structure members

4.1 Reference Design

This chapter describes a reference design of Recovery Domains that is based around maintaining an undo log for rollback and versioned memory for dependency discovery. Control flow is managed with `setjmp` and `longjmp` style operations. With these design elements, we can map directly many aspects of Recovery Domains, leading to an implementation that is understandable to the operating system programmer. This understandability, we feel, is critical in a reference design of something on which reliability depends. Unnecessary complication yields more opportunities for bugs and less overall trust in the system.

Recovery Domains are tracked at runtime by allocating an object for each new dynamic domain. The content of this object is summarized in table 4.1. Domains contain a parent pointer, which points to the previous recovery domain running on that thread. This is the domain that is restored when this domain ends. This pointer does not care if the parent domain caused this request or if this request was from an interrupt or other outside source. The parent is simply the domain which will be restored at completion.

The root pointer contains the closest parent domain which is independent. Basic domains are sub-requests and depend on their parent to commit.

Following the parents until the first independent, not basic, domain yields the parent that is ultimately responsible for committing this domain. This parent is stored in the root pointer to optimize dependence graph building as well as simplifying the check to see which domains can commit.

The peer and child pointers are used to hold a tree or list, depending on the context. The child pointer is used by a domain to point to a list of child domains (sub-requests). This is a singly-linked list formed by the peer pointer. The list is formed from the front, so if a domain is active, it is at the front of the list. The peer pointer is also used to form a list of independent domains. Independent domains cannot be children of another domain by definition and therefore will not have fellow children in their linked list. The implementation does, however, need to track all independent domains, which are the roots of commits, and does so with a linked list through the peer pointer.

The dependence set contains the out-edges of the dependence graph for a domain. These are the domains which must commit for this domain to commit and the domains which if any abort, cause this to domain abort. In the implementation, this is stored as a sorted array and accessed with a binary search since adding edges is less common than verifying their existence.

Three logs are kept by a domain. They are the undo log, the delayed action log, and the inverse action log.

The inverse action log maintains the inverse actions for sub-requests which are independent. This most commonly is `free`. The entries in the log are records of function addresses and arguments.

The delayed action log contains records similar to the inverse action log, but for actions which are independent and must not be executed until the domain which would execute them has committed. This is used so that resources that are semantically abstracted, such as allocations, are not reused

Lock	protects meta-data
Current Version	version number of location
Committed Version	version of last commit
Last Writer	last recovery domain to make a speculative write to the location

Table 4.2: Versioned memory meta-data

before they are known to be be available.

The undo log contains the state which must be reverted during a commit. Entries are records consisting of a locations, a size, contents, and the previous owner. The first three fields of the record contain the necessary information to restore a location to its pre-domain state. The previous owner field allows the rollback to restore the meta-data for the locations.

4.2 Versioned Memory

The reference design uses a form of versioned memory to keep track of the chain of writers to a location and the last commit of a location. Versioned memory is not used to undo writes, but only to manage dependencies.

Each memory location, logically, has meta-data containing a lock, the last writer, the current version, and the last committed version. These are summarized in table 4.2.

Although the lock protects each meta-data structure from concurrent access and update, it does much more. Access to the meta-data for a location is serialized by the lock, as is access to the location itself. A meta-data lock serves as a lock on the location which the meta-data covers. Since logging, discussed later, requires an atomic read and a write of a location, this lock doubles for that. The lock also blocks interrupts to prevent deadlock. An interrupt-started domain could try to update meta-data that was already be-

ing updated, and hence locked, by the domain it interrupted, thus leading to deadlock. Logging does not use a atomic-swap for the access to the location simply because all access is already covered by the meta-data lock and an atomic exchange operation (like all atomic operations) is expensive.

The current version field is maintained as a reference for the greatest value of the version number stored by any recovery domain. When a domain writes to the location, this value is incremented and logged by the writer. This provides an absolute ordering of writes to the location.

The committed version field indicates the latest version of the location which was committed. Since a location can be written by two threads without intervening reads, the two writes may be by independent domains. If the second writer commits and then the first domain reverts, we need to know that the second write must be preserved. Tracking the last committed version allows the system to discard writes of older versions during revert.

The last writer field stores the domain which wrote the current version of the location. If that domain has already committed, this is null. This is used by readers to form dependencies.

4.2.1 Global Versioning Structures

Storing a meta-data entry for every memory location would be prohibitively expensive. If an entry was maintained for every pointer-sized location the kernel could access there would be a 4:1 memory overhead. On 32 bit Linux, this would amount to using 800MB of the kernel's 1GB of reserved address space. Since this overhead is too high and much of the address space is sparsely accessed, we hash memory locations. The hash function simply takes the lowest order bits, minus the bottom two, to map pointer-sized locations to the hash-table.

Using a hash table can result in false dependencies between domains. However, using a dynamic structure, such as some form of a binary tree, causes significant slowdown. The structure is accessed once for every load and store issued in the kernel. Anything more complex than a direct index of the hashed location dominates the runtime. Thus the performance overhead of a more exact sparse structure is not worth the increase in precision of dependence tracking.

4.2.2 Load and Store Replacement

The recovery compiler must instrument the original program instructions to monitor the load and store traffic. This is done as a wholesale replacement of load, store, and atomic memory instructions. These instructions are replaced with calls into the recovery runtime which updates the necessary meta-data and performs the operations. Loads and stores are discussed below, atomic operations are not discussed but are very similar.

Figure 4.1 depicts the operations performed by the runtime for each store instruction. A store instruction is replaced by a call into the runtime which passes the location (pointer), the value to write, and the current recovery domain. The location is hashed to acquire the meta-data. The meta-data is locked and the previous writer is compared to the current writer. If they differ, the version for the location is incremented. The new version, the old value of the location, and the pointer are logged in the current recovery domain's log. The store is execute and the lock is released. If the current writer is the same as the previous writer for the location, only the store, logging, and lock release happen; no version number needs to be changed.

Figure 4.2 depicts the operations performed by the runtime for each load instruction. A load instruction is replaced by a call into the runtime which

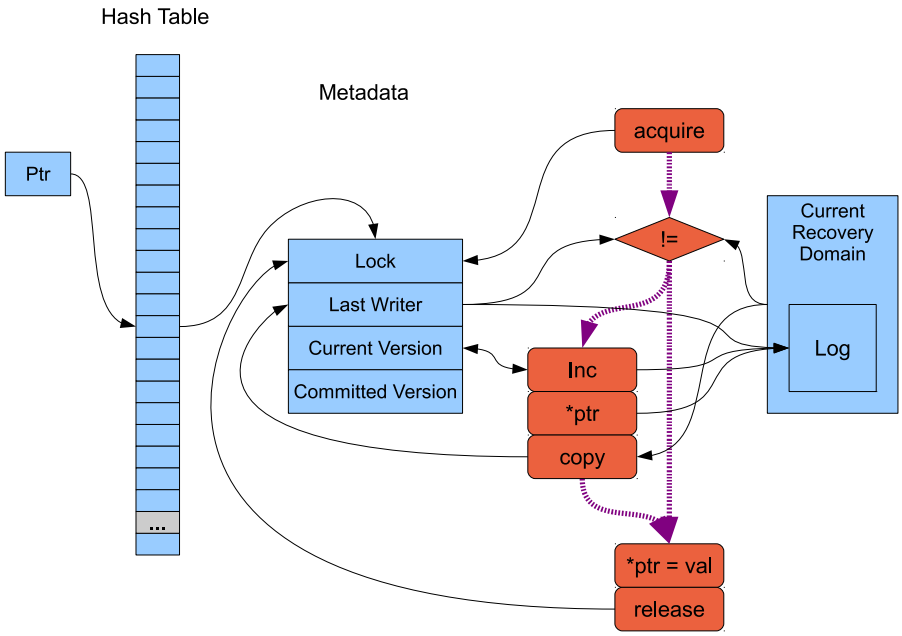

Figure 4.1: Store to a pointer

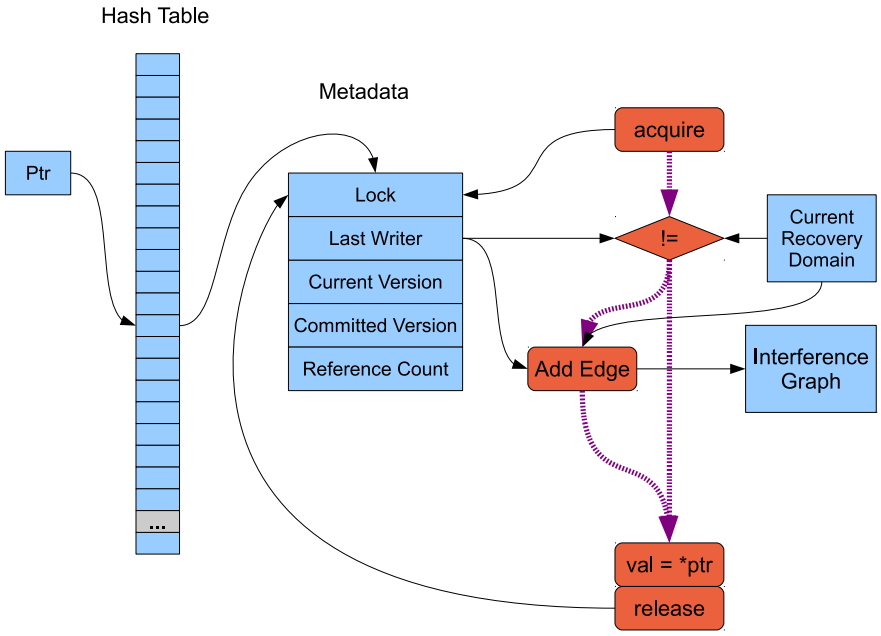

Figure 4.2: Load of a Pointer

passes the location (pointer) and the current recovery domain. The return value is the result of the load. The location is hashed to acquire the meta-data. The meta-data is locked and the previous writer is compared to the current reader. If they differ, an edge is added in the dependence graph from the reader to the writer. The load is execute and the lock is released. If the current reader is the same as the previous writer for the location, only the load and lock release happen.

4.3 Tracking Dependencies Between Recovery Domains

Every recovery domain tracks which domains it depends on and what domains depend on it, creating a dependence graph between domains. This dependence information is important because when a domain experiences an error, it allows the rolling back of only those domains which have been "tainted" by the error. This mechanism correctly handles dependencies that cross threads, which is important due to the inherently threaded nature of the systems we are interested in.

This tracking is represented as a graph. Each recovery domain contains a sorted array of pointers to domains it is dependent upon. This allows a binary search to test whether a dependency exists either for purposes of committing or for purposes of adding a dependency after a read. Only out-edges are stored in the graph.

On a read, meta-data maintained by the run-time is consulted to determine if the location contains a committed value, as explained in section 4.2.2. If so, no dependency information is updated. If it contains a speculative value, a (directed) dependency edge is formed to the domain responsible for the last

write of that location.

On a write, the meta-data for the location is updated to reflect the new writer. If the writer is an unlogged domain, then the write is committed, else the write is marked as speculative and the recorded writer helps subsequent readers form dependence edges.

Tracking dependencies between recovery domains is mainly useful between separate threads; nested domains identify dependencies between domains within the same thread. Domains are nested at run-time, and the nesting structure forms one or more trees that determine when and how domains can commit. To track nesting, the run-time maintains a stack of active domains. All code runs in some domain, even if that domain is the default unlogged domain. This ensures that all code participates in the maintenance of meta-data.

When a recovery domain starts, it inspects the current stack to find the currently active recovery domain. It records the domain as its parent so that, on exit, it may restore that domain to an active state. A basic domain will become a child of that domain and become dependent on it for commit. That is, when the basic domain exits, it will not commit but simply notifies its parent domain, which when it commits will commit the child. All other domain types, however, do not record their parent: instead they starts a new tree. They remember the previous active domain only for the purpose of restoring it on exit.

4.3.1 Graph Simplification and Search

When finding domains to commit, the entire dependence graph must be search to see if a domain can reach an active domain through the dependence edges. If it can, the domain may not be committed. This search is

implemented as a depth first search starting from the domain in question and stopping anytime an active domain is reached. All reachable domains may either be committed or have their dependence edges rewritten. If the domain the search started at can commit, then any domain visited in the DFS can commit. If not, then the set of active domains which prevent commit are a common set of dependencies for all visited domains. The edges can then be replaced by this set, simplifying further traversals.

Alternative Simplification

Initially, a transitive closure was computed by merging the out-edges of each domain with the out-edges of all domains it was dependent on. This is more expensive in both space and time than the DFS-based graph simplification.

4.3.2 Alternative Graph Representations

The primary alternative graph representation we tried was a dense matrix of bits. Each set bit represented an edge from the row domain to the column domain. This representation makes computing the transitive closure relatively easy. It also makes testing for the existence of an edge a constant time operation, rather than a $log(n)$ operation. The primary failing of this representation is its fixed size. The tracking structures need to be allocated statically to avoid requiring a call to the OS to acquire new memory if more domains than the matrix has room for are added. The number of domains in the system was seen to be very bursty. Thus much of the time the fixed size required examining more of the matrix than was necessary to compute the transitive closure. Then during a large burst of activity, the matrix would not have enough capacity for the number of domains in the system. The size of

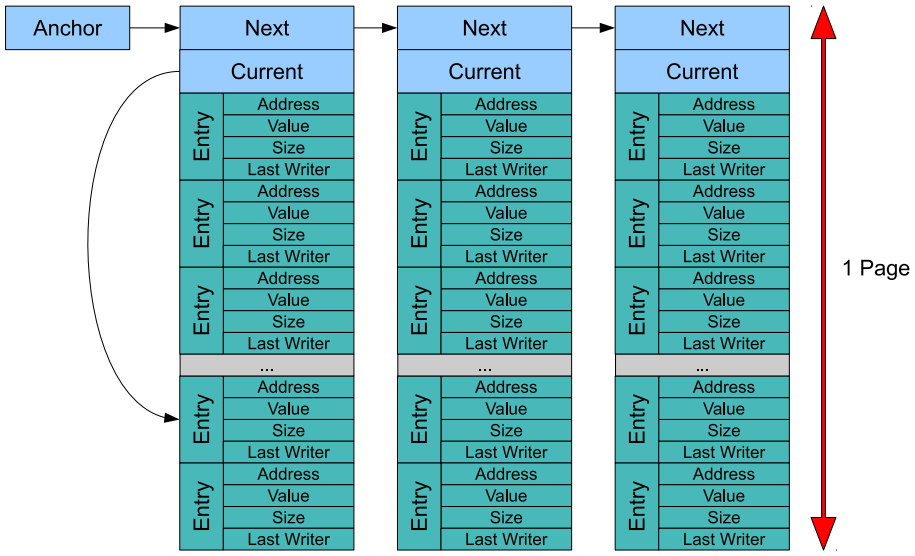

Figure 4.3: Paged log structure

the matrix is practically limited by the exponential growth in memory usage as the number of possible active domains increases.

4.4 Logging

Recovery Domains has several modes of logging. At the lowest level, memory writes are logged. This is the basis of rollback for a recovery domain. To increase flexibility and reduce inter-domain dependencies, operations may also be logged at the semantic level. Finally operations may be logged to be executed at the committing of a domain.

All logs use a list of pages as illustrated in figure 4.3. Log entries are stored in an array on a single, page-sized allocation. These pages are stored as a linked list. When a page is full, a new page is pushed on the linked list and entries are written to it until it is full. This provides very efficient

storage in space and time. The overhead per entry is two words amortized over the number of entries in a page. The frequency of the allocations is once per number of entries in a page. Further, single hardware-sized pages are used so that if all internal memory is exhausted and the recovery system must acquire memory from the operating system, the allocations will not require contiguous virtual or physical pages.

Semantic logging occurs when an operation is declared as a domain with a semantic inverse. In such cases, upon the successful completion of the operation, the success is logged with sufficient information to perform the semantic inverse operation. This structure allows two things. First it allows optimized handling of independent subsystems for which a clear higher-level logical structure is known, such as `malloc` (the inverse being `free`). Second it allows inverses for code which modifies the state of devices. Writes to memory-mapped devices or other device control channels cause state changes that are not reversible by writing the old values to the device since writes cause complex state changes. Semantic inverses allow operations to be given inverses which encode the knowledge of the controlled device. In the case where there are no dependencies on other executing domains, domains with semantic inverse need not have their log retained by the calling context since the reversal of that domain is logged semantically. During the execution of a domain with a semantic inverse, memory logging occurs so that in the case of a recovery event within the domain, the domain can be reversed. However, once the domain completes, the parent need only record the successful completion and the necessary information to call the inverse after a recovery event.

To allow resource managing subsystems such as `malloc` and `free` to not often interfere with each other, operations may be logged to be deferred until

the successful completion of the calling request. This prevents, for example, memory from being allocated and freed by one domain, then allocated by another domain. In such a case unnecessary dependencies are formed by due to the artifact of memory being reused by the allocator after a free. These dependencies should not exist. By deferring operations, many false dependencies can be avoided.

4.4.1 Alternative Log Structures

We tried several alternative structures for storing the log. Chief among these were a pure linked list and an array. A linked list, even with a very efficient allocator was less efficient than the hybrid structure we used. Much of that inefficiency comes not from the allocator, but from the overhead in storing the pointer to the next element. This pointer increased the size of each log entry and the store added expense to a very performance-critical portion of code (the store path). Since there is no constraint on the number of memory operations a recovery domain may execute, there was no fixed size array that would suffice.

4.5 Committing Recovery Domains

A commit of a domain comes when a domain meets the conditions specified in chapter 3.6, namely it and all domains it is dependent on are in the done state. The order in which domains are committed in the commit protocol does not matter.

When a domain is committed, as illustrated in figure 4.4, several parts of the domain structure, per-thread data, and global structures are updated. The commit updates the current recovery domain for the thread with the

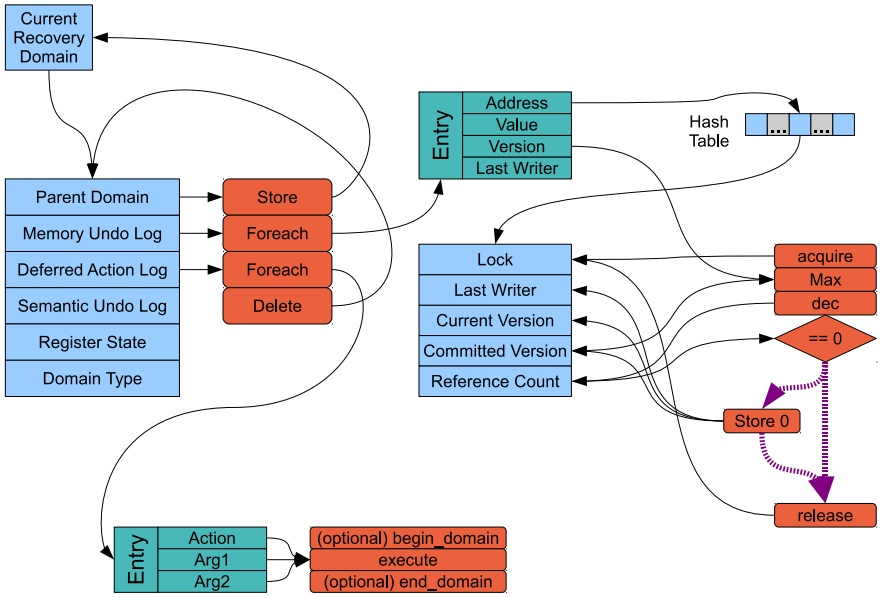

Figure 4.4: Committing a domain

one previously running prior to the start of this domain. If a domain was in the done state but not active, which is the case if it could not commit due to a dependency, this is omitted. For the entire commit procedure on all domains involved, a global lock, which protects recovery domain creation and destruction, is held.

The commit procedure walks the undo log, committing each write. This is done by taking the written location and hashing the pointer to acquire the meta-data for that location. The lock for the location is acquired and the version in the undo log is compared to the last committed version in the log. The meta-data is written with the maximum of the two version numbers. If the last writer was the committing domain, the last writer field is cleared. The lock is then released.

Committing a location can be done in arbitrary version order. Dependency edges ensure that true data-dependencies are maintained between read-

ers and writers. Committing a version does not change the current version of a location. This is because a later, uncommitted writer may still exist which will have incremented the version number. However, since the write we are committing overwrote existing data and all subsequent readers will depend on the new version or a later version, any reader of old versions already has the necessary dependencies on the writers. An older version may be rolled back, but this will be a no-op since the effect of the rolled back domain is already masked by a later writer.

After memory is committed, the deferred action log is walked, executing each deferred action. These actions are those operations which, due to being independent, must be delayed until it is known they should execute. The most common example is a resource release or deallocation.

4.6 Rolling Back Recovery Domains

A rollback is triggered when some run-time mechanism, either a part of the system or code inserted by an external tool, detects a potentially fatal error. During normal execution, the system must maintain an undo log for each speculatively written location, and a domain dependency graph for each domain. In addition, each domain maintains a list of deferred actions, such as resource frees. The recovery system rolls back a faulty domain (or an enclosing parent domain) and any other domains that are dependent on it. To roll back a domain, the recovery system restores any memory perturbations and undoes resources allocated for all dependent domains from the logs for those domains. This is illustrated in figure 4.5.

During rollback, all affected threads are halted at a known state, and one process walks the dependence graph of the faulting domain, rolling back any

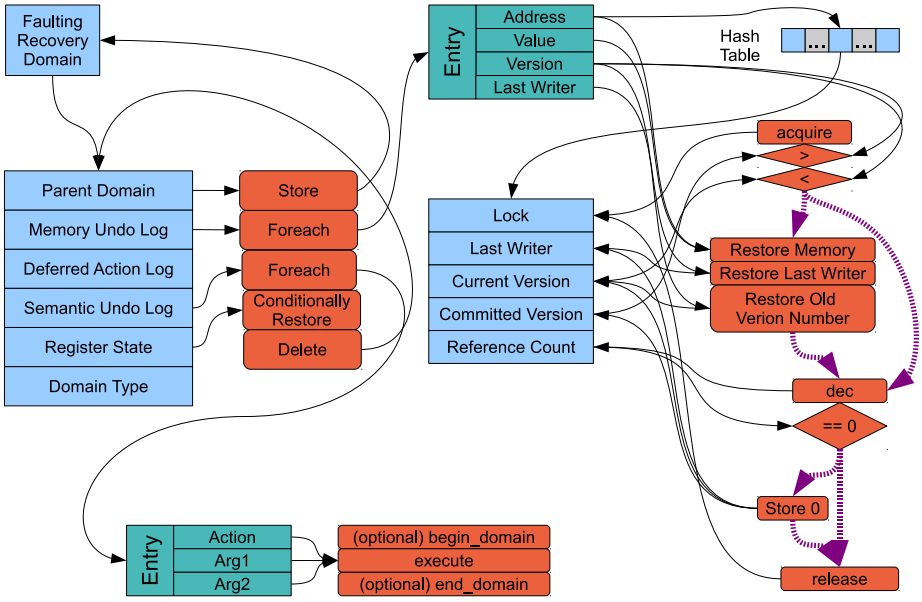

Figure 4.5: Rolling back a domain

dependent domain. Because versioning is kept for writes as part of the metadata, domains can be rolled back in any order and the rollback code ensures that the original value prior to all the rolled back speculative writes is restored. Reversible operations encountered in the log are reversed with their inverse function; by definition of such operations, the order in which these inverses are applied is irrelevant. Register state is restored to the point of domain entry for each active rolled-back domain, with the only change appearing as though the domain entry instruction returned an error (this is essentially setjmp and longjmp, but potentially operating on threads besides the current one).

4.6.1 Error Virtualization

One key design goal is to use existing error return paths to preserve failure semantics, expedite recovery, and simplify the implementation. Complex server systems like an OS have extensive error checking, with corresponding error return paths, for *anticipated* errors. In particular, many internal functions in such a system are programmed to handle error conditions when they occur, either by retrying an operation, or returning appropriate error information to their callers, which then continue the process. This process creates the error return path, which often propagates all the way back to the external client. The system specifies a semantic for error handling that clients (e.g., system calls) must use to deal with errors cleanly. By leveraging these existing error return paths, we can incorporate comprehensive error recovery from nearly all of a given system, such as an operating system, while *requiring relatively few changes to the code base*.

Each domain specifies an integer error return code, which is returned to its parent domain when an internal error is detected. The parent domain can then handle this error code as desired by the programmer. If most domains start at existing error checking points, then little further effort should be needed to perform recovery and error return from unexpected errors.

Within a kernel, asynchronous requests are similar to a system call. A domain places a request on some structure. At some point another thread services that request. These execution paths have defined ways for the worker to return errors to the requester. Thus this idiom is implemented as an independent domain for the worker which, if it fails, returns an error code though the same channel by which it would normally return an error code to the requester.

In the case of a kernel, when an error propagates up to the application, appearing as a failed system call, the kernel can choose to return a suitable error code for that system call. For applications that don't care about the error code, including those that don't wish to recover, this choice is unimportant. For other applications, if this is a pre-existing error code, no changes are needed to applications that use that system call. If it is a newly defined error code, applications that wish to recover would need to handle this code appropriately. If the error is persistent, i.e., retrying the same system call causes the error to repeat, then the application may have to compensate in some application-specific way or may simply be forced to die. In any of these cases, the kernel and other applications should not be affected.

4.7 Compiler Passes

The recovery compiler consists of a series of passes to transform the annotated kernel source into recovery-instrumented machine code. The first pass replaces all memory operations with calls to runtime routines that will perform the operation, log the operation, and track dependencies caused by it. The second pass interprets programmer-supplied annotations and maps these annotations to sequences of low level annotations.

The compiler components are implemented in the LLVM compiler framework [47]. Kernels are compiled through the normal C front-end and machine code is generated by the normal back-end; all transformations happen on optimized intermediate representation (IR) code. Source annotations exist in the original source code and are passed through the front-end unchanged.

The first pass replaces all memory operations with calls to runtime routines that will perform the operation, log the operation, and track dependen-

cies caused by it. This was explained previously in section 4.2.2.

The second pass interprets programmer-supplied annotations and maps these to sequences of lower-level annotations, namely `domain_begin` and `domain_end`, to be dealt with by the third pass. This pass allows the programmer to succinctly annotate the kernel (in this implementation, as a file specifying the functions to be treated as domains, their inverses if any, and their type). The programmer can directly use the low level annotations if necessary or convenient.

```
void* kmalloc(size_t size, int type) {
  //begin a logged, reversible domain
  char* buf = rec_domain_begin(log=1, reversible=1);
  //record current register state
  int iserror = rec_setjmp(buf);
  if (!iserror) {
    // kmalloc_orig is the original kmalloc, renamed
    result = kmalloc_orig(size, type);
    // end the kmalloc domain
    rec_domain_end();
    // log inverse function
    rec_log_undo(kfree, result);
  } else {
    // Error landing pad:
    // The user specified null as the error return
    // value for the kmalloc domain. At this point, no
    // kernel state has been restored by the runtime
    // as though domain_kmalloc had never been called.
    result = NULL;
  }
  return result;
}
```

Figure 4.6: Example of the recovery domain transformation on `kmalloc`. The new functions implemented in the runtime are underlined. `kmalloc_orig` contains the original code for kmalloc.

The third pass transforms low level annotations into operations to setup and tear down recovery domains, and uses `setjmp` to create a landing pad

for control flow after a domain aborts. Figure 4.7 shows the result of the third pass on the `kmalloc` function. `kmalloc` becomes a wrapper that hides the domain management from the callers. It sets up a domain with a call to `rec_domain_begin` which returns a buffer used for `setjmp`. Normally, the original code for `kmalloc` is executed. The domain is then committed since `kmalloc` is reversible; the inverse is logged in case the parent aborts; and the result value is returned to the caller. If, however, the `kmalloc` domain aborts, the error return code specified by the annotations, namely NULL, is returned to the caller. The runtime manages rolling back state and terminating the faulting domain before passing control to the landing pad.

4.8 Runtime

The runtime consists of 4 major components: recovery domain management, logging, memory data-flow detection, and rollback infrastructure. The runtime is SMP-safe and totals 1867 lines of C++ code (including all assertions and debugging code) and 30 lines of assembly. Compiled, it results in 36kB of program text, though for performance, it is linked to the kernel using LLVM to perform inter-procedural optimization.

Recovery domain management implements the domain stack as a linked list with the active domain (top of stack) at the front. It also tracks the type of domain and dictates whether operations should be logged or not, as well as managing the rollback or commit of domains.

Logging is implemented as a linked list of 4kB pages, each of which can hold about 150 logging records. When a page fills up, a new one is linked in. When a child domain terminates, its log is either linked in directly or, if it was a reversible domain and committed, its inverse is recorded. If a domain

reverse is marked for delayed execution (such as a resource free), the delayed function and its arguments are recorded.

Recovery domain management implements several data structures including the domain stack, the logs of writes and inverse actions, and the domain dependency graph. The domain dependency graph is stored as an out-edge set in each domain. The completed but uncommitted domains are kept in a list. The dependency graph is divided into two regions, the active domains and the completed domains. Two properties are exploited to reduce the size of the out-edge sets while still maintaining the necessary transitive closure of the graph. First, only edges in the active set will form new dependence edges. Thus all new edges will point from the active set to the completed set. Second, only edges from the committed set to the active set matter for computing if a domain in the completed set can commit. Because of this, the out-edge sets for completed domains only include edges to active domains. When an active domain is moved to the completed set, all nodes with edges to it are updated to include its edges to active domains (thus maintaining the necessary transitive closure) and its out-edges are pruned to only contain edges to active domains. This optimization greatly reduces the number of edges that must be kept.

Dependencies due to memory reads and writes are tracked by memory versioning. Memory addresses are hashed after masking off the lower bits so regions are at least 64 bits (though in practice are several times larger). No collision avoidance is done, so regions with the same hash value artificially alias. For each entry, a sequence number of the last write, a sequence number of the last commit, a count of domains with references to the location, and a reference to the last writer is maintained. On a write, the sequence number is incremented and the writer records the old value of the memory and the

sequence number of its write. Since a reference to that location is entered in a log, the count for the location is incremented. The writer is also recorded as the most recent writer of that region. On a read, the reader checks to determine if there is an uncommitted writer and, if so, adds an edge in the dependence graph to the writer.

The rollback mechanism simply walks the log in reverse order undoing operations and removing the current domain from the stack of domains. `longjmp` is used to return control to the landing pad created by the compiler. During rollback, other threads that are dependent on the current domain are also rolled back, directly modifying the saved state of the inactive thread (this requires Linux-specific knowledge; an OS-agnostic version would set a flag in the rolled back domain so on next execution, control would transfer to the recovery point).

The commit mechanism walks the log, updating memory regions to reflect the sequence number of the logged writes, if that update is greater than the sequence number of the last commit. Further, if any deferred actions are logged for the domain (and its children), they are executed (e.g. memory frees, reference count decrements).

4.9 Porting Linux

Two very different Linux kernels were ported to the recovery system. The first, which will be described here, is the SVA ported Linux 2.4.22 kernel as used in the SVA work [21]. The recovery system itself is completely independent of SVA – SVA is only used as an error detection mechanism. To demonstrate that and to gather experience porting another kernel, Linux 2.6.27 was also ported. Linux 2.6.27 has some fairly major structural differences. In the

first porting exercise, several interesting common cases were discovered that influenced the design of the recovery system. These include spin-locks, request queues, reference counting, and performance counters. As discussed below, extra porting effort went into these objects as it greatly reduced the number of dependencies between domains.

The starting point for a port to the recovery system is to identify entry points and allocators. For Linux, system call and interrupt entry points were annotated as independent recovery domains. The list of basic allocators (e.g. `kmalloc`, `kmem_cache_alloc`, `__alloc_pages`) were annotated as semantically reversible domains, and their inverses specified.

With this basic port, it became clear that spin-locks were a major cause of thread interference. Because atomic operations were modeled as reads and writes (with the writes logged only if the atomic operation succeeded), any two threads that accessed the same spin-lock (even if they did not contend for it) would form a read-after-write dependency. A basic spin-lock is binary, and is used for synchronization. What data-flow exists is really control flow masquerading as data flow. With these observations, successful spin-lock acquires and releases are modeled as write-only memory accesses with old values of '0' and '1' respectively. Trying to acquire a spin lock was not treated as a read operation. This preserves the property that a lock is released on a rollback while breaking unnecessary dependencies. Reader-Writer spin-locks were treated in a similar fashion.

Many counters used for performance statistics, e.g., the number of blocks written to a disk, are not used in any decision made by the kernel; they simply exist to be reported to user-space. These are not critical to the consistency of the kernel after (presumably rare) error recovery. Therefore counter increment and decrement functions were defined as unlogged, reversible domains.

On a rollback, the counter increment or decrement is undone, but other domains are not rolled back.

The kernel code implementing the `exit` system call has an internal function that does not return because the thread is terminated. Making this function (`do_exit`) an unlogged domain, however, is not attractive because it performs considerable work, including closing files, tearing down address spaces, and notifying other threads. Any of these operations could fail, in which case we would like to restore kernel state and return an error. To fix this mismatch, the kernel function `do_exit` was changed to return an `int` and made an independent domain. Code which called `do_exit` (not expecting it to return) was modified to go into an infinite loop calling `do_exit`. On a permanent fault, this could cause the thread essentially to leak and try to exit whenever it is scheduled, but this seems a better engineering tradeoff than allowing exit to perform large and complex state changes to global kernel data structures in an unrecoverable domain.

Correctly handling thread exit also required the introduction of the basic primitive `rec_thread_exit`. This primitive causes all domains on the stack to transition to the completed state. It is the last thing called before the kernel calls schedule from the thread, never to return to that thread again. Delayed commits and other resource frees for the defunct thread happen on the next attempt by any thread to commit.

Many objects in the kernel are reference counted. Handling these well reduces the number of unneeded dependencies. A reference counted object is acquired by calling a function on an existing object. This function acts very much like an allocator, in that it has an inverse, but serves to update the reference count of the object. A similar function exists for when an object is no longer needed. This function often handles the finalizing and freeing

Change	LOC
recovery hooks	9
counter conversions	28
moving functions out of headers	40
spin lock conversion	34
exit fixes	6
bootup fixes	3
fork	1
misc	11
Total	132

Table 4.3: Changes to Linux 2.4.22 by type and lines of code

the associated object when the reference count reaches zero. All the major structures in the VFS layer used this idiom. Thus the reference acquiring and releasing functions were declared as inverses of each other and acquisition functions as semantically invertible domains. Because of this, dependencies between threads that existed only because of a change to the reference count of an object were broken, without disturbing the garbage collection properties of the code. As a proof of concept, only the file-system was thoroughly ported in this way.

While many interesting cases were considered and ported (and several more primitives, such as queues, would benefit from special care), *no design changes were needed in porting the Linux kernel to use Recovery Domains*. Furthermore, only 132 lines of code were added or modified for the port of Linux 2.4.22 as broken down by type in Table 4.9.

The port of Linux 2.6.27 proceeded similarly, starting with the allocators, system calls, and interrupt handlers. The softirq handlers and the scheduler were also annotated. The hooks for runtime memory allocation were added. No major design changes were needed, even though several major kernel features, such as kernel preemption, were added in this kernel version. Furthermore, even fewer lines of code needed to be changed because (for ex-

ample) many macros had been moved out of headers and converted to inline functions. This port is operational, but never tuned to provide competitive performance. A new port of a non-SVA based Linux 2.6.27 is discussed in Chapter 5.

4.10 Discussion

Several limitations and special considerations regarding recovery domains are discussed here. Sources of deadlock introduced by Recovery Domains are discussed first. Ways to corrupt the runtime and prevent recovery are then discussed followed by a brief note on the output commit problem.

4.10.1 Avoiding Deadlock

The interaction between the process and thread management in the OS and the Recovery Domains commit protocol can be a subtle source of deadlock. Initially, to prevent corrupt state from leaking to user-space and to ensure that a request is never reported as successful if it still might be rolled back due to a dependency, system calls did not return until they committed.

Some system calls, however, block for reasons which necessarily cause interference (and hence dependencies) with other system calls. While an IO operation can block, it will eventually unblock due to the structure of the kernel. A system-call such as `sys_wait` will necessarily, at least in the Linux implementation, read and write memory from the control block of the very threads on which it is waiting. The state written happens to be read as a matter of course in several other system calls the waited-upon thread can issue. This causes a circular dependency: a kernel level dependency from the waiting thread to the thread being waited on and a recovery domain level dependency

from the waited upon thread trying to commit and return to user-space and the thread which will not commit until the wait condition is satisfied. The recovery system does not know about this cycle.

To avoid this, it is helpful to note that only one system call can get into this situation. We therefore treated it specially: having it transition out of a normal domain into an unlogged domain prior to blocking. This effectively causes it to commit early and not block any other request.

We further only make a good faith effort to commit a system call prior to returning to user-space. If yielding the processor to other requests does not, after a finite time, allow the system call to commit, it returns to user-space uncommitted and it will commit once its dependencies are met.

4.10.2 Corrupting the Recovery System

The recovery domain does not inherently protect itself from corruption by the kernel. This corruption can come from several sources. It is worth noting that many basic memory fault detectors will catch and prevent corruption. Faults can come from stray pointers, stray control flow, stray DMA operations, and incorrect page re-mapping.

Of these, stray pointers are the most likely. Many bugs are caused by incorrect pointer arithmetic or buffer overflows. Such an error could cause writes to memory used by the recovery system, thus corrupting it. Range checks could be employed in the runtime, since all writes are routed through the runtime, to prevent this, but it was felt that memory faults would be the first and most common fault detector deployed. Catching memory faults is left to the detector, but corruption to the runtime could be caught by the runtime.

Stray control flow is mostly solvable by preventing corruption to memory. However, since thread state is stored by the kernel, control flow can be corrupted by inbound writes to the structure storing the register state for inactive threads. Corrupted control flow can execute instructions which are not instrumented and not prevented from writing to the recovery runtime. There is no obvious way prevent this potential for corruption of control state and only limited ways to prevent corruption to the runtime.

The only way to partially prevent corruption is to keep the runtime from being writable by the kernel. This can be done through the page tables. Random code executing in the kernel can undo any page table protection allowing corruption. Incorrect page mappings cannot be solved for this very reason.

The last source, DMA, can be partially prevented by using an input-output memory management unit (IOMMU). IOMMUs allow restricting access to system memory to devices. While correct programming of an IOMMU can prevent hardware devices from corrupting the recovery runtime, correct programming can be prevented by one of the other sources of corruption or by programmer error.

4.10.3 Output Commit Mitigation

Like any system with rollback, output commit is a problem. Values can escape to the outside world though hardware before rollback happens. These values cannot be rolled back; the values may cause hardware to execute non-reversible operation, such as sending a network packet.

The structure of many drivers in the kernel help mitigate, but do not alleviate, the output commit problem. Many subsystems perform most work in a context which does not interact with hardware. The passing of data to the hardware device is separate from the processing of requests which take

the data from applications. Such a structure allows the system to successfully commit most of the complex processing of requests without touching hardware. The controlling of hardware devices is done after the application request commits.

Although this structure found in many drivers allows most complex logic to be recoverable, it is serendipitous, but not required. The recovery system currently cannot prevent output commit.

4.11 Conclusion

The reference implementation of Recovery Domains provides effective recovery from faults while being understandable by operating system programmers. Performance is good for several benchmarks, though low for an IO intensive benchmark. The system provides an impressive recovery rate from random faults inspiring confidence that recovery domains are good approach to recovery from operating system faults.

The design achieves a simplicity which lends credibility to the implementation. Versioned memory and undo logging form the basis of the design. Domains are allowed to finish without committing and a commit protocol ensures domains commit when they can.

Chapter 5

Design and Implementation of Compiler and Runtime Analysis and Optimizations

The reference implementation of Recovery Domains given in the previous chapter does not use analysis of the operating system by the compiler to optimize the recovery system. In fact, it depends on the compiler only for instrumentation of memory operations and inserting domain setup and teardown. We observe that the runtime recovery burden can be reduced through static compiler analysis, dynamic, run-time analysis, and additional knowledge about the kernel's use of locks. This analysis allows the system to selectively not monitor some memory operations while reducing the logging and versioning requirements of some other operations.

The new optimizations are based around:

Dependence Graph: At runtime, the dependence graph is analyzed to see if it, or specific domains, are complete (have all possible out-edges formed). In this case, no monitoring of loads is needed.

Locked Accesses: At compile time, all loads and stores are analyzed to see if they are accessing an object which is protected by a lock. Locks and the objects they protect are identified by a new annotation. Locked accesses do not need locking or versioning on writes and no monitoring of loads.

Fresh Access: At compile time, all loads and stores are analyzed to see if they are accessing an object which has been allocated but is not yet visible to

other requests. Fresh accesses need no monitoring of reads or writes.

This chapter describes these optimizations as well as the runtime changes necessary to support them. A new port of Linux 2.6.27 to Recovery Domains is used to demonstrate the effectiveness of the optimizations.

5.1 Dynamic Analysis of Dependence Graph

Every read is normally monitored to see if it forms a read-after-write dependence with an uncommitted domain. The dependence graph is not between individual locations, rather it is between domains. It is therefore unnecessary to monitor a read if the dependence graph cannot have an edge added to it by the read. We take advantage of this property both globally and locally.

To simplify dynamic analysis and to improve overhead in general, we store two dependency graphs. The first graph is the full dependency graph between all domains. The second graph is that graph projected onto just the domains that form the roots of requests. These roots are exactly the independent domains. Each independent request may have many generations of descendants in the form of normal domains.

Computing whether a domain may commit requires the transitive closure of the subset of the dependency graph reachable from the domain which is trying to commit. Computing a transitive closure is expensive and storing all edges for the closed graph costs memory and time. The time may be paid more than once as the dependence graph will be traversed again if a domain is not able to immediately commit. To minimize edges, we treat the sub-graph of all independent domains specially. When a domain forms a dependency, this edge is mirrored in the independent domain subgraph. That is, $A \rightarrow B$ forms $A.root \rightarrow B.root$. Since all child domains have an implicit dependency

on their parent, it is not necessary form dependencies on finished child domains which are waiting on their parent. We can directly form a dependence on the active parent. This prevents many edges from being formed to inactive domains.

Dynamically, the runtime considers the completeness of the dependence graph. If the sub-graph of the dependence graph of independent (root) domains is complete, then no operation can add a dependence edge. The runtime detects this condition by comparing edge counts to the number of independent domains. When this comparison shows that all possible edges have been formed, a global flag is set. The compiler-generated load instrumentation checks this flag and if it is set, performs the load in place rather than calling into the runtime. The flag is cleared when a new domain is created, as there is now a new target for dependencies. This provides substantial savings by changing all reads (which account for roughly 66% of the memory operations) in the common case from a call to the runtime to a check of a global flag. Examples of instrumented code are in section 5.4.

While a global flag is simple, some work can be avoided by reasoning about an individual domain's dependencies. When a single domain has all possible dependencies, it does not need to read version information to add dependence edges even if other domains still do.

We take advantage of the mirroring of edges into the independent-domain subgraph by ignoring child domains when checking if a domain can commit. Since all dependencies are projected onto the independent-domain subgraph, computing the transitive closure and checking that all dependencies can commit is performed on the subgraph rather than on the entire graph.

```
asmlinkage long sys_munmap(unsigned long addr,
                           size_t len)
{
  int ret;
  struct mm_struct *mm = current->mm;

  /* Annotation marking this function as the start
   * of a recovery domain
   */
  REC_SYSCALL();

  profile_munmap(addr);

  down_write(&mm->mmap_sem, mm);
  /* down_write calls:
   * rec_acquire(&mm->mmap_sem, mm,
   *             sizeof(mm_struct));
   */
  ret = do_munmap(mm, addr, len);
  up_write(&mm->mmap_sem, mm);
  /* up_write calls:
   * rec_release(&mm->mmap_sem, mm,
   *             sizeof(mm_struct));
   */
  return ret;
}
```

Figure 5.1: Example simple lock annotation

5.2 Locked Memory

We add one new annotation to mark memory covered by a read-writer or exclusive lock. This annotation specifies four things: the lock location, the lock acquisition point, the lock release point, and the region of memory covered by the lock. An example use of the annotation is shown in Figure 5.1 in which the `mm_struct` for the current process is locked for exclusive write. The lock is annotated with information about which portion of memory it covers, in this case all of the `mm_struct`. The end of the locked region is an-

notated. The annotations make no assumptions about the use of the memory after the release of the lock.

Not shown in the example is that an annotation can mark the objects pointed to by fields as also being covered under the lock, including linked structures. This allows objects with linked lists of trees, for example, which are protected by the lock, to be considered locked with the object. This is a fairly common occurrence and turns out to be very important in practice. For example, one of the most exercised functions in the benchmark postmark, `find_vma`, traverses a red-black tree which is referenced by a field in a structure and covered by a lock in that structure.

The lock annotation is optional and need not be applied to every type of object. The recovery system operates correctly without any lock annotation, but the annotation provides opportunity for the compiler to reduce overhead.

5.2.1 Optimizations for Locked Memory

Locked objects, which are very common in a kernel, have several useful properties with respect to logging and versioning. First and most obviously, if an object is exclusively locked for writing, then the meta-data handling the versioning of the memory locations in that object do not have to be locked for updating. Objects locked for shared-reading, likewise, do not require version meta-data locking when performing a read as the meta-data is not changing. Dependence graphs still must be updated, but this is done in a request-local manner.

More interestingly, a lock provides a single memory location that can be used as a proxy for dependency tracking for the objects under lock. If all accesses to lock-protected regions of an object are performed after the lock is acquired, then any dependence edge between requests which could form

will have already formed by virtue of acquiring the lock. A write lock will update the lock location, marking itself as the last writer. Since a writer must first read the lock to acquire it, it will form a read-after-write dependency on the previous writer. Any subsequent reader will first check the lock and acquire it for reading. This operation is a read-after-write dependence and will form an edge in the dependence graph. Writers form a dependency chain due to the serializing nature of the lock and readers will transitively depend on all uncommitted writers simply by depending on the last writer in the chain. Therefore all reads of a locked object can be unmonitored. Since writes are already serialized, writes to the object need only perform logging, not versioning. The lock provides a proxy version for the entire object or memory region.

Locked Memory Analysis

The analysis to find locked accesses is shown in Figure 5.2. The locked property is acquired for objects at the annotation and dropped at the release annotation. Operations on the object are locked if they meet three conditions: there are no paths to them which do not go though a lock annotation, after the lock annotation there are no unlocks, and there are no joins with unlocked objects. Initially, calls are considered unlocking operations. Calls are analyzed to see if they preserve the locking of their arguments. Calls which do preserve an argument's locking status are not considered unlock operations for that argument. Additionally, we determine if an argument is always locked. Such arguments may be optimized as locked objects.

This analysis is iterated until a fixed point is reached. Finding function arguments which are lock-preserving lengthens the periods in which an object is locked, potentially causing a call previously considered as receiving an

```
function unlocks(Op, obj) {
  case Op is a call:
    if obj passed to arg not in SafeArgs
      return true;
  case Op is an unlock annotation:
    return true;
  default:
    return false;
}

do {
  /* trace all locked objects */
  foreach(obj in {locked annotations})
    foreach use in uses(obj)
      if (no unlocks on path obj to use)
        LockedOperations.insert(use);

  /* trace through function arguments to see
     if they are lock preserving */
  foreach(arg in {function arguments})
    if (forall use in uses(obj):
        !unlocks(use, obj))
      safeArgs.insert(arg);

  /* see if calls are always locked */
  foreach(arg in safeArgs)
    if (at each call to function for arg:
        actual arg is locked)
      {locked annotations}.insert(arg);
} while (any result changed);
```

Figure 5.2: Simplified lock analysis.

unlocked argument to be show to always receive a locked object. More call arguments considered lock-preserving can lengthen the apparent lifetime of locks, causing more arguments and operations to be seen as locked.

5.2.2 Misuse of the Lock Annotation

Locks are not used perfectly and we must not expect lock annotations to be used perfectly either. We therefore consider what happens when an annotation is missing from a normally-annotated lock and when a normally-locked object is accessed without the lock. The first case is fairly easy to prevent when porting by folding annotation into the common locking code. An example of this is seen in Figure 5.1 where the locking annotation is put in down_write rather than at each use of the lock. The second case implies existing race conditions in the kernel and finding these bugs is not our goal (though a race detector is welcome to use Recovery Domains to recover if it detects a race). Though we expect this to cause kernel bugs, we want to have an understanding of what happens during recovery in such a case.

Due to the non-invasiveness of Recovery Domains when there is no fault, the behavior of the kernel does not change in the presence of either of these errors. Absent a fault, Recovery Domains will do no more than changing the interleaving of accesses in a lock-consistent manner.

When an object is properly locked but the lock is not annotated there is no loss of recoverability. As with annotation, the lock will still serve as a proxy for dependencies that may be formed. Unnecessary work will be done checking and updating versioning, but since mutual exclusion is ensured by the lock, recovery can reconstruct the order of operations from the undo logs even though the locked writes didn't record a version number. Operations in the unannotated path will be individually versioned, but all writes will be

between the initial version of the lock and the write that clears it. Any other reader or writer will log the version of the lock when they lock the object prior to writes or reads. This allows the recovery system to reconstruct a total ordering to writes to a specific location. The only harm is a missed opportunity to reduce overhead.

The kernel may have a race condition bug in which an object is accessed without acquiring the locks normally required to do so. The recovery system cannot ensure correct recovery when a race condition involves one domain with an annotated, locked access and a domain with an unlocked access (note that without lock annotations, correct recovery would happen).

5.3 Fresh Memory

An object is fresh when it is allocated and hasn't yet been exposed to other requests. Exposure happens when a pointer to it is stored in a location readable by another domain. This is known as escaping. As long as the pointer isn't stored where another domain can read it, the object hasn't escaped. Until the object escapes, we can optimize operations on it knowing that no other domain will be reading or writing the locations.

Objects are often initialized before escaping. These accesses which initialize the object can be optimized to not do any logging or versioning. Since the recovery action for an allocation is a deallocation, the contents do not matter until another request can read them. For an object to escape, a pointer to it must be written to memory reachable by another request. This write which causes the escape of the object also effectively versions the object relative to other domains. The escaping write will set the version on the memory location which holds the pointer to the allocated object and mark the writer as

the allocating domain. For the object to actually escape, another domain must read that location. This read will form a dependency between domains. This is the same dependency edge which would form if each location in the object had been versioned prior to the object's escape, thus alleviating the need to track the initial write to the fresh object.

Initially we mark objects from allocators as being fresh. We then trace through the control flow of the code, finding locations where an escape happens. This can be either because the address was written into another object or global memory or because it is passed to a function. If along all paths to a particular access, the pointer don't escape then the access is considered fresh.

We then look at functions which have arguments which are fresh in all contexts in which the function is called. We repeat the previous analysis on the argument, since it is fresh. If the argument remains fresh throughout the function, we can reconsider all calls to the function as not killing the fresh property of the object. This allows us to repeat the previous analysis no longer considering calls to this function as causing the object to escape; i.e. a fixed point is reached.

5.4 Compiler Analysis

The compiler passes start by finding the fresh memory and locked memory annotations on pointers and performing an inter-procedural, flow-sensitive propagation of these properties on the heap objects. Standard analysis techniques are used to identify constant memory objects. These properties are queried by an instrumentation pass for each load and store and atomic operation to check for several optimizations. Locked, fresh, or constant reads need not have any instrumentation. Normal reads have two paths generated

for them, one which calls into the runtime and one which directly executes the load, as described in Section 5.1. The optimization in the Section 5.1 in which a load is fast-pathed based on the dependence graph for that request (rather than the global state of the dependence graph) is performed inside the runtime. Locked writes are simply logged and not versioned. Fresh writes are unlogged and not versioned.

Since we propagate information inter-procedurally, we can perform these optimizations in functions which are passed heap objects. This, in practice, helps in common cases of initialization functions. Objects, once constructed, are often passed to functions which initialize various parts of them. Some may clone an existing object, some may initialize a data structure (such as a linked list) embedded in the object, etc. Likewise, many functions have a precondition that the object they receive is locked. We are able to observe that this is the case from propagating the locking analysis and take advantage of this.

Figures 5.3 and 5.4 show the transformation on LLVM intermediate representation as the equivalent transformation on C of a load and store, as well as a sketch of the runtime.

5.5 Runtime Support

For clarity, we describe the full runtime though it is very similar in structure to that presented in Chapter 4. The runtime consists of three main components: undo logs, the dependence graph, and the memory versioning. Memory is versioned on a word sized boundaries. Meta-data is looked up in a 2^{16} entry hash-table using a direct-map hash function. Meta-data entries in the table have a lock, a last writer, and a current version number. The lock is im-

```
/* Original operation */
/* i32 X = *Y */
/* RD is the current recovery domain */

if (fastLoads)
  X = *Y;
else
  X = rec_log_load_i32(Y, RD);

/* simplified runtime overview */
i32 rec_log_load_i32(Y, RD) {
  Meta = get_meta_for_ptr(Y);
  Meta->lock();
  i32 retval = *Y;
  RD->depend_on(Meta->oldwriter);
  Meta->unlock();
  return retval;
}
```

Figure 5.3: Example load transform

plemented by stealing a bit from the last writer pointer. The last committed version number is not maintained for entries as it is implicit in the undo logs. This trades efficiency in the common case for extra overhead when recovering from a fault (which is rare).

Undo logs store information about writes. They store the old value, the size of the value, the location, and the version number of that write. Bit-stealing schemes are used to encode the size in the other values, providing a good space savings. Logs are stored as a singly-linked list of pages which contain arrays of log entries. The page size is chosen to be the same as the hardware page size.

The dependence graph is a simple graph embedded in each domain. Roots of the graph (independent domains) are stored as a singly linked list with the pointers embedded in each node. Additionally, a list of root domains is kept. Domains are reference counted to prevent the need to scan dependencies of

```
/* Original operation */
/* *Y = Val */
/* RD is the current recovery domain */

rec_log_store_i32(Val, Y, RD);
/* or if Y is locked: */
rec_log_store_fastpath_i32(Val, Y, RD);

/* simplified runtime overview */
void rec_log_store_i32(Val, Y, RD) {
  Meta = get_meta_for_ptr(Y);
  Meta->lock();
  i32 oldval = *Y;
  *Y = Val;
  Version = ++Meta->version;
  RD->log_write(oldval, y, sizeof(i32), Version);
  Meta->unlock();
}

void rec_log_store_fastpath_i32(Val, Y, RD) {
  Meta = get_meta_for_ptr(Y);
  i32 oldval = *Y;
  *Y = Val;
  RD->log_write(oldval, y, sizeof(i32), 0);
}
```

Figure 5.4: Example store transform

all domains or store an inverse graph to determine the lifetime of a domain structure. Committed domains may still be a dependence edge for a domain which has not committed (dependencies are directional) and we integrate the clearing of committed domains into other update actions on the edges of a node.

To allow the recovery system to operate, the OS must provide a unique identifier for each thread. We implement this as a pointer stored at the bottom of each kernel stack which holds the active domain for that stack. The contents of this pointer are opaque to the OS and managed completely by the runtime.

5.6 Porting Linux 2.6.27

We annotate Linux 2.6.27 with 360 recovery domains. These cover all interrupts, system calls, allocators, and some other functions. We chose to annotate the timer interrupt's updating of the system time as an unlogged domain since this domain interfered with almost all other domains. These annotations also cover work queue dispatch routines, running each item in the work queue as a separate domain.

We annotated 104 locking locations covering mainly the `mm_struct`. This structure is used and locked extensively in the file-system and virtual memory subsystems. While there are other structures that could be annotated, this lock is the most performance-critical in the benchmarks.

Several inline assembly operations, such as atomic instructions and bit twiddling, were transformed into C code or gcc intrinsics. These are represented in LLVM and allow us to transform them into the appropriate runtime call. Inline assembly code had to, in this way, be mostly eradicated from

the kernel so as to not miss any memory writes. Since we do not guarantee consistency of user applications, we could leave the assembly routines which copy into and out of user-space alone.

Some inter-procedural optimizations had to be disabled in LLVM when processing the whole-kernel byte-code. Per-directory byte-code was fully inter-procedurally optimized in all directories but `arch/x86/kernel`. Other minor changes went into the kernel to reduce its dependency on gcc behavior.

5.7 Conclusion

Static and dynamic analysis provide a strong basis for reducing the amount of tracking necessary for Recovery Domains. Small amounts of additional annotation expose the structure of parts of the kernel to the recovery compiler allowing several new optimizations to be applied. Keeping in the spirit of being able to incrementally port a kernel to Recovery Domains, these optimizations can be applied incrementally also.

Chapter 6

Results

We implemented the systems described in Chapter 4 and Chapter 5 and evaluated them on a variety of workloads. We considered three metrics during evaluation:

Coverage: The theoretical coverage of the recovery technique in terms of fraction of execution covered by the technique.

Survivability: The ability to recover from injected faults in the covered portions of code.

Performance: The overhead incurred during normal (i.e., fault-free) execution of the kernel.

We did not directly measure the recovery time, which would give a measure of availability under potential denial-of-service attacks. We have observed that recovery times are extremely short, far shorter than typical times for a partial or complete reboot of the system.

Two implementations of Recovery Domains exist, one for the system described in Chapter 4 and one for the system described in Chapter 5. For the first implementation of Recovery Domains, we use the Linux 2.4.22 kernel used in the previous memory safety work in SVA [21] and a port of Linux 2.6.27 to SVA. These ports were limited to a few benchmarks due to bugs in the SVA runtime; all benchmarks that would run on the kernel without Recovery Domains ran on the kernel with Recovery Domains. For the second

implementation of Recovery Domains with the optimizations in Chapter 5, we use a native (non-SVA) Linux 2.6.27 kernel. We replace much of the inline assembly code in this kernel with compiler intrinsics which replicate the behavior. This kernel runs all benchmarks we tried without error.

6.1 Methodology

We select a variety of workloads on which to benchmark. These are:

postmark: a mail server benchmark with 1000 simultaneous files and 500000 transactions

bzip2: compressing a 17MB wav file at default options

gcc: compiling liblame

povray: rendering the install test scene

scp: transferring five 17MB wav files

scp -C: transferring five 17MB wav files with compression

apache: transferring an empty HTML file 100000 times

apache (k): same as apache but reusing a single TCP connection

postgresql: PostgreSQL running pgbench to simulate a TPC-B [76] like workload

We ran each benchmark at least six times, after an initial warm-up run, and looked at cumulative result of those consecutive runs. For network benchmarks we measured a maximum standard deviation of less than 1% in runtimes and considerably lower for other workloads. Overhead was calculated

against an equivalently-compiled kernel without recovery instrumentation. This kernel is source-identical to the recovery kernel and compiled with identical optimizations. Coverage was measured by using a modified runtime which tracks whether a memory operation was performed in a logged (recoverable) or unlogged domain. We instrument all memory operations and classify them as reads, writes, or atomic operations.

We test on an Intel Core i7-860 in KVM, Linux's in-kernel hypervisor, with frequency scaling and hyper-threading disabled. The client for network tests is connected to the test machine with one 1Gb Ethernet switch. The kernel network and block devices are the virtio devices provided by KVM. To validate the results using KVM, we ran the benchmarks directly on native hardware (an Athlon MP). The results we obtained were similar, showing that the results on KVM are representative of real hardware.

6.2 Survivability

To test how often a fault was unrecoverable (within the covered fraction of code measured above), we inserted potential fault-injecting code in every basic block of the kernel. During kernel execution, this code triggered a fault (that would normally be fatal) in a randomly chosen basic block every $300,000 + rand(0...300,000)$ basic blocks (where rand was recalculated after a fault). These faults were only injected during logged intervals to focus on the theoretically covered fraction of code. Faults were repeatedly injected until the kernel crashed, deadlocked, or otherwise obviously failed. The above injection rate corresponded to roughly about four faults every second of kernel execution. At this rate, many non-trivial kernel operations would take a fatal fault without Recovery Domains.

For the reference system, we repeated this experiment five times, each running a workload that included booting up the kernel, logging in as root and beginning to run the postmark application. Over these five runs, the kernel survived an average 35.4 faults (range: 17 to 72) before crashing. This means that on average, the kernel survived over 97.2% of faults in the covered portion of the execution.

For the optimized system, we performed a similar, but more targeted, experiment. Faults were injected as before, but only after a target workload was started. This tests a more realistic workload than booting the system and running postmark. Apache was the first workload. We performed 16 fault injection experiments on apache. The kernel survived an average 21 consecutive faults (range: 2 to 98). Postgresql was the second workload. We performed 6 fault injection experiments on postgresql. The kernel survived an average of 17 consecutive faults (range: 3 to 35).

The optimized system has fewer average successful recoveries than the reference system. However, these tests were run on more varied and more realistic workloads. Further, the runtime used was newer and less well tested. We examined which kernel domains encountered the final, fatal fault in each of the runs, but, perhaps surprisingly, did not notice any clear pattern there. It is difficult to isolate exactly what kernel state corruption caused the fault to be fatal in each experiment. We observed that the final, non-survived fault occurred in the kernel entrypoints `sys_open`, `sys_read`, `do_softirq`, `sys_mprotect`, `sys_sendfile`, and `sys_brk`. The entrypoint `do_softirq` was the most common. The frequency of crashes in these entrypoints corrosponds roughly with the percentage of time spent by the kernel in these requests.

Benchmark	Reference Implementation		Optimized Implementation	
	#Mem Ops	Coverage	#Mem Ops	Coverage
find	47M	67%	-	-
postmark	70B	96%	47.6B	97%
gcc	130M	31%	196.4M	96%
bzip2	63M	34%	88.3M	80%
povray	-	-	11.8M	71%
scp	-	-	682.6M	97%
scp -C	-	-	570.6M	96%
apache (k)	-	-	797.0M	69%
apache	-	-	2.5B	87%

Table 6.1: Percent of Dynamic Memory Operations By Domain Type

We suspect, but have not verified, that recovery mechanism failures were due to conditions in which recovery mechanisms could not recover, due to an unlogged domain, for example, or due to bugs in the implementation of the recovery system. Distinguishing these appears non-trivial due to the difficulty in debugging kernel code and the corruption of state (full-system simulators with deterministic replay have been the most useful in debugging these failures, but very slow). Several crashes, we suspect, are due to not completely handling all memory-like state. Page-table base-pointers and interrupt flags are examples of state which could be easily tracked by the runtime, but currently are not.

Overall, the key observation we make from this experiment is that the optimized recovery kernel is able to survive a large number of faults before failing. Together, the experimental results show that Recovery Domains are now efficient enough to be practical for real-world use, while still remaining highly effective at recovering from faults.

6.3 Coverage

An important metric in evaluating the recovery mechanism is how often the kernel is executing in code regions that are recoverable. Since the implementation "trusts" the scheduler and small portions of other codes as annotated by the programmer with unlogged domains, this will be less than the entire execution of the kernel.

As a proxy for execution time, we measured the number of memory operations (loads and stores) performed in each type of domain. we used several benchmarks: `postmark` (which simulates a mail server), `find`, `gcc`, `bzip2`, and a Linux kernel compile. The last two were not run on the SVA 2.4 kernel due to instability inherited from SVA. `Postmark` was run for 500K file transactions, `find` searching a tree of 2537 directories and 37822 files totaling 960MB for a specific filename, `bzip2` compressing a 17MB file, and `gcc` compiling liblame (a multimedia library) at -O3. Compiling `liblame` involved compiling 20 C files (totaling about 28 KLOC) and linking them into a shared library with `ld`. The testing of `find` was not duplicated for the implementation with optimizations because the original filesystem had been lost by the time the second system was implemented.

Table 6.1 shows that for file-system intensive applications like `postmark` and to some extent `find`, coverage ranges from 64% to 97% in the reference implementation. However for read and write intensive workloads, coverage is considerably lower. File-system intensive applications spend more time exercising the data structures in the file-system and VFS-layer code, rather than just doing device IO, thus more of the execution time is spent in system call code, which has excellent coverage. In the implementation discussed in Chapter 5, most of the interrupt handlers are treated as logged domains which

improves coverage in IO intensive workloads dramatically.

When we look at the implementation with optimizations, we see in Table 6.1 that coverage for postmark is mostly unchanged between the two implementations, but bzip2 and gcc show a 2.4x and 3.1x increase in coverage. This comes from running various deferred-work work-queues that are run after an interrupt in recoverable domains, unlike in previous work. We run the time-updating code in the timer interrupt as an unlogged domain which accounts for most of the unlogged operations in povray.

For bzip2 and gcc, we see that both execute a higher fraction of memory operations in recoverable domains and a larger absolute number of operations in recoverable domains. Table 6.4 shows that we have a considerable performance gain in these cases. Thus we are not reducing overhead simply by having fewer operations.

The number of kernel memory operations in each benchmark is not comparable between the old and new Recovery Domains implementation. Since the numbers are for different kernel versions, they are quantifying different algorithms and behaviors. However they sever to illustrate the amount of work the various benchmarks do while in the kernel.

The coverage numbers show that the performance gains we achieve come without reducing the percentage of the system that is recoverable. Rather, we improve coverage considerably.

6.4 Performance

Ideally, during error-free execution, Recovery Domains would impose as little overhead as possible. We first examine the performance overheads of the reference system from Chapter 4 and compare it to several baseline kernels

without Recovery Domains. Then we turn to the optimized system from Chapter 5, comparing it to the reference system.

6.4.1 Performance of Reference System

To isolate the overhead of Recovery Domains in the reference implementation, we measured benchmark run-times under three different kernels: the original 2.4.22 kernel compiled with gcc, the SVA ported 2.4.22 kernel compiled with LLVM, and the SVA ported 2.4.22 kernel with Recovery Domains, also compiled with LLVM. Comparing the first two options shows the overhead due to SVA alone. However, the overhead of the recovery techniques in this paper (and their design) are relatively orthogonal to any overheads caused by SVA itself, and comparing the latter two kernels isolates the overhead of the recovery techniques. All these measurements were taken using KVM (Linux's support for virtualization hardware on modern processors) on an Intel Core2 6420 running at 2.13GHz. We used the same benchmarks as in the coverage experiment with the same machine configuration. All performance measurements used the average of three runs; the variability was very low.

Memory usage by the recovery system was never directly measured. We noted, however, that the statically allocated 32MB of memory was never exhausted.

	Native Kernel	SVA	Recovery	Recovery vs. SVA
postmark	124	178	1004	5.6x
bzip2	13	12	13	1.08x
gcc(liblame)	23	23	29	1.26x

Table 6.2: Run-times (seconds) of benchmarks on Linux 2.4.22

	Native Kernel	Recovery	Recovery vs. Native
postmark	336	2741	8.16x
bzip2	12	16	1.33x
gcc(liblame)	28	55	1.96x
kernel compile	683	1988	2.9x

Table 6.3: Run-times (seconds) of benchmarks on Linux 2.6.27.5

Table 6.2 shows that the system-call intensive Postmark program is slowed down by about a factor of 5.6x. The other two benchmarks show low overhead due to Recovery Domains: 8% and 26% respectively. Overall, although the overhead for postmark is high, we believe this benchmark represents an extreme case for Recovery Domains. Furthermore, we are optimistic that these overheads can be greatly reduced by eliminating significant bottlenecks in this prototype implementation.

In addition to the Linux 2.4.22 kernel, we ran the benchmark applications on a port the 2.6 Linux kernel and measured the overhead of my recovery kernel relative to a native 2.6 Linux kernel. This port was functional but untuned due to both using a new runtime and the behavioral differences in the different kernel versions. Preliminary results are encouraging with 1.33x overhead for bzip2, 1.96x overhead for gcc, and 8.16x overhead for postmark. In addition, a kernel compile benchmark of Linux 2.6.27.6 configured with "allyesconfig" was run with 2.9x overhead. The results are summarized in table 6.3.

6.4.2 Performance of Recovery Domains With Optimizations

We evaluate the performance of Recovery Domains with optimizations by comparing to three cases. We compare to results for the runtime from chap-

Benchmark	Reference runtime		New Runtime for Linux 2.6.27	
	2.4.22	2.6.27	Non-optimized	Optimized
postmark	460%	716%	744%	134%
bzip2	8%	33%	3%	1%
gcc	26%	96%	13%	4%
povray	-	-	2%	2%
scp	-	-	50%	12%
scp -C	-	-	19%	4%
apache	-	-	5%	0%
apache (k)	-	-	20%	5%
postgresql	-	-	38%	12%

Table 6.4: Overheads as a percent of runtime over the same kernel without Recovery Domains

ter 4 for the Linux 2.4.22 kernel, the results using the same runtime on the Linux 2.6.27 kernel, the current runtime without the optimizations described in this chapter on the Linux 2.6.27 kernel, and the optimized Recovery Domains on the Linux 2.6.27 kernel. Since many optimizations and quite a bit of tuning went into the new runtime, comparisons between the optimized runtime and the runtime from Chapter 4 are given for perspective and completeness. The comparison between the 2.4 and 2.6 kernel gives an intuition for how much the recovery workload differs between the two very different Linux kernels.

Comparing performance of the optimized system to the reference implementation shows dramatic improvement. Though the reference implementation has a limited number of benchmarks, for those it does have we compare extremely favorably. Table 6.4 shows the percent overhead of recovery on various benchmarks. In postmark, which is a pathological case of file-system overhead, we see a factor of 3.4 improvement over the results for the reference system on 2.4.22. A more interesting comparison is the new runtime with and without the optimizations we describe. Here we see a factor of 3.8

Benchmark	All Optimizations	without load fastpath	without fresh memory	without locked memory
postmark	134%	601%	175%	156%
bzip2	1%	5%	1%	1%
gcc	4%	12%	6%	6%
povray	2%	2%	2%	0%
scp	12%	42%	17%	14%
scp -C	4%	16%	7%	6%
apache	0%	4%	2%	1%
apache (k)	5%	16%	10%	7%
postgresql	12%	35%	15%	16%

Table 6.5: Overheads of Recovery Domains without various optimizations

improvement.

While the improvement in `postmark` is notable, because `postmark` was a worst case, the improvement in more realistic workloads is just as dramatic. All except `postmark` have overheads of 12% or less with optimized recovery, and all but two are 5% or less. Moreover, without the new optimizations described in this work, the overheads would be much higher, e.g., 41% for `PostgreSQL` and 50% for `scp` without compression.

To evaluate how much each individual optimization contributes to these improvements *in the presence of all other optimizations*, Table 6.5 shows the effects of turning off each optimization while leaving all others enabled. Each should be compared with the overheads with all optimizations enabled (column 2). We see that all the optimizations matter, though disabling fast-path loads hurts the overheads the most.

We find that fresh-memory analysis is especially helpful in the network intensive benchmarks. The most data transfer intensive benchmark, scp, exemplifies this. We found during profiling that a kernel structure was being cloned in the network stack, causing a significant number of writes to the

fresh object.

We also see a gain from locked memory analysis, but not as large as the other optimizations. This is, in part, due to the extra annotation burden. Annotating locks requires understanding the locking discipline for each structure in the kernel and each type of lock. While many important cases are fairly easy to annotate, others are too fine grained. Most of the performance improvement we see here comes from annotating the locks of `mm_struct`. This plays an important role in optimizing common functions in the virtual memory system. However, trying to apply the same optimization to journal heads in the journaling layer used by ext3 proves significantly harder and unprofitable, at least to the extent we annotated it. Unlike the virtual memory system which locked whole objects and trees with a single lock, the journal layer took out locks to find and set individual bits. These operations meant the journaling code did little work in explicit locked regions and the cost of using the lock as a proxy for the object was more than the cost of the read and write that were protected by the lock.

In addition to the tests we list above, we also tested a program, mplayer, that has soft real-time requirements for frame decoding, to ensure that it is able to meet those requirements. In our tests, we found that mplayer dropped no frames in any configuration of Recovery Domains optimizations.

Chapter 7

Example Use With a Model Fault Detector

Recovery Domains enable recovery from detected faults, but do not provide a method of detecting faults. There have been significant research efforts to develop automatic fault detectors and to integrate those into operating systems. XFI [83], SVA [21, 22], CCured [55], SFI [84], and SAFECODE [24] are all examples of user- or kernel-space fault detectors. These and similar fault detectors improve safety but not reliability. Recovery Domains can be used by these fault detectors to provide a complete system capable of not only detecting faults, but recovering from faults and continuing execution.

Work on Recovery Domains was specifically inspired by work on Secure Virtual Architectures (SVA). Specifically, SVA provides a basis for the memory-safe execution of operating system kernels written in C (or another non-typesafe language). When memory safety of the operating system built on SVA is violated, SVA halts the machine. Further reflection on this scenario directly led to designing Recovery Domains so that memory faults could be recovered from, not merely detected. This chapter uses SVA as a prototypical fault detector to both illuminate the interaction between fault detector and Recovery Domains and to provide, speculatively, a complete use of Recovery Domains.

7.1 SVA Overview

SVA consists of two major components. The first is an API which wraps access to hardware functions, such as page tables, allowing higher-level, compiler-driven reasoning about the actions of operating systems. The second component, built on that foundation, is a memory safety system.

The hardware abstraction ABI provides a way to understand, restrict, abstract, and modify the use of (mostly) processor features by the operating system. For example, abstracting page tables to the ABI allows systems, such as memory safety, to validate that state changes conform to certain rules. Memory safety requires that kernel pages not be double-mapped, which it can enforce by monitoring changes to the page table which must occur via the ABI.

SVA contains a version of SAFECODE [24] specialized for operating system environments. This provides several important safety guarantees for operating systems written in unsafe languages, such as C. SVA aims to enforce fine-grained (object level) memory safety, control-flow integrity, type safety for a subset of objects, and a sound operational semantics even with dangling-pointer errors.

These guarantees are ensured by instrumenting the code to monitor memory accesses and pointer arithmetic. SVA uses a pointer analysis (Data Structure Analysis [48]) to discover which pointers may point to which object and to optimize bounds checks and object lookup based on this analysis. Type-safe objects, as discovered by the pointer analysis, require fewer checks. Pointer analysis allows instrumentation of indirect function calls to ensure the target function is one known to be acceptable to that call-site by the compiler. Pointer analysis allows optimizations over existing object bounds stor-

age techniques. Existing techniques store bounds information either as "fat pointers", which embed object bounds inside pointers by enlarging the pointers, or in a global table of all objects. Fat pointers, as used in CCured [55], break existing code by changing the size of pointers, causing interfaces to change and disrupting hard-coded pointer arithmetic in structures containing pointers. A large table of all objects in the system, as used by Jones and Kelly [43], creates a scalability problem as the number of objects grows. Rather than having a table which maps all objects to object meta-data (such as start address and length), tables are stored per points-to set discovered by the pointer analysis. This reduces the number of entries which must be searched through to find an object (or the lack of an object) for an arbitrary memory address.

The instrumentation amounts to: recording the creation of objects, including their bounds; checking bounds after pointer arithmetic; checking that objects exist before a load or store; checking that an address is the start of an object before a deallocation; checking that an indirect call is calling an expected function, found by compiler analysis; and recording the deallocation of objects.

7.2 SVA As a Driver For Recovery

As a fault detector, SVA is ideal. It detects faults at the time a bug occurs, rather than after corruption. When a fault is detected, the system is halted. This prevents corrupted state in the OS. Further, by providing memory safety, the recovery runtime is protected from memory bugs in the operating system. This alleviates many of the sources of potential corruption to the runtime by the operating system. Because a fault is triggered before

a stray write (or read) happens, no memory error can corrupt the runtime. A detector which triggered a fault far from the activation site could allow corruption to happen to the runtime.

Recovery Domains also do not interfere with SVA's notion of 'object'. SVA monitors allocations and deallocations, forming runtime knowledge of what language-level objects exist in the kernel address space. Since allocators are undone by an inverse operation, namely the deallocator, the existing SVA monitoring code for those allocators will maintain a consistent notion of objects even in the event of a rollback.

The use of a kernel ported to the hardware ABI can simplify a few important operations. First, atomic operations exist as calls to the ABI which are compiled to LLVM intrinsics. This removes the need for either porting of atomic operations to intrinsics just for recovery or the recognition of common inline assembly code sequences (which tend to be unique to each operating system). Secondly, since the register state is handled by the ABI for interrupts and process switching, the in-memory format of the saved registers is known. This allows the recovery system to directly restore the register and control state of suspended processes rather than having to instrument threads to verify that they were rolled back while suspended and to perform the register and control restoration after they are awakened.

SVA thus forms a model fault detector for Recovery Domains. Neither interfere with the assumptions of the other. SVA is greedy; SVA triggers faults without allowing the corrupting action to happen. When a memory fault happens, no corrupt state has been written thus protecting the recovery system from its most prominent source of bugs that could prevent recovery.

```
void func1() {
  void* x = alloca();
  sva_register(x);
  ...
  sva_unregister(x);
  return;
}

void func2() {
  void* y = alloca();
  sva_register(y);
  ...
  sva_unregister(y);
  return;
}

void func3() {
  void* z = alloca();
  sva_register(z);
  func1();
  func2();
  ...
  sva_unregister(z);
  return;
}
```

Figure 7.1: Overlapping stack allocations

7.3 SVA Integration

Recovery Domains and SVA can coexist with few changes. SVA's memory safety compiler is first run over the kernel instrumenting it for safety. Then the Recovery Domains compiler is run over the result. The runtimes for both systems are linked into the kernel.

The systems do not mesh perfectly. One change is required for correctness. Two other changes will lower the overhead of the combined system.

One area where stacking the transforms does not work is in the case of stack allocations. Address-taken local variables or programmer-allocated

stack memory cause the creation of non-heap objects which must be tracked by SVA. SVA requires the registration of these allocations with the runtime as objects, which it performs automatically as part of its compiler passes. Recovery Domains assume stack allocations will be undone when the stack is rolled back. During recovery, stack objects registered with the SVA runtime must be unregistered. This cannot be achieved by registering inverse actions with the Recovery Domains runtime to undo the registration of stack objects. Consider figure 7.1. The stack objects in `func1` and `func2` may overlap when those functions are called from `func3`. Since only one function is active, this isn't a problem in normal execution. However, if inverse actions were registered, by the end if `func3`, there would be three registered inverse actions to undo stack objects, yet these objects would already be freed. For heap objects, Recovery Domains defer deallocation to prevent this case, but for stack allocations this is not possible since control flow implicitly deallocates the objects. To solve this, either stack objects need to be promoted to heap objects or Recovery Domains need to provide an inverse action stack which can have actions removed if they no longer apply.

Two other changes are desirable, but not necessary to integrate the two systems. The main effect of not performing these optimizations is extra logging and loss of optimization opportunities. First, SVA introduces new memory locations to store bounds information obtained from the runtime. These address-taken temporaries should be ignored by the recovery system. These are easily identified since their only uses are by SVA runtime function calls. Second, many objects would appear address-taken and escaping by the recovery system if considered without knowledge of SVA's instrumentation. Objects are passed to the SVA runtime regularly. This does not represent a true escape and should be ignored by the recovery system.

```
struct fib_info * fib_create_info(
  const struct rtmsg *r, struct kern_rta *rta,
  const struct nlmsghdr *nlh, int *errp) {
...
  if (fib_props[r->rtm_type].scope > r->rtm_scope)
     goto err_inval;
...
  fi = kmalloc(sizeof(*fi)+nhs*sizeof(struct
               fib_nh), GFP_KERNEL);
...
  memset(fi,0,sizeof(*fi)+nhs*sizeof(...));

  if (rta->rta_priority) {
     temp = rta->priority;
     fi->fib_priority = *temp;
  }
...
}
```

Figure 7.2: Original kernel source for purposes of an SVA and Recovery Domains example.

7.3.1 Source Transformation Example

To clarify the combined compiler transformation of source by the safety compiler from SVA and the recovery compiler from recovery domains, we present an example of how the transformations stack. The example is simplified in several ways. First, for readability, the example is given at the C language level. This involves no loss of generality. Second, the accessing and caching of the active recovery domain is omitted. Details on that can be found in previous chapters.

Figure 7.2 is taken from the SOSP 2007 paper [21] on SVA. The example is an excerpt of `fib_create_info` from the Linux 2.4.22 network stack. This function allocates a data structure for new routes. The example shows dynamic array indexing, allocation, memory functions (which are dealt with specially by both SVA and Recovery Domains), and memory accesses.

```
   MetaPool MP1, MP2;
 2
   struct fib_info * fib_create_info(
     const struct rtmsg *r, struct kern_rta *rta,
     const struct nlmsghdr *nlh, int *errp) {
   ...
 7   //look up object bounds and then check the access
     getBounds(MP1, &fib_props, &s, &e);
     boundscheck(s, &fib_props[r->rtm_type].scope, e)
     if (fib_props[r->rtm_type].scope > r->rtm_scope)
        goto err_inval;
12 ...
     fi = kmalloc(sizeof(*fi)+nhs*sizeof(struct
                  fib_nh), GFP_KERNEL);
     pchk_reg_obj(MP2, fi, 96, NULL, SVA_KMALLOC);
   ...
17   //check bounds for memset without lookup since
     //we know the start and size from the kmalloc
     boundscheck(fi, (char*)fi + 95, (char*)fi + 96);
     memset(fi,0,sizeof(*fi)+nhs*sizeof(...));

22   //check that rta is a valid object
     lscheck(MP1, rta);
     if (rta->rta_priority) {
        //check that rta->rta_priority is valid
        temp = rta->priority;
27      lscheck(MP2, temp);
        fi->fib_priority = *temp;
     }
   ...
   }
```

Figure 7.3: Example of kernel source instrumented by SVA to provide memory safety. Added instrumentation is in bold.

Figure 7.3 comes directly from [21] and shows the various instrumentation performed by the safety compiler. The added instrumentation is in bold. On line 8 the bounds of `fib_props` are looked up and the indexing on line 9 is checked by line 9 to ensure it stays in bounds. Line 15 registers an object newly created by the allocator with the runtime. Line 19 checks that the memset will stay inbounds. The check is against an object locally allocated so the bounds information can be forwarded from the allocation site to the check rather than looking it up.

Figure 7.4 builds on figure 7.3 by including instrumentation for recovery. New instrumentation is in bold. To improve readability, temporary variables, whose names start with `rd`, are introduced to store the result of loads.

While this is a straight-forward replacement of loads and stores by `rec_dom_load` and `rec_dom_store`, a couple things should be explained. First, on line 28, `memset` was replaced by a call into the runtime to perform the memset. Memset streams a number of stores and can be specialized to more optimally interact with both locking, logging, and checks for optimized code paths. Second, `kmalloc` is an allocator and treated specially by the Recovery Domains system. However it does not appear so from this example. This is because the setup, tear down, error virtualization, and inverse logging for the allocator happen inside a wrapper created around `kmalloc`. This simplifies the transform as no call sites need to be modified when introducing Recovery Domains. Thus the call appears unchanged in this example.

In the full system, the fresh memory optimization would remove the call to `rec_dom_memset` and call a memset routine which performed no logging. This is possible because `fi` has not escaped by line 28.

```
MetaPool MP1, MP2;

struct fib_info * fib_create_info(
  const struct rtmsg *r, struct kern_rta *rta,
  const struct nlmsghdr *nlh, int *errp) {
...
  //look up object bounds and then check the access
  getBounds(MP1, &fib_props, &s, &e);
  rd1 = rec_dom_load(&r->rtm_type, ActiveRD);
  boundscheck(s, &fib_props[rd1].scope, e)
  rd2 = rec_dom_load(&r->rtm_scope, ActiveRD);
  rd3 = rec_dom_load(&fib_props[rd1].scope, ActiveRD);
  if (rd3 > rd2)
     goto err_inval;
...
  fi = kmalloc(sizeof(*fi)+nhs*sizeof(struct
               fib_nh), GFP_KERNEL);
  //kmalloc is an independent domain and has
  //effectively registered an kfree as an inverse by:
  //rec_dom_log_inverse(kfree, fi)
  pchk_reg_obj(MP2, fi, 96, NULL, SVA_KMALLOC);
...
  //check bounds for memset without lookup since
  //we know the start and size from the kmalloc
  boundscheck(fi, (char*)fi + 95, (char*)fi + 96);
  //memset is specialized in the recovery domain
  //runtime for efficiency
  rec_dom_memset(fi,0,sizeof(*fi)+nhs*sizeof(...),
                 ActiveRD);
  //check that rta is a valid object
  lscheck(MP1, rta);
  rd3 = rec_dom_load(&rta->rta_priority, ActiveRD);
  if (rd3) {
     //check that rta->rta_priority is valid
     lscheck(MP2, rd3);
     rd4 = rec_dom_load(rd3, ActiveRD);
     rec_dom_store(&fi->fib_priority, rd4, ActiveRD);
  }
...
}
```

Figure 7.4: Example of kernel source instrumented by SVA to provide memory safety then instrumented by Recovery Domains to provide recovery. Instrumentation added by Recovery Domains is in bold. ActiveRD is short for the active recovery domain. Accessing the active recovery domain is omitted for clarity.

7.3.2 Faulting Example

If a fault occurred at line 31 in figure 7.4, the SVA runtime would call `rec_abort`. The machine would be paused as the memory state was reverted. Since `kmalloc` is an independent domain, its changes to the state would not be reverted in this phase. After the memory state was reverted, the inverse operations would be performed. One such operation would be to call `kfree` with the pointer returned at line 16. This would undo the effect of the `kmalloc` semantically. Control flow would resume from the start of the domain and recovery would be complete.

Since SVA cares about objects and their boundaries, maintaining matched frees to allocations is required during recovery to keep consistent meta-data for the SVA runtime. Thus the inverse for `kmalloc` needs to be a wrapper around `kfree` which also calls SVA to unregister the object being deallocated.

7.3.3 Differences Preventing Merging of Runtimes

Although it would be nice to be able to share work between the SVA runtime and the recovery runtime, there is not an obvious way to do this. This comes from fairly different requirements on meta-data and what is instrumented.

SVA is primarily interested in objects. Objects are the language level construct against which it is basing its protections. Thus it must lookup meta-data for a region of memory corresponding to an object from an arbitrary address. To do this it maintains a modified splay tree in which the nodes are non-overlapping memory ranges. This allows it to find meta-data for an object given any address inside the object. This lookup supports checking for existence of objects (as in the `lscheck`) as well as retrieving the bounds of objects (via the `getBounds` call). Since meta-data is per-object, it is desirable to

minimize per-location overhead. Having a range lookup structure minimizes space usage by having a fixed overhead per-object rather than per-location and not having any memory use for unallocated memory.

Recovery Domains concerns itself with changes to memory locations. It is not tied to higher level concepts like objects. Further, to protect the entire kernel, it must protect allocators, which operate on pre-C-language-object memory. Thus, by necessity, it cannot operate strictly on an object level. Objects allow semantic optimization of recovery through inverses, but are not necessary constructs for recovery; recovery semantics make sense from a memory location standpoint, not from an object standpoint. Since potentially every load and store is instrumented, and loads and stores are a large fraction of the code executed, every effort is made to ensure extremely fast processing of them. To do this, a direct lookup structure, in this case a hash-table, with no chaining, is used to provide fast constant-time lookups. Speed requires that two locations which hash to the same meta-data bucket alias.

To place recovery meta-data inside the object meta-data of SVA would require using per-location memory, greatly expanding the size of meta-data and making it variable length. Each load and store would have a considerably more costly lookup, even if the required node was near the root of the tree. Placing SVA object meta-data inside recovery meta-data, by adding a chain of objects aliased by a location would increase the overhead of meta-data by 33%. Creating and destroying objects would also be more costly as all locations in that object would have to be updated.

There are situations in which some work can be shared. When a domain starts or ends, the recovery system must take a global lock. SVA, similarly needs a lock when registering allocated objects. These locks could be shared. A similar situation exists on an SVA load-store check.

7.4 Conclusion

SVA integrates with Recovery Domains with almost no changes to either. A few changes are necessary to keep SVA's runtime consistent with the kernel allocators in the event of rollback. SVA's memory safety pairs well with recovery since errors are detected before they are executed. This allows precise error recovery of the affected threads since the fault and the error are at the same location.

SVA's memory safety protects the recovery runtime from several methods of corruption which could prohibit recovery or cause recovery to produce unintended state. SVA, as published in [21] is vulnerable to the same page re-mapping corruption which recovery domains is, but work has been done on preventing this in [20].

Chapter 8

Related Work

In addition to the projects already discussed in this dissertation, work on Recovery Domains is related to several categories of previous research: transactional systems, techniques for recovering from faults in operating systems, and programming language support for recovering from faults. Many projects focus on fault *isolation* within the OS through new OS architectures, changes to commodity OS kernels, or language-based techniques.

This chapter starts with a discussion on the limits of recovery, which drives the design of Recovery Domains. Then three basic recovery techniques are reviewed, from very basic to complex. Then, specific recovery systems proposed in the literature are discussed. Finally, the studies of the limitations of both automated recovery and various recovery mechanisms are reviewed. Avoiding the limits of various recovery techniques, in part, drives the design of Recovery Domains and will be discussed at the end of the chapter.

8.1 Limits of Recovery

Automatic recovery is not without its limitations. Several studies look at the types of bugs encountered by various systems and whether re-execution would re-trigger them, how much and how often state should be saved to avoid checkpointing corrupted state, and how automatic can recovery be and still be successful.

Chandra and Chen [16] look at the types of faults encountered by several applications and classify them into three categories. The first category, comprising 72-87% of faults were independent of the operating environment and would be re-triggered by any generic, transparent recovery mechanism. Of the remaining faults, half were dependent on an operating environment that was unlikely to change. The remaining faults were transient faults, faults dependent on non-determinism or on state likely to have changed during transparent recovery. This supports the argument of this dissertation that permanent bugs (the first two categories in the study) are an important target for recovery, and systems which merely recover from transient faults by small changes to the environment are insufficient to handle large classes of faults.

Further work by Lowell, Chandra, and Chen [51] show a tension in transparent recovery between not losing user-visible state and losing corrupted state, hence allowing recovery. If execution since the last checkpoint has not caused output, rollback can happen with no user-visible effects. This can be ensured by checkpointing after at least every operation that causes user-visible changes. However, they show this has a high risk of saving corrupt state (thus producing incorrect output). If checkpoints are less frequent, corrupt state is saved less often, but corrupt output is visible to the user. This argues in favor of the decision to not make recovery domains transparent to programmers. By exposing faults, but providing strong guaranties about state, the programmer is able to reason about state after a fault and control the visibility of the fault (by failing the operation, retrying, etc.) as appropriate for that operation.

Chandra and Chen [17] also study how successfully a system recovers from faults using checkpointing on a continuum of state sizes and frequencies. At one end, programmer-supplied checkpointing saves a subset of pro-

gram state when the programmer deems it necessary. By minimizing the amount of state saved, the likelihood of the state being corrupted, and hence cause the fault to be re-executed, is minimized. At the far end of the spectrum is automatic checkpointing which saves the entire state before any output-commit. Although application-specific checkpointing performs significantly better than generic checkpointing, faults injected into the application cause 1-19% percent of application-specific recovery systems to recover incorrect state and 27-41% of generic recovery systems to recover incorrect state, leading to incorrect results.

8.1.1 Limits of Recovery-Oriented Operating Systems

Several operating systems have been constructed with recovery as a goal of the design. These systems illustrate the need for recovery domains even in the presence of novel system organizations.

Minix3 [41], a microkernel, includes a server responsible for restarting failed servers. In practice, this creates an ad-hoc fault handling system. Since servers have state, a failed server loses the state of all clients using it. In the case of the network server, all open connections would be lost. This puts the burden of recovery on every client for every type of server they use. Failures of some components, such as the file system, prevent recovery since the reincarnation server depends on them. While state can be stored in a state storage server, doing so increases the risk of restoring corrupted state, as well as introducing more dependencies for successful recovery. Recovery Domains would supplement this recovery mechanism by exposing failed requests to the client in a uniform way (or at least in a way they should be able to handle in normal fault-free operation) while preventing unexpected state loss due to a restart.

Chorus [64] includes support for a type of sub-system-specific checkpointing and restart ability [6]. Servers (or actors in Chorus) may checkpoint a representation of their state in a persistent region of memory. To the programmer this is simply support for programmer-supplied checkpointing discussed in 8.1. On a server restart, all state and resources except the checkpointed state are destroy, then the server is restarted and initialized with its saved state. As discussed in 8.1 this approach does not guarantee correct state after a restart.

EROS [71, 70] includes continuous checkpointing. Continuous checkpointing periodically marks all objects as copy-on-write and asynchronously writes out the state to disk. If an object in the checkpoint is modified it will be copied and the copy modified, thus not tainting the in-progress checkpoint. However, correctness of restart in this system is dependent on not having corrupted state in the checkpoint, a problem discussed in 8.1.

CuriOS [23] allows servers to bind client state as protected objects. These objects are strongly isolated, they are only available to the server when it is processing the client's request. The hope is that on a server restart, having client state protected allows a server to continue processing requests from the client. This again depends on not saving corrupted state. If state is corrupted and all necessary state to process a request is per-client, then after several retries, the client and its state can be killed. Further, the authors acknowledge that "restarting a service that has visible external effects may not always result in correct behavior". So while more carefully designed to isolate client state that exists in servers and bind that state to the client, this is not sufficient to ensure recovery. This approach is similar to MicroReboot [15, 14].

8.2 Recovery Techniques

Recovering from errors has been a concern of programmers as long as there have been misbehaving hardware, misbehaving software, and users.

8.2.1 Ad-hoc

The first ad-hoc recovery technique is ubiquitous: error codes. Error codes exist in operating system interfaces [99, 96, 97, 4], in standard libraries [1], shared libraries [3], and in process termination. Error codes are no more than a signaling mechanism indicating to the caller that an error needs to be handled. The state of the faulting system and what actions should be taken vary as widely as the users of error codes.

Exceptions [2, 5] are a common feature of modern programming languages. The semantics of exceptions vary widely between languages. They range from being a signalling and control mechanism in languages such as Java to also performing basic recovery in languages such as C++. When they perform recovery they often are limited to destroying objects on the stack and executing their destructors. This enables the common idiom of "release on destroy" whereby a resource, such as an open file, is released when the wrapping object is destroyed. This, however, is not a recovery feature of the language, but the disciplined use of exceptions to help with recovery. Exceptions, in practice, even within the same language, are used in different ways and to different extents by programmers, making it impossible to have a uniform exception handling regimen across all libraries a program may use. Further, exceptions are part of the interface to a library forcing the application to adopt the library programmer's exception handling methodology for those portions of the code which interface with that particular library; other

libraries may implement and constrain exceptions differently.

8.2.2 Isolation

Isolation is the notion of containing and limiting a portion of the system such that a fault in it can be dealt with without terminating the entire system. The most widely deployed form of isolation is the process. A process is, on modern operating systems, isolated from other processes and the operating system by the virtual memory management hardware (as controlled by the operating system). This contains the process state and prevents stray corruption of other process' state. Language-level isolation is used by several systems; such as SPIN [10], Singularity [42, 26], JavaOS [68], SafeDrive [95], SFI [84], and XFI [83]; to provide isolation to components of an operating system. These specific systems will be discussed in more depth in Section 8.3.2.

8.2.3 Checkpointing and Rollback

Using checkpoints for fault tolerance is well studied [61, 25] and many mechanisms have been proposed for it [46, 13, 67, 79, 88]. While approaches vary by the fault model they are targeting and the initiator of roll-backs (sensors, programmer driven, host driven), they must all capture the state of a system in such a way as to be able to restore the pre-fault state after the fault. There are limitations associated with how often a fault can be recovered from as discussed in 8.1. In general, like Recovery Domains, using checkpoint and rollback for recovery involves a separate fault detection mechanism and rules governing the restart of the system. Checkpointing and Recovery Domains both suffer from the output-commit problem: actions that are visible outside the system.

Checkpoint-based recovery has to contend with non-determinism in re-execution: a random event during normal execution will not happen during re-execution. How to handle these events which are not recorded in the checkpoint affect the amount of involvement required of the programmer. In some cases, such as a random number generator, using a different number during re-execution may be acceptable. However, a key-press (which is essentially a random event) should not be lost because of a rollback.

Checkpoints and re-execution have been discussed for use as debugging aids for a considerable time [94, 27, 8, 11, 18, 75, 82, 81]. Notably, checkpoints are used for deterministic debugging by King et. al. in [44] for operating systems by Srinivasan et. al. in [75] for applications. These systems are intended as debug aids and are not concerned with the continued execution of a system, but with allowing programmers to find the root cause of a fault. Recovery Domains are not concerned with providing a convenient mechanism for debugging a fault, but for recovering a system and continuing after a fault.

8.2.4 Transactions

Many of the low-level mechanisms we use are borrowed from database systems. These include the use of undo logs for recovery; the tracking of dependencies between undo-able operations (domains); the distinction between "protected" database operations (like Basic Domains), unprotected operations (Transparent Domains) and "real" operations (Unlogged Domains); and the use of open-nested transactions [54] with "compensating transactions" [34, 9] (reversible domains with programmer-specified undo operations).

Nevertheless, Recovery Domains are novel in several ways. First, none of the previous transactional systems we know of were designed to achieve a similar goal, namely recovery of entire systems from *unanticipated* errors. Achieving consistent state of a database under expected operating conditions gives recovery from anticipated errors as a side effect. Recovery from unanticipated, non-fatal errors are not handled however. All fatal errors should trigger the recovery system, to ensure consistent data, when the database is restarted. Second, transactional memory systems and database systems integrate recovery with several other aspects of the system. Recovery from anticipated errors is integrated with synchronization and data management to achieve "failure atomicity" and "execution atomicity" because they share mechanisms for logging, conflict detection, and rollback. Third, databases transactions focus on the recovery of the data in the database while Recovery Domains focus on the recovery of the internal state of the system, not on the data it is managing.

Programming language mechanisms are also being developed to add transactions to new or existing parallel programs [36, 53], but those mechanisms also are designed for both optimistic concurrency as well as recovery. In contrast, we introduce recovery mechanisms semi-automatically into an existing multi-threaded software system where the synchronization mechanisms are pessimistic (e.g., locks, semaphores, or monitors), i.e., they do not support rollback of program state. Therefore, we have to introduce logging and rollback into such existing systems, and have to optimize these by taking advantage of transparent and reversible operations wherever possible. Furthermore, Recovery Domains do all this *semi-automatically* to minimize the manual effort expended by the programmer.

TxLinux [63] exploits transactional memory within an OS kernel to simplify programming of mutual exclusion. It introduces a new primitive called a cooperative transactional spinlock for combining optimistic and pessimistic synchronization. It does not aim to improve the recoverability of the OS kernel in the presence of unanticipated faults, either within or outside critical sections. Like this work, we could leverage hardware support for transactional memory techniques. We would be using transactions not to replace locking in the kernel but to replace locking in the run-time.

Locus [86], QuickSilver [38], and TxOS [56] all add transactional semantics to operating system services. These systems do not aim to recover from *internal faults* within the operating system *per se*: an unexpected internal kernel error can crash the system just as with an ordinary, non-transactional kernel. These systems do have primitive mechanisms for rolling back and restarting intervals of code, but using these mechanisms for fault recovery requires significantly different policies, e.g., (a) treating dependences between threads as benign and tracking the dependences to identify which intervals must be rolled back if an error occurs; (b) designing appropriate error reporting policies when a request encounters an error and must be rolled back; (c) designing appropriate, and minimal, annotations to facilitate the design; and (d) developing optimizations to achieve acceptable performance. The reference implementation of Recovery Domains addresses (a-c) to achieve nearly automatic recovery but with arguably prohibitive overhead for production use. The optimized implementation addresses (d) by proposing new compiler optimizations to bring the overheads down to acceptable levels.

8.3 Recovery Systems

Based on these recovery techniques, many recovery systems have been developed. The first class of systems we discus extend languages with new features which explicitly provide enhanced recovery. The second class we investigate are operating system recovery systems. Some are based on standard checkpointing techniques, other use various forms of isolation (both hardware and software) with recovery. Finally, we look at a collection of recovery systems which are based on extensions to checkpointing.

8.3.1 Language Based

Some programming language extensions or programming models exist to attempt to improve error handling and the correctness of error return paths. Recovery Domains generalize and unify these models; these extensions can be treated as special cases of Recovery Domains, albeit ones with nice language-level syntax.

Weimer and Necula [85] extend Java exception handlers to have a stack of "compensation" code to release resources acquired before an exception is raised. This is motivated by observing that exception handlers rarely preserve invariants (such as lock state) or interface requirements. They present a language extension and runtime which allow blocks of code to be marked with compensation code that is executed if an exception unwinds past a given point. This can be seen as a semantically invertible recovery domain executing in an unlogged parent domain. While compensation stacks help prevent high level mistakes during exception handling, they do not provide low-level guarantees of the state of the program after exceptions.

Low-level memory state guarantees are provided by Shinnar et. al. [72] who extend the exception model of C# to support exceptions with memory undo. Their language extensions also support user defined hooks to undo arbitrary operations. Neither work handles memory dependence tracking and rollbacks across multiple threads. Without cross-thread rollback, memory state is still undefined after an exception.

Xu et al. [89] describe a language-agnostic programming model that enables error recovery for concurrent object-oriented programs. They define mechanisms for cooperative exception handling and (like database systems) take advantage of transactions in the underlying language for recovery as well. Since we are recovering commodity operating systems written in assembly and C code, we have neither the luxury of simply extending a language exception mechanism, nor can we rely on certain programming styles.

Rudys et al. [65] instrument Java code to allow runaway threads to be stopped without corrupting the runtime. This is done by inserting checks in loops to check if a thread has been killed. Blocking calls made to native code spawn a new thread with the original thread waiting on it. With these restructurings, a thread can be destroyed even if it is in a system call or runtime function. The goal of this work is simply to supply the equivalent of 'kill' for threads that exceed their resource limit without causing the JVM and the application hosting it to die. In this sense, this work is not concerned about general recovery, simply recovery from a narrow class of faults. Further, this work depends on the memory safety of the language implementation. This contrasts with Recovery Domains which is not meant as a way to terminate runaway threads in a safe manner but provides inter-thread recovery from any fault, including potentially resource limitations and infinite loops, given the correct detector.

Rudys et al. [66] extend runaway thread termination in [65] to preserve sane state in the event of a termination. An object-granularity transaction system is created and code is transformed to run all tasks in it. They have several assumptions: all transactions start at thread creation and end at thread termination, threads are short lived, and threads only handle one request before they terminate. Consistency of output channels and resources, such as open file descriptors, are not maintained. If circular dependencies are created, the youngest thread is killed. The basic approach and set of assumptions is from studying Java-based servers with general frameworks that create instances of programmer-supplied classes to handle requests. Overheads range from 6x to 7x in array light-code to 23x in their worst-case. Recovery Domains are more general: allowing request nesting, handling circular dependencies without rollback, and not making assumptions about the threading structures of a program.

8.3.2 OS Architectures for Isolation

The general technique of checkpoint and recovery has been well-studied in the past [25] and has been applied to operating systems. Bressoud and Schneider use deterministic hypervisor-level replay to replicate the state of a system remotely, thus facilitating efficient fail-over recovery for operating systems [12]. Deterministic replay subjects the system to the potential to either save corrupt state or re-execute a fault after rollback as discussed in Section 8.1.

The Rio Vista project [52] makes in-memory file-system caches persistent by using battery-backed RAM, and can use this persistence to recover data after reboots. I/O Shepherding [35] adds a new layer below the file system to unify file-system reliability in order to cope with storage faults. These

recovery systems target a different, narrower category of fault than Recovery Domains.

Recent projects describe alternative OS architectures that improve fault isolation by breaking OS-level subsystems into isolated components, thus isolating faults to a single subsystem. Microkernels [7, 33, 37, 40] run OS-level subsystems as user-mode server processes with a small underlying kernel. Although this provides memory isolation, if a process is restarted due to a fault, state is either lost in the traditional microkernel approach or state is restored, running the risk of restoring corrupted state. The risk of restoring corrupted state is discussed in section 8.1. Microkernels with recovery capabilities are discussed in 8.1.1. Microkernels pair well with Recovery Domains as microkernels force a very request-oriented structure on the kernel, dovetailing nicely with the basic unit of recovery in Recovery Domains: a request.

Virtual machine monitors have been used to isolate device driver faults from the rest of the kernel [50, 29]. This isolates code which is often buggy, but does nothing to recover from faults in the core kernel, faults in the VMM, or state loss by a restarted device driver. These projects use a VMM to provide isolation, the use of address spaces or instrumentation by systems such as Nooks [77] is discussed in Section 8.3.2.

Further, projects such as Singularity [42, 26], SPIN [10] and JavaOS [68] implement operating systems in type-safe languages, eliminating many silent errors. The projects use the compiler-supported, language-based isolation provided by memory safety as an important component of their architecture, but do not extend the language semantics to recover from runtime errors. Fault recovery in these systems require the correct handling of exceptions by the programmer rather than being handled in an automatic way as done in

Recovery Domains.

Recovery Domains is a more general mechanism that is language-agnostic and targets commodity operating systems and user-space systems. There is an enormous amount of engineering-hours invested in existing systems and rewriting all these systems is both unreasonable and too time and resource consuming to be a practical solution to reducing faults and providing recovery today. Further, these systems mainly provide memory safety, which is only one type of potential error. Recovery Domains allow recovering from any error for which a detector exists.

Direct Fault Isolation

In addition to alternative OS architectures, several recent projects focus on retrofitting commodity operating systems for improved fault isolation. Nooks [78] implements a driver isolation layer for Linux by using memory-management hardware to catch errant memory accesses made by buggy drivers.

Nooks statically wraps kernel extensions and drivers to provide isolation, logging, and recovery. All interactions of an extension are logged by the wrapper and isolation is enforced by copying kernel objects used by the extensions into and out of the address space. Nooks takes the isolation and monitoring approach one step further and rebuilds drivers' internal state by replaying driver-level calls after a fault in a device driver [77]. Nooks focuses on extensions to the kernel, including device drivers and makes rigid assumptions about their boundaries. Although some code is shared, different types of extensions require different wrappers. Wrappers for network drivers must be written which are separate from wrappers for sound cards or wrappers for file systems. Recovery Domains are request-oriented and are defined

by their entry points, subsuming any executed code in-between. This makes them simple to specify and uniform across the entire system, not specific to each subsystem. Further, complex interactions between components and extensions are naturally incorporated into a request. One of Nooks' most interesting features is its ability to restore sane state to a hardware device by replaying requests to the hardware. Recovery Domains are flexible enough to allow incorporation of Nooks for interacting with hardware devices. Recovery Domains are always on and cover the entire code base, including the core kernel and are not limited to extensions.

Like Nooks, SafeDrive [95] isolates drivers from the rest of the operating system, but SafeDrive uses language-level mechanisms on legacy C code to enforce memory isolation. SVA [21] uses language-level mechanism on legacy C code to enforce fine-grained memory safety for entire kernels, and XFI [83] uses binary instrumentation to interpose on memory accesses for OS extensions to enforce isolation.

SafeDrive [95] adds type annotations, in the form of bounds information, to Linux drivers. The source is then type-checked and transformed by the compiler to enforce those types. This provides software-based isolation. Safedrive recovers from type-safety violations by assuming a driver is restartable. SafeDrive logs specific actions, such as registering device names or new filesystems, and undoes them to restart the driver. However, not all global state changes are logged, only state updated though well defined mechanisms provided by the kernel. Thus a driver directly accessing core kernel data structures can leave the kernel in an inconsistent state after recovery. SafeDrive, like Nooks, has only been evaluated with drivers (which are extensions with well-defined interfaces).

Vino [69] protects a kernel from errant extensions (not drivers) by implementing software fault isolation to limit fault propagation and transactions to recover. All access to the kernel by extensions are mitigated through accessors functions with associated undo operations. These are logged as part of the transaction preventing the transaction from needing to log operations performed by the core kernel on behalf of an extension. Vino is a specially designed operating system, not an attempt to provide recovery to commodity systems like Recovery Domains, Nooks, and SafeDrive do.

These projects are complementary to this work; they focus on drivers and extensions. Projects such as Nooks provide a more powerful mechanism for reasoning about hardware interaction than recovery domains at the cost of having to write wrappers for each class of device driver. Recovery Domains supply recovery to the entire kernel, though may not always recover from a fault in a request which modifies device state. A combination of strong driver specific recovery, such as Nooks provides, to enable recovery for requests with low-level hardware accesses and Recovery Domains to enable recovery to the rest of the system would be a very powerful and comprehensive recovery system.

8.3.3 Extending Checkpoints

Checkpoints have many variations for automatic recovery.

MicroReboot

A variation on checkpointing and isolation is to isolate data from code while having stateless components. MicroReboot [15] maintains separate and consistent state for Java objects and can restart individual object without restarting the entire application. On a fault, just the faulting components are

restarted. This is, in spirit, the application level version of microkernels with data-isolation, such as those discussed in 8.1.1. This approach to recovery requires the redesign of applications and suffers from limitations similar to those shown in 8.1.1, namely potentially saving corrupted state.

Environment Changing

A further variation on checkpointing aimed at recovery, as done by Qin et. al. [60], is to change the environment in which a system is operating. If a fault is triggered, rollback and re-execution occurs but something is changed. Many things may be changed, such as memory initialization, delaying frees, padding allocations, reordering messages, adjusting scheduling, changing signal delivery, dropping network requests, etc. This approach, like most checkpointing approaches, also risks saving corrupt state. If state is corrupted and checkpointed far enough in advance of a symptom of a bug being manifest, the system will not be able to recover. Further, if the manifestation of a bug is not caught during re-execution yet the bug was triggered, recovery will erroneously succeed. As the set of environment changes increases, the set of possible changes the system can make becomes the power set of the set of changes. This means that a sufficiently complex implementation (one with many ways to change the environment) must have heuristics to prune the search space, which may cause it to miss a combination of changes that would avoid the bug.

8.3.4 Reducing Functionality

Systems based on error virtualization, such as ASSURE [74, 73] use error virtualization in much the same way as Recovery Domains. ASSURE was concurrently developed and was published in the same conference as recovery

domains [49]. ASSURE differs from recovery domains in several key aspects. First, with ASSURE, recovery points are determined by off-line analysis using fuzzing. Operating-system-supported checkpointing is used to snapshot the system at recovery points. In the event of a detected fault, the most current (or older if necessary) checkpoints are re-executed on a shadow machine. Re-execution searches for a recovery point which can be used as an error virtualization point. A rescue point is accepted by the re-execution search and the production binary is patched at the error to use it if the recovery point allows the application to continue, not change semantics, and process further requests. The use of binary patching allows ASSURE to be used on existing binaries, rather than requiring recompilation as Recovery Domains do. The systems differ in where application specific knowledge is stored. ASSURE stores it in the recovery systems, while Recovery Domains store the information in the original system's code base. These reflect the differing goals of the two systems. ASSURE focuses on recovery for binary applications, whereas Recovery Domains focus on recovery for systems which will be recompiled to use the recovery system.

Failure-oblivious computing [62] instruments C code with checks to ensure memory safety. At runtime, if a memory safety condition is violated, such as an out of bound store or a load, no memory state is changed but execution continues with a fabricated value.

Chapter 9

Conclusion

Recovery from faults is important for reliable systems. This dissertation proposes and evaluates **Recovery Domains** – a practical approach to recovery from faults in request-oriented operating systems. Recovery Domains provide a low overhead method of enabling recovery from faults detected by automated fault detectors with minimal changes to an operating system kernel.

Recovery of operating systems is a difficult problem subject to several important constraints, namely,

- Operating system recovery must work with commodity systems with few changes. There is a massive investment in developing modern operating systems; operating systems such as Linux and Microsoft Windows have estimated minimum development costs ranging to more than a billion dollars [87, 30]. Developing a new operating system or making major architectural changes to an existing one thus is prohibitively expensive.

- Recovery must work with commodity operating systems without changing the interface to the operating system. There is a broad range of application software built upon the ubiquitous commodity operating systems. Changing all these systems is not feasible, so to supply improved reliability for these systems, recovery must not require changes

to the interface they interact with.

- Recovery must work with the entire operating system. Faults can occur in any portion of an operating system. Automated tools to protect against faults can cause faults in locations which are unanticipated by the original programmer. A recovery system must be able to handle faults anywhere.

This dissertation presents **Recovery Domains** as a way to provide recovery to operating systems subject to these constraints. Recovery Domains attempt to meet the ideal recovery system presented in Chapter 2.2 subject to some inherent constraints. These goals are met, with the noted limitations, by Recovery Domains, as summarized:

Automatic recovery: Recovery Domains provide completely automatic recovery from the point a fault detector triggers recovery. The work of recovery happens in the Recovery Domains runtime and compiler transforms handle the instrumentation of the system to use the recovery system. All details of runtime monitoring happen without programmer intervention.

For the entire system: Recovery Domains provide recovery for almost the entire operating system. A few things, such as the core scheduler and early boot code are implicitly trusted. The programmer can exclude some additional code paths from protection by making them unlogged domains, but this is not required by the system.

Minimal porting effort: Recovery Domains require some, though minimal, porting effort. Request boundaries and how to deal with an error after recovery require manual annotation, but this annotation burden is very

low (a couple hundred lines of code for a code base of over four million lines in Linux 2.6).

For commodity systems: Recovery Domains are suitable for commodity systems. This is demonstrated with two ports of architecturally different versions of the Linux kernel and no reason is known that they should not be usable unmodified on other commodity systems, such as Windows, OS X, or the BSDs.

Recover from detected errors of any type: Recovery Domains are agnostic to the type of error that triggers rollback. A fault detector is necessary to detect faults and trigger rollback, but that is beyond the scope of Recovery Domains. Recovery Domains were inspired by real fault detectors and is designed to work with them.

Recovery not usable as a denial of service: Recovery Domains do not completely mask the existence of a fault. Faults are reported to the initiator of a request. This allows higher-level, existing logic to deal with the fault. The fault is masked from other tasks in the system. Because there is no automatic retry mechanism, faults will not keep reoccurring due to the recovery system.

Clear semantics: Recovery Domains have clear semantics. Both the action and aftermath of recovery and execution under non-faulting conditions is specified.

Recovery Domains, as an organization for operating system recovery, are a close match to common operating system architecture. The implementation-independent design of the recovery system is given in Chapter 3. Recovery is organized around the notion of request, recognizing that

requests can in themselves issue requests. Recovery Domains allow for the notion of context-independent requests to allow semantic recovery for requests which can be logically undone (rather than restoring memory state) and do not depend on the state of the requester. Recovery Domains track dependencies between themselves and other executing domains to ensure that consistent state is used by the entire system. This enables recovery or multiple threads, even if shared state is corrupted. Recovery Domains have a clear and simple programming model which does not require many modifications to a system.

We developed a reference design and implementation of Recovery Domains suitable for a standard operating system. This design appears in chapter 4. This implementation is based on memory versioning, undo logging, and a dependence graph. Memory versioning allows finding dependency information between domains based on reads and writes. Undo logging provides for restoring memory state in the event of a fault by restoring the initial or last committed contents to modified memory locations. The dependency graph records dependencies found at runtime by the memory versioning between active domains. The graph encodes the information necessary to determine when a domain may commit. The system was implemented in LLVM. Linux 2.4.22 was ported to the system and we measured the survivability, overhead, and coverage of the system. We found that the system could recover from, on average, 33 consecutive faults. The overhead ranged from 8% to 460%. The fraction of memory operations executed in recoverable domains ranged from 97% for kernel intensive code to 31% for code which spent most of its kernel time handling interrupts (the timer interrupt was marked as an unlogged domain).

Building on the reference implementation, we integrate compiler analysis and optimizations, as well as runtime analysis and optimizations, to dramatically lower the overhead of Recovery Domains. Analysis of the dynamic state of the dependence graph provides opportunities to elide monitoring of reads. When memory is protected by locks, we can use the memory version on the lock as a proxy for the locations covered by the lock and avoid versioning those locations while the lock is held. Memory which is allocated but has not become visible to other requests needs no instrumentation. These optimizations combine to lower Recovery Domains overhead to less than 15% in all but one test case.

Finally, since Recovery Domains require a fault detector, we give a speculative integration of the memory safety detector of SVA with Recovery Domains. A couple of cases are noted, for example stack object registration, which require additional work, beyond simply combining the transforms and runtimes, to maintain consistent meta-data in the SVA runtime. SVA and latter work on that platform, provide solutions to many of the potential sources of corruption to the recovery meta-data that could prevent correct recovery. This example integration provides a picture of a complete system which combines fault detection with automatic recovery.

Recovery Domains provide a compelling, practical, and easy to use system for recovering from faults in commodity operating systems. We hope that it inspires programmers to make more reliable systems; safety tool designers to not just detect faults in their designs, but use Recovery Domains to recover from the detected faults; and researchers to continue exploring recovery. Ultimately though, we hope through this work and continuing work in this area that people come to expect computer systems to be reliable and that the tools and techniques to make that happen come into wide-spread use.

References

[1] Ieee std 1003.1-2001 (posix.1).

[2] Iso/iec 14882:1998.

[3] *Xlib Programming Manual*. O'Reilly & Associates, Inc.

[4] *Macintosh Volume VI*. Addison-Wesley, 1991.

[5] *Java Virtual Machine Specification, The*, 2 ed. Prentice Hall PTR, April 1999.

[6] Abrossimov, V., and Hemann, F. Fast error recovery in chorus/os: The hot-restart technology. Tech. Rep. CSI-T4-96-34, Chorus Systems, Inc., August 1996.

[7] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for unix development. In *Proceedings of the USENIX Annual Technical Conference* (Atlanta, GA, USA, July 1986), pp. 93–113.

[8] Agrawal, H., DeMillo, R. A., and Spafford, E. H. An execution-backtracking approach to debugging. *IEEE Software 8*, 3 (1991), 21–26.

[9] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Becker, D., Fiuczynski, M., Chambers, C., and Eggers, S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the ACM symposium on Operating systems principles (SOSP)* (Copper Mountain, CO, USA, 1995), pp. 267–284.

[11] Boothe, B. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI)* (New York, NY, USA, 2000), ACM, pp. 299–310.

[12] Bressoud, T. C., and Schneider, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 80–107.

[13] Bronevetsky, G., Marques, D., Pingali, K., Szwed, P., and Schulz, M. Application-level checkpointing for shared memory programs. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (New York, NY, USA, 2004), ACM, pp. 235–247.

[14] Candea, G., Cutler, J., and Fox, A. Improving availability with recursive micro-reboots: A soft-state system case study, 2003.

[15] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. Microreboot–a technique for cheap recovery. In *6th Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004), pp. 31–44.

[16] Chandra, S., and Chen, P. M. Whither generic recovery from application faults? a fault study using open-source software. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN, formerly FTCS-30 and DCCA-8)* (Washington, DC, USA, 2000), IEEE Computer Society, pp. 97–106.

[17] Chandra, S., and Chen, P. M. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 91.

[18] Chen, S.-K., Fuchs, W. K., and Chung, J.-Y. Reversible debugging using program instrumentation. *IEEE Transacitons on Software Engineering 27*, 8 (2001), 715–727.

[19] Cinque, M., Cotroneo, D., Kalbarczyk, Z., and Iyer, R. K. How do mobile phones fail? a failure data analysis of symbian os smart phones. *Dependable Systems and Networks, International Conference on 0* (2007), 585–594.

[20] Criswell, J., Geoffray, N., and Adve, V. Memory safety for low-level software/hardware interactions. In *Proceedings of the Eighteenth Usenix Security Symposium* (August 2009).

[21] Criswell, J., Lenharth, A., Dhurjati, D., and Adve, V. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the ACM symposium on Operating systems principles (SOSP)* (Stevenson, WA, USA, October 2007), pp. 351–366.

[22] Criswell, J., Monroe, B., and Adve, V. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)* (2006).

[23] David, F. M., Chan, E., Carlyle, J. C., and Campbell, R. H. Curios: Improving reliability through operating system structure. In *Proceedings of the 2008 Symposium on Operating Systems Design and Implementation (OSDI)* (2008), pp. 59–72.

[24] Dhurjati, D., Kowshik, S., Adve, V., and Lattner, C. Memory safety without runtime checks or garbage collection. In *Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES)* (San Diego, June 2003).

[25] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Survey 34*, 3 (2002), 375–408.

[26] Fahndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G. C., Larus, J. R., and Levi, S. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of EuroSys* (Belgium, April 2006).

[27] Feldman, S. I., and Brown, C. B. Igor: a system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging (PADD)* (New York, NY, USA, 1988), ACM, pp. 112–123.

[28] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[29] Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., and Williams, M. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the on demand IT InfraStructure* (Boston, MA, USA, October 2004).

[30] Frazer, K. Building secure software: how to avoid security problems the right way. *SIGSOFT Software Engineering Notes 27*, 2 (2002), 71–72.

[31] Ganapathi, A., and Ganapathi, A. Why does windows crash. Tech. rep., University of Berkeley, 2005.

[32] Ganapathi, A., Ganapathi, V., and Patterson, D. Windows xp kernel crash analysis. In *Proceedings of the 2006 Large Installation System Administration Conference* (2006), pp. 12–22.

[33] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference* (1990).

[34] Gray, J. The transaction concept: Virtues and limitations. In *Proceeding of the International Conference on Very Large Databases* (1981), pp. 144–154.

[35] Gunawi, H. S., Prabhakaran, V., Krishnan, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Improving file system reliability with i/o shepherding. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP)* (New York, NY, USA, 2007), ACM, pp. 293–306.

[36] Harris, T., and Fraser, K. Language support for lightweight transactions. In *Proceedings of the ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (New York, NY, 2003), ACM Press, pp. 388–402.

[37] Hartig, H., Hohmuth, M., Liedtke, J., Schonberg, S., and Wolter, J. The Performance of u-Kernel-Based Systems. In *Proceedings of the 1997 Symposium on Operating Systems Principles* (October 1997).

[38] Haskin, R., Malachi, Y., and Chan, G. Recovery management in quicksilver. *ACM Transactions on Computer Systems* (1988), 82–108.

[39] Henning, M. The rise and fall of corba. *ACM Queue 4*, 5 (June 2006), 28–34.

[40] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. Minix 3: a highly reliable, self-repairing operating system. *SIGOPS Operating System Review 40*, 3 (2006), 80–89.

[41] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. Reorganizing unix for reliability. In *Asia-Pacific Computer Systems Architecture Conference* (2006), pp. 81–94.

[42] Hunt, G. C., Larus, J. R., Abadi, M., Aiken, M., Barham, P., FÃĎhndrich, M., Hodson, C. H. O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., and Zill, B. An overview of the Singularity project. Tech. Rep. MSR-TR-2005-135, Microsoft Research, October 2005.

[43] Jones, R. W. M., and Kelly, P. H. J. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging* (1997), pp. 13–26.

[44] King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (Berkeley, CA, USA, 2005), USENIX Association, pp. 1–1.

[45] Klensin, J. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), Apr. 2001. Obsoleted by RFC 5321, updated by RFC 5336.

[46] Laadan, O., and Nieh, J. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC)* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–14.

[47] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the Conference on Code Generation and Optimization (CGO)* (San Jose, CA, USA, Mar 2004), pp. 75–88.

[48] Lattner, C., Lenharth, A. D., and Adve, V. S. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Diego, CA, USA, June 2007), pp. 278–289.

[49] Lenharth, A., Adve, V. S., and King, S. T. Recovery domains: an organizing principle for recoverable operating systems. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (New York, NY, USA, 2009), ACM, pp. 49–60.

[50] LeVasseur, J., Uhlig, V., Stoess, J., and Gotz, S. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).

[51] Lowell, D., Chandra, S., and Chen, P. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2004), pp. 289–304.

[52] Lowell, D. E., and Chen, P. M. Free transactions with rio vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)* (New York, NY, USA, 1997), ACM Press, pp. 92–101.

[53] Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R. L., Saha, B., and Welc, A. Practical weak-atomicity semantics for java stm. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures (SPAA)* (New York, NY, USA, 2008), ACM, pp. 314–325.

[54] Moss, J. E. B. *Nested Transactions: an Approach to Reliable Distributed Computing*. PhD thesis, MIT Labratory for Computer Science, 1981.

[55] Necula, G. C., Condit, J., Harren, M., Mcpeak, S., and Weimer, W. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems 27* (2005), 2005.

[56] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating systems transactions. In *Symposium on Operating Systems Principles (SOSP)* (2009), pp. 161–176.

[57] Postel, J. Internet Protocol. RFC 791 (Standard), Sept. 1981. Updated by RFC 1349.

[58] Postel, J. Transmission Control Protocol. RFC 793 (Standard), Sept. 1981. Updated by RFCs 1122, 3168.

[59] Postel, J., and Reynolds, J. File Transfer Protocol. RFC 959 (Standard), Oct. 1985. Updated by RFCs 2228, 2640, 2773, 3659.

[60] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: treating bugs as allergies - a safe method to survive software failures. In *Symposium on Operating Systems Principles (SOSP)* (2005), pp. 235–248.

[61] Randell, B., Lee, P., and Treleaven, P. C. Reliability issues in computing system design. *ACM Computer Survey 10*, 2 (1978), 123–165.

[62] Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and Beebee, W. S. Enhancing server availability and security through failure-oblivious computing. In *Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI)* (2004), pp. 303–316.

[63] Rossbach, C. J., Hofmann, O. S., Porter, D. E., Ramadan, H. E., Bhandari, A., and Witchel, E. TxLinux: Using and managing hardware transactional memory in an operating system. In *Proceedings of the Twenty First ACM Symposium on Operating Systems Principles (SOSP)* (October 2007).

[64] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Lèonard, P., and Neuhauser, W. Overview of the chorus distributed operating systems. *Computing Systems 1* (1991), 39–69.

[65] Rudys, A., and Wallach, D. S. Termination in language-based systems. *ACM Transactions on Information System Security 5*, 2 (2002), 138–168.

[66] Rudys, A., and Wallach, D. S. Transactional rollback for language-based systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN)* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 439–448.

[67] Sancho, J. C., Petrini, F., Davis, K., Gioiosa, R., and Jiang, S. Current practice and a direction forward in checkpoint/restart implementations for fault tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18* (Washington, DC, USA, 2005), IEEE Computer Society, p. 300.2.

[68] Saulpaugh, T., and Mirho, C. *Inside the JavaOS Operating System.* Addison-Wesley, Reading, MA, USA, 1999.

[69] Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, WA, October 1996), pp. 213–227.

[70] Shapiro, J. S., Smith, J. M., and Farber, D. J. Eros: A capability system. Tech. rep., University of Pennsylvania, 1997.

[71] Shapiro, J. S., Smith, J. M., and Farber, D. J. Eros: a fast capability system. In *Symposium on Operating Systems Principles* (1999), pp. 170–185.

[72] Shinnar, A., Tarditi, D., Plesko, M., and Steensgaard, B. Integrating support for undo with exception handling. Tech. Rep. MSR-TR-2004-140, Microsoft Research, Dec. 2004.

[73] Sidiroglou, S., Laadan, O., Keromytis, A. D., and Nieh, J. Using rescue points to navigate software recovery. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 273–280.

[74] Sidiroglou, S., Locasto, M. E., Boyd, S. W., and Keromytis, A. D. Building a reactive immune system for software services. In *In Proceedings of the USENIX Annual Technical Conference* (2004), pp. 149–161.

[75] Srinivasan, S., Kandula, S., Andrews, C., and Zhou, Y. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference* (2004), pp. 29–44.

[76] Stets, R., Gharachorloo, K., and Barroso, L. A. A detailed comparison of two transaction processing workloads. In *In WWC-5: IEEE 5th Annual Workshop on Workload Characterization. Stets et* (2002), ACM Press, pp. 189–222.

[77] Swift, M., Annamalai, M., Bershad, B., and Levy, H. Recovering device drivers. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)* (Nov 2004).

[78] Swift, M., Bershad, B., and Levy, H. Improving the reliability of commodity operating systems. In *Proceedings of the 19th Symposium on Operating Systems Principles* (New York, 2003).

[79] Ta-Shma, P., Laden, G., Ben-Yehuda, M., and Factor, M. Virtual machine time travel using continuous data protection and checkpointing. *SIGOPS Operating Systems Review 42*, 1 (2008), 127–134.

[80] Traiger, I. L. Trends in systems aspects of database management. In *International Conference on Databases* (1983), pp. 1–21.

[81] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. Automatic on-line failure diagnosis at the end-user site. In *Proceedings of the 2nd conference on Hot Topics in System Dependability (HOTDEP)* (Berkeley, CA, USA, 2006), USENIX Association, pp. 4–4.

[82] Tucek, J., Lu, S., Huang, C., Xanthos, S., and Zhou, Y. Triage: diagnosing production run failures at the user's site. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (New York, NY, USA, 2007), ACM, pp. 131–144.

[83] Úlfar Erlingsson, Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: Software guards for system address spaces. In *Symposium on Operating System Design and Implementation (OSDI)* (Seattle, WA, USA, November 2006), pp. 75–88.

[84] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *IN PROC. OF THE 14TH SYMPOSIUM ON OPERATING SYSTEM PRINCIPLES* (1993), pp. 203–216.

[85] Weimer, W., and Necula, G. Finding and preventing run-time error handling mistakes. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2004), pp. 419 – 431.

[86] Weinstein, M. J., Jr., T. W. P., Livezey, B., and Popek, G. J. Transactions and synchronization in a distributed operating system. In *Symposium on Operating Systems Principles (SOSP)* (1985), pp. 115–126.

[87] Wheeler, D. A. Linux kernel 2.6: It's worth more! `http://www.dwheeler.com/essays/linux-kernel-cost.html`, February 2010. As of August 2009.

[88] Xu, G., Rountev, A., Tang, Y., and Qin, F. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE)* (New York, NY, USA, 2007), ACM, pp. 85–94.

[89] Xu, J., Randell, B., Romanovsky, A., Rubira, C. M. F., and Wu, Z. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS)* (Washington, DC, USA, 1995), IEEE Computer Society, p. 499.

[90] Ylonen, T., and Lonvick, C. The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard), Jan. 2006.

[91] Ylonen, T., and Lonvick, C. The Secure Shell (SSH) Connection Protocol. RFC 4254 (Proposed Standard), Jan. 2006.

[92] Ylonen, T., and Lonvick, C. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006.

[93] Ylonen, T., and Lonvick, C. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253 (Proposed Standard), Jan. 2006.

[94] Zelkowitz, M. V. Reversible execution. *Communications of the ACM 16*, 9 (1973), 566.

[95] Zhou, F., Condit, J., Anderson, Z., Bagrak, I., Ennals, R., Harren, M., Necula, G., and Brewer, E. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 2006 Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, WA, USA, November 2006), pp. 45–60.

[96] Linux 2.6.30 include/asm-generic/errno-base.h.

[97] Linux 2.6.30 include/asm-generic/errno.h.

[98] `http://marketshare.hitslink.com/report.aspx?qprid=10&qptimeframe=Y&qpsp=2009`. As of August 2009.

[99] `http://msdn.microsoft.com/en-us/library/ms679320.aspx`. As of August 2009.

[100] `http://w3counter.com/globalstats.php?date=2009-05-31`. As of August 2009.

[101] `http://www.top500.org/stats/list/33/osfam`. As of August 2009.