Recovery Domains: An Organizing Principle for Recoverable Operating Systems *

Andrew Lenharth Vikram Adve Samuel T. King
University of Illinois at Urbana-Champaign
alenhar2@uiuc.edu, vadve@uiuc.edu, kingst@uiuc.edu

Abstract

We describe a strategy for enabling existing commodity operating systems to recover from unexpected run-time errors in nearly any part of the kernel, including core kernel components. Our approach is dynamic and request-oriented; it isolates the effects of a fault to the requests that caused the fault rather than to static kernel components. This approach is based on a notion of "recovery domains," an organizing principle to enable rollback of state affected by a request in a multithreaded system with minimal impact on other requests or threads. We have applied this approach on v2.4.22 and v2.6.27 of the Linux kernel and it required only 132 lines of changed or new code: the other changes are all performed by a simple instrumentation pass of a compiler. Our experiments show that the approach is able to recover from otherwise fatal faults with minimal collateral impact during a recovery event.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability—*Fault-tolerance*

General Terms Design, Measurement, Reliability

Keywords Akeso, Recovery Domains, Automatic Fault Recovery

1. Introduction

As software systems increase in complexity, they become less reliable and more vulnerable to security exploits. Software developers are increasingly turning to automated tools to help find and stop security vulnerabilities and bugs. A number of powerful, emerging tools use a combination of compile-time and run-time checks to guarantee that an important class of safety properties (e.g., type or memory safety) will never be violated [16, 5, 29, 11, 4]. However, in most of these systems, an error detected at run-time causes fail-stop behavior, exposing high-availability systems to possible denial-of-service attacks.

Commodity operating systems are examples of systems that are crucial to achieving the high availability needed by modern services. System crashes cause low-end embedded systems to become unresponsive, result in lost data for mid-range laptop and desktop systems, and induce expensive outages for high-end server systems. System outages for high-end server systems have been estimated to cost companies millions of dollars per hour of downtime [7].

Unfortunately, no existing approaches recover automatically from faults in core kernel components. Nooks [25, 24] and SafeDrive [29] provide fault isolation and recovery for device drivers by interposing on the communication between drivers and the remainder of the kernel. Neither system, however, can tolerate faults in the remainder of the kernel. In fact, errors in the core of a 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 have significant output commit problems because they directly interface to hardware.

In this work, we propose an organizing principle for operating system recovery called "*recovery domains*," and a system called Akeso based on this principle, that enables complex multithreaded systems to recover from run-time faults in nearly arbitrary components. Our system provides strong recovery semantics, has minimal and localized effects if a recovery event is triggered, is easy to interface with by programmers or automated tools, requires little change in the source code, and is general enough to handle very demanding multi-threaded, state intensive, request-oriented software systems.

Unlike previous approaches [25, 24, 29, 3] that handle faults in OS extensions and drivers, Akeso is request-

* This work was supported in part by the National Science Foundation under grants CNS 07-16768 and CNS 07-09122. This version contains small revisions in the related work from that published at ASPLOS 2009. The original paper is available from ACM or by request.

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

oriented in the sense that it handles recovery at the request level, e.g., a system call or interrupt in an OS, rather than at the sub-system or component level. This means that a fault due to a request will ideally only affect that request, leaving the rest of the system intact and running unaffected by the fault. This view of faults as a property of a request is a stark contrast to both language-based recovery and driver recovery that view faults as a property of static code regions or transient states. This difference allows Akeso to handle fault for the *entire* kernel, and handle permanent faults gracefully whereas other recovery systems would re-encounter the fault during recovery[13].

To provide request-oriented recovery, Akeso binds each new request to a new recovery domain dynamically. As this request executes, Akeso tracks all state changes that result from the request and any dependencies that form with other recovery domains, such as through concurrent accesses of shared variables, so that all potentially faulty state can be rolled back after detecting a fault. When a fault occurs, Akeso uses logging and rollback techniques [10, 19, 6] to remove all state changes resulting from the faulting request, including state changes made by dependent recovery domains, thus restoring the system back to a consistent state automatically.

We apply Akeso to a commodity operating system, the Linux kernel, which is a particularly challenging test case for our techniques. Operating systems are massively multithreaded, event driven, and enormous software artifacts, all of which complicate our task of retrofitting an existing commodity operating system with recovery capabilities. However, using an annotation language and compiler support, we port the entire Linux kernel to Akeso without any significant design changes and using only 132 lines of code. Our resulting system recovers from 97% of errors outside of interrupt handler code, and adds between 1.08x and 5.6x run-time overhead for our benchmark applications.

One novel optimization we make is to use existing recovery paths in the kernel as part of our recovery protocol. This optimization helps to minimize how much state must be rolled back (in the same and other threads) on recovery, to reduce the changes needed to kernel code, and to preserve the error semantics seen by applications.

Our approach resembles Transactional Memory (TM) systems in that we use logging and rollback for restoring system state on a failure [10, 19, 6] (see [15] for a summary of extensive recent work in this area). Unlike TM systems, however, we only focus on recovery from failures and not on isolation between parallel computations (because of this distinction, we use the term "recovery domain" instead of "transaction"). As with TM systems, many of the low-level mechanisms we use are borrowed from the database literature, but our use of them is novel since we retrofit recovery semantics into existing software systems automatically, rather than introducing recovery semantics at the design inception and expressing them manually.

2. Example Fault and Recovery

The MCAST_MSFILTER integer overflow bug [23] serves as an illustrative example for fine-grained and request oriented recovery. This vulnerability allows an attacker to gain root privileges or otherwise compromise the system. This bug is detected by the SVA memory safety checker [4], but results in a fault due to the fail-stop manner of SVA. Rather than halting the kernel, we would like to recover from the attack without affecting the state of any TCP or UDP socket or any thread except the attacking (or buggy) thread.

The combination of the SVA memory checker and Akeso will cause a fault to be raised on the out-of-object access, modifications to state (such as the increment of the file reference count) to be rolled back to the start of the system call, and an error value returned to userspace. Akeso will undo only the effects caused by the attacking thread and will allow the system, including the IP stack, to continue running in a consistent state.

If we treated recovery as an operation on sub-systems or drivers, then this error would require the termination and restart of the entire ipv4 stack. This would, on a normal machine, have wide ranging effects, effectively causing all network using applications (and applications simply using sockets to localhost for IPC) to fail.

3. Recovery System Semantics and Overview

Akeso 's goal is to recover from unexpected faults automatically when a fault is detected in a request-oriented client system. It does so by automatically rolling back only the computation and state associated with the individual request that caused the fault. This technique achieves key benefits of both programmed exception handling recovery and checkpoint-rollback recovery: it enables fine grained recovery (like exception handling), yet does so largely automatically (like checkpoint and rollback). Akeso defines an error world view that can interoperate with existing error handling regimens and provides a set of capabilities with predictable semantics to programmers and automated tools.

3.1 Recovery Events

Responsibility for triggering a rollback of a client system (e.g., a kernel) is that of the client code. The recovery portion of Akeso exists to supply clean semantics and the necessary runtime to perform a rollback; it does not attempt to decide when one is needed. A rollback is triggered by executing a recovery instruction. This instruction may have been inserted by automated transformation of the source to enforce certain safety properties, e.g. SVA [4], or it may have been inserted by a programmer directly using the recovery instructions to simplify client system design.

3.2 Recovery Domains, the basic entities

To manage and optimize rollback recovery, Akeso organizes execution into intervals called recovery domains. A *recovery domain* is an interval of program execution identified by the program text. Recovery domains are composable, i.e., they may be nested. During normal execution, state changes during a domain are recorded and data dependences between recovery domains are tracked. Recovery domains have semantics partly like transactions in a shared memory system. Like transactions, they may be rolled back so that they appear as if not to have executed (during error recovery). Unlike transactions, recovery domains provide neither atomicity nor isolation. Memory dependencies formed between domains do not cause domains to be rolled back, but are merely recorded in case a fault occurs; in fact, there is no predefined behavior during "normal" execution that causes a domain to be rolled back. Any memory dependences between different threads must be synchronized as needed by the client code.

To minimize the overheads (during normal execution) of dependence tracking and of resolving dependences at domain commits, Akeso defines multiple kinds of domains with different logging and commit behaviors. Essentially, some of these are optimizations of the "normal" domain behavior. The kinds of domains are defined in Section 4.1.

Not all client code is recoverable. Code that involves explicit output commit, e.g., an actual disk write or a network output operation cannot be rolled back. Some low-level hardware interactions such as processor scheduling may also be difficult to roll back, depending on the client design. Akeso defines a special *Unlogged* domain that the client programmer can use to identify such segments of code.

3.3 Programmer Involvement

A client system, e.g., an OS kernel, may be ported to Akeso by identifying recovery domains and providing some information about them. Programmers may specify recovery domains at two levels of granularity. The high level specification is function based in which an annotated function's entry and exit designate the start and stop of a recovery domain. These may be paired with undo functions or other attributes. A low level interface is also exposed allowing a programmer to insert recovery domains at any point in the code, potentially with arbitrary recovery code. For example, a first pass at inserting recovery domains into a kernel may be to make all interrupt handlers and system calls recovery domains. This may be refined by adding various allocators and their inverses. For better recovery ability for requests that cross thread boundaries, work queues may be instrumented at a low level so the operation executes independent of the work queue dispatch routine and thus attaches to the request which caused the entry in the work queue.

3.4 Normal and Recovery Execution

During normal execution, client code executes without behavioral change, except the overheads to maintain metadata about memory operations. This metadata includes data structures to record state changes, manage domains, record recovery points, and record cross-domain dependencies. A domain is allowed to "commit" once all of the other domains it "depends" on complete execution (a cycle of dependent domains are detected and committed together).

If a recovery event is triggered within a domain, the domain is usually rolled back so that it appears not to have executed, and any domains dependent on it (either directly or transitively) are rolled back as well. These dependent domains may be on other threads, effectively rolling back all the dependent threads. Rolling back a domain undoes state modifications by that domain, restores the stack and registers (including program counter) to the beginning of that domain, and then performs whatever action is specified for that domain, such as returning an error code, retrying the operation, or any other arbitrary action. For example, if a `kmalloc` is rolled back, the caller will see an error value of 0 if that was what was specified as the error code for that recovery domain.

3.5 Predictable Semantics

Akeso provides the following (predictable) error semantics to client code. To explain this semantics, 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 4;
- 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 en error.

Ideally, the error recovery semantics of Akeso should provide a foundation for rolling back individual threads of execution (requests) independent of other threads and of the state of the underlying hardware. Briefly, when a fault occurs, it should appear to the rest of the system that nothing happened, and appear to the faulting thread as though control was transferred to a predefined point with no modification to global state. In essence, it should appear to the faulting thread not that a fault happened, but that it queried the

system which reported that if it had tried the operation it would have failed.

Compared with this ideal goal, Akeso has two limitations in practice. First, the faulting thread (and the overall system) may "see" changes to low-level hardware state that cannot be rolled back. Second, "innocent" threads, e.g., belonging to a different application, may have domains dependent on the faulting one because of client-level data flow and Akeso currently continues as if that domain itself encountered an error. This causes potentially innocent threads to experience side effects of an error. Retry mechanisms to restart "innocent" threads, thus further reducing the impact of a fault, are planned but not yet implemented.

4. Recovery System Design

We now describe in more detail how we achieve the recovery semantics described above. Section 4.1 discusses recovery domains in more detail. Section 4.2 describes our techniques for tracking dependencies between different recovery domains. Section 4.3 discusses our techniques for committing fault-free recovery domains and Section 4.4 describes our techniques for rolling back recovery domains after detecting a fault.

4.1 Recovery Domains

The basic unit of recovery is a *recovery domain*. A recovery domain is an interval of execution demarcated by calls to the domain start and end operations. Recovery domains log modifications to kernel state and also track dependencies with other domains, for use during successful commit and during rollback error recovery.

All kernel code runs in some recovery domain. Each recovery domain executes within a single kernel thread. Cross-domain (and hence cross-thread) dependencies are logged. 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 in the kernel and kernel code that has semantic inverses.

Several types of recovery domains exist to encompass a range of behaviors in the kernel as well as allow progressive porting of the kernel to use recovery domains. There are four domain types: the basic domain, the reversible domain, the transparent domain, and the unlogged domain, as briefly summarized in Table 1.

We call a domain *recoverable* if the kernel can recover from an error during the execution of the domain. Basic, reversible, and transparent domains are all recoverable, whereas unlogged domains are not. All domains perform their writes to kernel state "in place," i.e., to the original memory locations and not copies. All writes by the former three kinds of domains are considered *speculative* until the domain *commits*. These writes may be rolled back on an error. Writes by unlogged domains are non-speculative since they will never be rolled back.

4.1.1 Basic Recovery Domains

Basic domains are the most common unit of recovery. These domains log all writes, recording values that must be restored on a rollback, and monitor reads to discover runtime data dependencies on other domains.

The *parent domain* of any domain D is the domain that is executing when the entry point of domain D is reached and D begins execution. By definition, the parent domain will be in the same thread. Certain basic domains do not have a parent, including the entry point for an interrupt handler, for a system call from user space, or a task in an asynchronous work queue (the latter technically could have a parent but just does not record one). We refer to such a domain as a "*Root*" domain. All other basic domains have a parent.

Basic domains that have a parent do not commit on normal completion: instead, their metadata is merged with that of their parent, thus passing on the responsibility for committing kernel state to their parent. This commit behavior is similar to that of "closed nested" transactions in the TM literature [26] but, like all recovery domains, a key difference from transactions is that they are never rolled back during normal execution. Basic domains that do not have a parent wait for their dependencies to complete and then commit normally, as explained in Section 4.3.

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 domain entry function appears to return with the specified error condition. Control is thus returned to the parent at the domain entry point, which can then handle the error as desired.

To see why some "internal" domains may not record a parent, consider a request that spans multiple threads, e.g., a disk read, waiting on a network socket or waiting on keyboard input. In these cases parts of the logical request (the read) might span several threads as the device interrupts will be handled in the current active thread (e.g., on Linux). Operations that involve placing operations on work queues are also a source of cross thread domains. Logically, any operation performed on behalf of a domain in another thread should ignore dependencies with its "calling context" and form dependencies with the thread requesting the operation. We can accomplish this simply by not marking a parent for the entry domain of such an operation: such a domain will be dependent on the source thread's requesting domain and complete or fault with that domain. In this manner a well ported kernel will have domains for asynchronous operations essentially floating and attach to the thread that caused them when they exit.

Domain Type	Recoverable?	Logged?	Root?	Error Semantics	Success Semantics	Example Uses
Basic	Y	Y	Y/N	rollback and return error	if parent, merge with parent else commit when dependencies commit	any operation for which the kernel already handles failure, e.g., `open`
Reversible	Y	Y	Y	rollback and return error	commit and log success with parent	operations with semantic inverses that can be deferred, e.g., allocators, reference counts; *not* locks
Transparent	Y	Y	Y	rollback and skip domain	commit	operations that do not affect kernel semantics, e.g., LRU numbering of pages; disk cache read-ahead policies
Unlogged	N	N	N	abort	no-op	device manipulation code; optionally other low-level code, e.g., context-switch

Table 1: Primary Types of Domains

4.1.2 Reversible Domains

Reversible domains exist for operations for which a semantic inverse 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). We refer to this 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.

When a reversible operation is complete, we can commit it *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 logging and dependency work of a 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 operation is logged with the parent. If the parent must rollback, this inverse operation will be called to logically undo the operation of the reversible domain. So for example, a kmalloc will be logged and if the domain calling kmalloc aborts, kfree will be called on the allocated object. In principle, internal logging is *optional* and could be turned off by the kernel programmer, although our system leaves it on.

Since many reversible domains often involve the ownership or allocation of resources, we must ensure that a release operation (the inverse of an acquire) does not occur until the parent domain commits. Free, for example, must be delayed until its parent commits, else any malloc would become dependent on that domain. Thus inverse operations 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 acquire of a different lock. Instead, we handle synchronization differently, as described in Section 5.2.

4.1.3 Transparent Domains

Transparent domains are a straightforward optimization that can be exposed by programmers. There are certain operations that are strictly used for performance optimization and have no impact on the semantics of the kernel. 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. Another example is read-ahead operations for disk caches. 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 kernel intact.

4.1.4 Unlogged Domains

The final type of domain is the unlogged domain. This domain is for extremely trusted and performance sensitive code as well as for code which must not abort. In our current implementation, there are three kinds of unlogged domains: (a)

code sequences that write directly to external devices or networks; (b) the low-level context switching code, which replaces one executing thread with another; and (c) interrupt handlers, which are short and asynchronous. For example, code sequences writing to devices must not be rolled back because a sudden interruption and roll back of kernel state can leave the kernel and device in rather different opinions of the current state of the world.

An unlogged domain effectively commits each write as it is issued. This also overwrites any previous (speculative) write to each such location, preventing later readers from forming dependencies on those memory locations with the previous speculative writer. Note that the use of an unlogged domain does not preclude the specification of a semantic undo function that may be used by a parent domain which aborts to logically undo the effects of the unlogged domain. For example, when porting Linux 2.6 to Akeso , we designated the low-level allocator, `__alloc_pages_internal` , an unlogged domain and declared `free_pages` its semantic undo operation.

4.2 Tracking Dependencies Between Recovery Domains

Every recovery domain tracks what 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 rolling back of only those domains that 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 kernel.

On a read, metadata maintained by the run-time is consulted to check if the location contains a committed value. 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 metadata 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 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 kernel code runs in some domain, even if that domain is the default unlogged domain. This ensures that all kernel code participates in the maintenance of metadata.

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. A reversible or transparent domain, however, does not record its parent: instead it starts a new tree. It remembers the previous active domain only for the purposes of restoring it on exit.

4.3 Committing Recovery Domains

A domain goes through up to three states in order: Speculative, Completed, and Committed. All but unlogged domains begin in Speculative state. When a domain exits it transitions to 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 below. Since circular dependencies can form, 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. This is expensive in practice, so cycles in the sub-graph of the dependence graph containing only completed domains are collapse maintaining the graph as a directed acyclic graph.

When a basic domain not having a parent or having an unlogged parent transitions from Speculative to Completed state, it attempts to enter the Committed state immediately; this step is explained below. When a basic domain having a logged parent transitions from Speculative to Completed state it merges with its parent such that in subsequent execution the two domains are synonymous.

When a 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 4.1.2). If a reversible domain has a logged parent, the domain adds a dependence edge from the parent to itself. This is so that if the reversible domain must abort due to another thread, its parent will abort too, rather than using invalid results. 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 walks its write log and marks as non-speculative all memory locations 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 metadata may be deleted. Note that for an unlogged domain, all these operations are no-ops as it will never have logged memory writes so 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 dependences 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 of delayed operations may involve updating shared state.

4.4 Exception Events and Rolling Back Recovery Domains

A rollback is triggered when some run-time mechanism, a part of the kernel or inserted by an external tool, detects a potentially fatal error. Akeso then aims to roll back the faulty domain (or an enclosing parent domain) and any other domains that are dependent on it. To roll back a domain, Akeso restores any memory perturbations and undoes resources allocated for all dependent domains.

During rollback, all processors are halted at a known state, and one process walks the dependence graph of the faulting domain, rolling back any dependent domain. Because versioning is kept for writes as part of the metadata domains can be rolled back in any order, 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).

One key design goal is to use existing error return paths to preserve failure semantics, expedite recovery, and simplify our 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 semantics 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 the operating system while *requiring relatively few changes to the kernel* and *few or no changes to applications*.

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 kernel programmer. If most domains start

```
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);
    rec_domain_end();              //end the kmalloc domain
    rec_log_undo(kfree, result); //log inverse function
  } 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 1: Example of the protection domain transformation on kmalloc. The new functions implemented in the runtime are underlined. kmalloc_orig contains the original code for kmalloc.

at existing error checking points, then little further effort should be needed to perform recovery and error return from unexpected errors.

Asynchronous requests within the kernel 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 a independent domain for the worker which if fails, returns an error code though the same channel it would normally return an error code to the requester.

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.

5. Implementation

Akeso is an OS-agnostic compiler and runtime system that together provide the recovery semantics described earlier. Below we describe the implementation of Akeso and then describe how we ported Linux to use Akeso for recovery.

5.1 System and Runtime Implementation

Akeso consists of two main components. The first is a set of compiler transforms to instrument the kernel 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.

An important component in a deployed system is the detection mechanism that signals faults. The recovery system implementation is independent of any detection mechanisms the kernel programmer chooses to apply. A motivating example, as mentioned in the introduction, is SVA which implements a detector for fine grained memory safety. However, other error detection mechanisms could be used with this recovery system.

5.1.1 Compiler Passes

The compiler components are implemented as a series of passes in the LLVM compiler framework [12]. 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, 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.

The third pass transforms low level annotations into operations to setup and tear down protection domains, and uses `setjmp` to create a landing pad for control flow after a domain aborts. Figure 5 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.

5.1.2 Runtime Implementation

The runtime consists of 4 major components: protection domain management, logging, memory dataflow 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.

Protection domain management implements several data structures including the domain stack, the logs of writes and inverse actions, and the domain dependence graph. The domain dependency graph is stored as an outedge set in each domain. The completed but uncommited 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 outedge 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 outedge 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 outedges 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. The rollback mechanism simply walks the log in reverse order undoing operations and removing the current domain from the stack of domains. The commit mechanism walks the log updating memory regions to reflect the sequence number of the logged writes. Further, if any deferred actions are logged for the domain (and its children), they are executed (e.g. memory frees, reference count decrements).

5.2 Linux Port

We ported two very different Linux kernels to the recovery system. The first, which will be described here, is the SVA ported Linux 2.4.22 kernel as used in our SVA work [4]. Akeso itself is completely independent of SVA – we only use that as an error detection mechanism. To demonstrate that and to gather experience porting another kernel, we also ported Linux 2.6.27, which 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 protection domains. We then annotated the list of basic allocators (*kmalloc*, *kmem_cache_alloc*, *__alloc_pages*) as reversible domains, and specified their inverses.

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. With this observation, we model successful spin-lock acquires and releases as write-only memory accesses with old values of '0' and '1' respectively. This preserves the property

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 2: Changes to Linux 2.4.22 by type and lines of code

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, root 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 a root 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 letting exit perform large and complex state changes to global kernel data structures in an unrecoverable domain.

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 takes care of finalizing and freeing the associated object when the reference count reaches zero. All the major structures in the VFS layer used this idiom. Thus we declared the reference acquiring and releasing functions as inverses of each other and unlogged 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. Time permitted only the filesystem to be thoroughly ported this way.

Benchmark	#Mem Ops	Logged Reads	Logged Writes	Unlogged Reads	Unlogged Writes	Coverage
postmark	70B	96%	1%	2%	1%	97%
find	37M	46&	18%	27%	9%	64%
gcc	130M	23%	8%	52%	17%	31%
bzip2	63M	24%	10%	49%	17%	34%

Table 3: Percent of Dynamic Memory Operations By Domain Type

While we considered and ported many interesting cases (and several more primitives, such as queues, would benefit from special care), *no design changes were needed in porting the Linux kernel to Akeso*. Furthermore, we only modified or added 132 lines of code for our port (Table 5.2).

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.

6. Evaluation

We considered three metrics in our evaluation:

- The theoretical coverage of the recovery technique in terms of fraction of execution covered by the technique;
- The ability to recover from injected faults in the theoretically covered portions of code; and
- The overhead incurred during normal (i.e., fault-free) execution of the kernel.

Although we do not report recovery times, we have observed that recovery times are extremely short, far shorter than typical times for a partial or complete reboot of the system.

Unless stated otherwise, all experiments were performed using the ported 2.4 Linux kernel described earlier. Other methodological issues are described separately for each experiment below.

6.1 Coverage

An important metric in evaluating the recovery mechanism is how often the kernel is executing in code regions that are recoverable. Since our implementation "trusts" the scheduler, the interrupt handlers and the page allocator (and all code called by them), this will be less than the entire execution of the kernel. As noted earlier, we can expand this fraction substantially with a little more effort to isolate the actual hardware interactions for the former two.

As a proxy for execution time, we measured the number of memory operations (loads and stores) performed in

each type of domain. We use several benchmarks: postmark (which simulates a mail server), find, gcc, bzip, and a Linux kernel compile. The last two were not run on the SVA 2.4 kernel due to instability inherited from SVA. All benchmarks were run 4 times consecutively and coverage measured over that interval. 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.

Table 3 shows that for filesystem intensive applications like postmark and to some extent find, coverage ranges from 64% to 97%. However for read and write intensive workloads, coverage is considerably lower. Filesystem intensive applications spend more time exercising the data structures in the filesystem 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. We are investigating locations in the interrupt handlers that could be marked as logged domains to improve coverage in IO intensive workloads.

6.2 Random Fault Injection

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. 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 visibly failed. The above injection rate corresponded to roughly about 4 faults every second of kernel execution. At this rate, many non-trivial kernel operations would take a fatal fault without our techniques.

We repeated this experiment 5 times, each running a workload that included booting up the kernel, logging in as root, and beginning to run the postmark application. Over these 5 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. In the future, we aim to investigate why the recovery mechanisms failed in a small number of cases.

An interesting observation in this experiment was that when faulting, it was never observed that a kernel thread caused the roll back of the state of another thread. This argues favorably for the notion that recovery can be applied surgically to only small portions of the kernel related to a single task, rather than entire subsystems or drivers.

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

6.3 Performance

Ideally, during error-free execution, the recovery mechanism would impose as little overhead as possible. To isolate this overhead, 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 our recovery support, also compiled with LLVM. Comparing the former two 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 coverage experiment with the same configuration. All performance measurements used the average of three runs; the variability was very low.

Table 4 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 our recovery techniques: 8% and 26% respectively. Overall, although the overhead for postmark is high, we believe this benchmark represents an extreme case for Akeso . Furthermore, we are optimistic that these overheads can be greatly reduced by eliminating significant bottlenecks in our prototype implementation.

7. Limitations

The design of our recovery system has two main limitations. First, it does not extend reliability down to the level of device interaction. Second, it does not always preserve the consistency of application state (even in the non-error case): it only provides for the consistency of kernel state. We believe both limitations can be overcome by using some of the techniques in the literature that specifically target those areas.

To ensure consistent kernel state and agreement between the kernel and hardware devices on each others state, we could incorporate existing driver-oriented recovery mechanisms [24]. Recovery of the driver would be delegated to the driver recovery mechanism, while responsibility for the kernel state would be a matter for the proposed system. In such a system, calls into the driver would be logged as reversible domains which would employ the driver recovery mechanisms to perform a rollback.

The second limitation is we provide no guarantees about the consistency of user space state. We try to preserve the semantics of the operating system and not introduce behaviors that could never occur in the absence of the recovery mechanism, but this is not always possible. A prime example occurs with a system call such as wait(). Such a system call causes a semantic dependence between two threads at the kernel level, which is invisible to our dependence tracking. In the event that the thread being waited for forms a dependency to the waiter, a dependence loop is formed and deadlock would occur. To prevent this, we allow system calls to return to user space uncommitted (that is, the kernel changes are still speculative). This breaks the dependence cycle but also may expose the application to speculative state. Although our system aggressively commits any transaction it can, it is still possible that we have to roll back a system call that has already returned to user space. In such cases we kill the process. This behavior only affects a few system calls and we believe this can be remedied with careful insertion of basic, root protection domains around the wait queue code to insulate threads from each other.

8. Related Work

In addition to the projects we already discussed in this paper, our work 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. We compare these to our work briefly below.

Many projects focus on fault *isolation* within the OS through new OS architectures, changes to commodity OS kernels, or language-based techniques. These projects are complementary to our work: they find and isolate faults, whereas our goal is to recover from the faults after they have been detected.

8.1 Transactional Systems

As noted in the Introduction, many of the low-level mechanisms we use are borrowed from database systems, including undo logging, dependence tracking, unlogged domains, and reversible domains [8, 1], Nevertheless, our work is novel in several ways, most of these were described in the Introduction. In addition, previous TM systems and database systems integrate support for correct recovery ("failure atomicity") with optimistic synchronization for "execution atomicity": the two goals share mechanisms for logging, conflict detection, and rollback. 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 we have to optimize these by taking advantage of transparent and reversible operations wherever possible. Furthermore, we do all this *semi-automatically* to minimize the manual effort expended by the programmer.

TxLinux [17] exploits transactional memory within an OS kernel to simplify programming of mutual exclusion. They do not aim to improve the recoverability of the OS kernel in the presence of unanticipated faults, either within or outside critical sections. Like their work, however, we could leverage hardware support for transactional memory techniques to greatly reduce the run-time overheads of logging and rollback.

Transactions can be combined with error virtualization to provide recovery to applications. Sidiroglou et al. [22, 21] use runtime monitoring to detect faults and force control to *recovery points*. Recovery points are chosen either statically by profiling an application with a large number of bad inputs or by emulating the faulting region of code at runtime in an isolated test run. Code regions which have faulted in the past are always run in emulation, allowing detailed monitoring. Native execution resumes after the vulerable window is past. Recovery domains, unless told otherwise by the programmer, use a similar error virtualization scheme. However, the recovery action can be more complex than simply failing the request and returning a virtualized error. Further, recovery domains explicitly support multi-threaded code with complex interactions and do not assume recovery can be limited to a single thread.

8.2 OS Techniques for Recovering From Faults

Our system introduces new techniques and principles for recovering from operating system faults. The general technique of checkpoint and recovery has been well-studied in the past [6] 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 [2]. The Rio Vista project [14] 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 [9] adds a new layer below the file system to unify file-system reliability to cope with storage faults. Nooks [24], SafeDrive [29], and Vino [18], log resource allocations and interactions with the kernel to free allocated resources after a driver or extension crash, thus recovering from driver and extension faults without rebooting the OS. Nooks takes this general approach one step further and rebuilds driver internal state by replaying driver-level calls. Microreboot [3] maintains separate and consistent state for Java objects and can restart individual object without restarting the entire application.

Our techniques allow operating systems to recover from OS-level faults without requiring a reboot, and we handle faults in the entire operating system rather than just extensions or drivers.

8.3 Programming Language based Techniques

Some programming language extensions or programming models exist to try to improve error handling and the correctness of error return paths. Weimer and Necula [27] extend Java exception handlers to have a stack of "compensation" code to release resources acquired before an exception is raised. Shinnar et. al. [20] 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.

Xu et al. [28] describe a 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 OSes written in assembly and C, we have neither the luxury of simply extending a language exception mechanism, nor can we rely on certain programming styles. Recovery domains allow a wider variety of actions to be taken in recovery code, e.g. invalidating the entire disk cache, rather than being limited to resource release operations.

9. Conclusions

In this paper we presented Akeso, the first system for recovering automatically from faults for entire commodity operating system kernels. We introduced the concept of a recovery domain that divided the kernel into separate logical components for recovery, and we showed how we could use these recovery domains to recover from faults within operating system kernels. Based on fault injection experiments, our kernel withstood 35.4 randomly injected faults before crashing, and this high-level of reliability required only 132 lines of code to be changed in the operating system. Despite the heavy compiler-inserted instrumentation needed for recovery, our mechanisms had surprisingly low overhead for some benchmarks, but moderately high overhead for others. We are currently working to bring down these overheads.

References

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

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

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

[4] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, pages 351–366, Stevenson, WA, USA, October 2007.

[5] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 144–157, Ottawa, Canada, June 2006.

[6] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3), September 2002.

[7] W. Feng. Making a case for efficient supercomputing. *Queue*, 1(7):54–64, 2003.

[8] J. Gray. The transaction concept: Virtues and limitations. In *Proc. Int'l Conf. on Very Large Data Bases*, pages 144–154, 1981.

[9] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 293–306, New York, NY, USA, 2007. ACM.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. Int'l Conf. on Comp. Arch. (ISCA)*, pages 289–300, New York, NY, USA, 1993. ACM Press.

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

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. Conf. on Code Generation and Optimization*, pages 75–88, San Jose, CA, USA, Mar 2004.

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

[14] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 92–101, New York, NY, USA, 1997. ACM Press.

[15] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 69–80, New York, NY, USA, 2007. ACM.

[16] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 2005.

[17] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *SOSP '07: Proceedings of the Twenty First ACM Symposium on Operating Systems Principles*, October 2007.

[18] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX Symposium on Operating System Deisgn and Implementation*, pages 213–227, Seattle, WA, October 1996.

[19] N. Shavit and D. Touitou. Software transactional memory. In *Symp. on Principles of Distrib. Comp.*, pages 204–213, New York, NY, 1995. ACM Press.

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

[21] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using rescue points to navigate software recovery. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 273–280, Washington, DC, USA, 2007. IEEE Computer Society.

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

[23] P. Starzetz and W. Purczynski. Linux kernel setsockopt MCAST_MSFILTER integer overflow vulnerability, 2004. http://www.securityfocus.com/bid/10179.

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

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

[26] I. L. Traiger. Trends in systems aspects of database management. In *In Int'l Conf. on Databases*, pages 1–21, 1983.

[27] W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes, 2004.

[28] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, page 499, Washington, DC, USA, 1995. IEEE Computer Society.

[29] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *USENIX Symposium on Operating System Deisgn and Implementation*, pages 45–60, Seattle, WA, USA, November 2006.