CuriOS: Improving Reliability through Operating System Structure

Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, Roy H. Campbell
University of Illinois at Urbana-Champaign
{*fdavid,emchan,jcarlyle,rhc*}*@illinois.edu*

Abstract

An error that occurs in a microkernel operating system service can potentially result in state corruption and service failure. A simple restart of the failed service is not always the best solution for reliability. Blindly restarting a service which maintains client-related state such as session information results in the loss of this state and affects all clients that were using the service. CuriOS represents a novel OS design that uses lightweight distribution, isolation and persistence of OS service state to mitigate the problem of state loss during a restart. The design also significantly reduces error propagation within client-related state maintained by an OS service. This is achieved by encapsulating services in separate protection domains and granting access to client-related state only when required for request processing. Fault injection experiments show that it is possible to recover from between 87% and 100% of manifested errors in OS services such as the file system, network, timer and scheduler while maintaining low performance overheads.

1 Introduction

Operating system reliability has been studied for several decades [39, 19, 34, 46], but remains a major concern today [47]. Operating system errors can be caused by both hardware and software faults. Hardware faults can arise due to various factors, some of which are aging, temperature, and radiation-induced bit-flips in memory and registers (Single Event Upsets [30]). Software faults (bugs) are also very common in large and complex operating systems [13].

In the past, designs for reliable computer systems have used redundancy in hardware and OS software to attempt recovery from errors [5, 6]. Redundancy can mask transient and permanent hardware faults as well as some software faults [4]. However, it does not address the insidious problem of the propagation of undetected errors [34]. Additionally, these systems are extremely expensive to build and use [44].

Errors in a monolithic OS can easily propagate and corrupt other parts of the system [22, 52], making recovery extremely difficult. Microkernel designs componentize the OS into servers managed by a minimal kernel. These servers provide functionality such as the file system, networking and timers. User applications and other OS components are modeled as clients of these servers. Inter-component error propagation is significantly reduced because, in many microkernel designs, servers usually execute in their own restricted address spaces similar to user processes [25, 37].

Recovery from a microkernel server failure is typically attempted by restarting it. The intuition behind this approach is that reinitializing data structures from scratch by restarting a server usually fixes a transient fault. This is similar to microrebooting [10]. In Minix3 [25], for example, server restarts are performed by the *Reincarnation Server* [47]. If the server managing a printer crashes, it causes a temporary unavailability of the printer until it is restarted. Unfortunately, this approach to recovery does not always work. Many OS services maintain state related to clients. In such cases, a server restart results in the loss of this state information and affects all clients that depend on the server. For example, a failure of the file system server in Minix3 impacts all clients that were using the file system. Simply restarting the file system server does not prevent errors from occurring in these existing clients. Reads and writes to existing open files cannot be completed because the restarted server cannot recognize the file handles that are presented to it. Thus, while stateless servers such as some device drivers can be restarted to recover the system, this technique is not applicable for many important OS services that manage client-related state.

Writing clients to take into account OS service restarts and state loss is a possible solution. This requires clients to subscribe to server failure notifications and can re-

sult in increased code complexity. Another possible solution is to provide some form of persistence to the server's client-related state information. This allows a restarted server to continue processing requests from existing clients. Some microkernel operating systems like Chorus and Minix3 support the ability to persist state in memory through restarts; but they do not use this functionality for OS servers and, currently, only provide it as a service for user applications or device drivers.

Attempts to solve the state loss problem by simply persisting server state across a restart do not address the possible corruption of this state due to error propagation. An error that occurs in an OS server, like a typical software error, can potentially corrupt any part of its state [27] before being detected. This highlights yet another significant limitation of traditional microkernel systems. While such systems minimize inter-component error propagation, nothing prevents intra-component error propagation.

Checkpointing OS service state in order to mitigate the effects of error propagation is not a viable solution because rolling back to a consistent system state requires checkpointing of client state as well. Additionally, multiple checkpoints may have to be maintained in order to avoid rolling back to an incorrect state. This may be expensive in terms of memory and performance.

In this paper, we present CuriOS, which adopts an approach that significantly minimizes error propagation between as well as within OS services and recovers failed services transparently to clients. We accomplish this by lightweight distribution, isolation and persistence of client-specific state information used by OS servers. Client-specific state is stored in client-associated, but client-inaccessible memory and servers are only granted access to this information when servicing a request. Because this state is not associated with the server, it persists after a server restart. This distribution of state is illustrated in figure 1. A server failure that occurs when servicing a client can only affect that client and the restarted server can continue to process other requests normally.

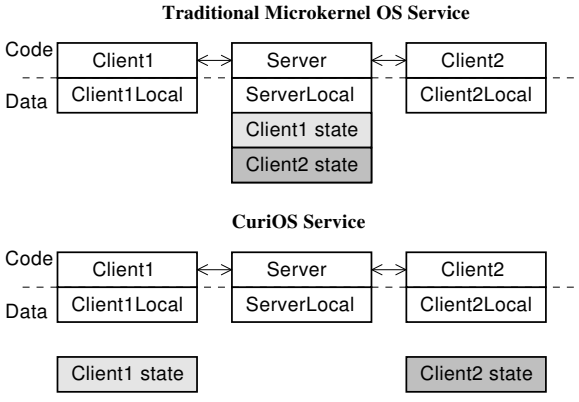

Figure 1: State Distribution

Distribution of state information from servers to clients for fault tolerance is not new. Researchers have exploited this technique to improve the reliability of file system services in distributed operating systems such as Sprite [53] and Chorus/MiX [33]. A more widely known example is Sun's stateless Network File System (NFS) [41]. But these designs do not protect the state information from being manipulated by clients and leads to various security problems such as those with NFS [51, 32]. Our design supports safe distribution of state by protecting the state from modification by clients. Our implementation is also lightweight because we use virtual memory remapping instead of memory copying to grant access to state. Additionally, we provide a generic framework for implementing distributed state and recovery for any OS service, not just the file system.

CuriOS is written in C++ and is based on the Choices object-oriented operating system [8]. It is being developed to provide a highly reliable OS environment for mobile devices such as cellular phones powered by an ARM processor.

Our work is complementary to other research in OS error detection such as the language-based type-safety techniques used in SafeDrive [55] and software guards used in the XFI system [50]. Employing such techniques in CuriOS can improve error detection latency and further reduce error propagation.

A preliminary design for CuriOS is available in a previous publication [18]. The contributions of this paper include:

1. A comparison and analysis of the effect of memory errors on OS services of several popular microkernel architectures, some of which are designed for reliability.

2. A detailed description of the state management framework implementation in CuriOS that reduces intra-component error propagation and enables transparent OS service recovery.

3. An evaluation of the CuriOS design using fault-injection experiments performed on several OS services.

The remainder of this paper is organized as follows. We investigate several related operating systems in Section 2. In Section 3, we look at the results of our investigations and present our observations for an operating system design that supports transparent recovery. Section 4 presents a brief introduction to the CuriOS architecture and details the framework used to manage OS

service state information. Section 5 describes the design and implementation of a few transparently restartable CuriOS components and drivers. We present an evaluation of several aspects of our current implementation in 6. We discuss a number of additional related topics in Section 7 and conclude in Section 8. In this paper, we limit our scope to the reliability aspects of CuriOS. A short discussion of some security issues is available in Section 7.

The dependability related terms used in this paper conform to the taxonomy suggested by Avizienis et al [3]. All names that use the font `Class` represent C++ classes.

2 Related Operating Systems

Some microkernel operating systems that are closely related to our work are Minix3 [25], L4 [37], Chorus [40] and EROS [43]. For the evaluation of each of these microkernel-based operating systems, we manually inject memory access errors into different OS servers to explore the effect of an OS error on its reliability. A memory access error is the typical manifestation of a hardware or software fault in an OS [52].

In all our experiments, a memory access error results in the termination of the OS server. Table 1 shows the results of our experiments. The effect of server termination after encountering the memory access error is shown in the third column. The last column presents our analysis of whether a restarted server will continue serving existing clients correctly (if restartability support were included in the corresponding OS). Except for Minix3, which already implements restartable services, this observation is based purely on source code analysis. Brief explanations for our conclusions are provided in each row. The entries in the last column for Minix3 are actual experimental results.

The rest of this Section discusses reliability aspects of the previously mentioned systems and several other related operating systems in more detail.

2.1 Minix3

Reliability support in Minix3 is provided by the *Reincarnation Server* which is able to restart both failed services and device drivers. Server restarts work well only for device drivers [47, 24]. This is substantiated by our experiments (Table 1). The file system server crashes on all invalid memory accesses and results in an unusable system. Even if the file system server were restarted correctly, existing open files would be inaccessible because of the lost server state.

Minix3 includes a data store server that can be used to store state that persists after a failure induced restart. The Minix3 data store provides some protection from errors in a server because it resides in a separate address space from the server. It has been used to implement failure resilience for device drivers [26]. A drawback of the data store approach is the additional communication and data copying overhead involved. This approach also does not restrict intra-component error propagation.

2.2 L4/Iguana

Iguana [35] is a suite of OS services that are implemented for the L4 microkernel [37]. This comprises basic OS services such as naming, memory management, timer and some device drivers. Our experiments study the behavior of some Iguana services when they encounter memory errors. Unlike Minix3, there isn't any support for restartable services. An analysis of the source code shows that server restartability, if implemented, still does not solve the problem of preventing the corruption of state and recovering it. As an example, the Iguana timer service maintains information about clients to which it periodically sends messages. This information will be irrecoverably lost upon a restart. A stateless server like the serial driver, on the other hand, can be restarted and may continue to work for existing clients.

More complex functionality such as a file system is part of the L4Linux [23] suite, which implements a complete Linux system as a user-mode server. Since most of the functionality required by Linux applications is implemented in this server, the reliability of all L4Linux applications depends on the reliability of this server, and thus, this design is not any more reliable than the normal monolithic Linux OS. This has been improved to some extent by isolating device drivers in separate virtual L4Linux servers [36].

2.3 Chorus

The Chorus OS [40] is designed for high reliability and is used in several telecommunication systems. In contrast to Minix3 and L4, services are executed in privileged mode and share the same address space as the microkernel. Chorus includes "Hot Restart" technology [1] that allows servers to maintain state in persistent memory and resume execution quickly after a failure. Unlike both the design we use in CuriOS and the Minix3 data store, all allocated persistent memory in Chorus is permanently mapped into the server domain. There is no mechanism in place that prevents state information saved in the allocated persistent memory from being potentially corrupted by an error that occurs in a server. Unfortunately, Chorus' operating system services do not take advantage of the "Hot Restart" functionality.

Table 1: Microkernel Operating System Recoverability after Server Failures

μkernel	Failed Server	Immediate Effect	After Restart
Minix3	File System (fs)	System unusable.	× Server is not restarted because the *Reincarnation Server* depends on the file system. Also, all current file system state information is lost.
	Network (inet)	All existing network connections fail.	× Restart does not help re-establish connections because state information is lost.
	Random Numbers (random)	Temporary read failure.	✓ Once the server is restarted, client reads begin working again.
	Printer Driver (printer)	Temporary printer access failure.	✓ Print job completes successfully after spooler retries request to the restarted printer server.
L4	Timer (ig_timer)	System unusable.	× All clients stop receiving timer interrupts. Restart does not help because clients waiting on interrupts can't re-register.
	Name Server (ig_naming)	No immediate effect.	× But many critical services inaccessible because lookup of registered names fail. Restart does not help because all registered clients need to re-register.
	Serial (ig_serial)	Serial port inaccessible.	✓ Request retries will eventually work.
Chorus	File System (vfs)	System unusable.	× Restart does not help because file system state information is lost.
	Network (netinet)	System unusable.	× Restart does not help recover existing network connections.
	Timer (kern)	System unusable.	× Restart does not address clients waiting on timeout.
EROS	Memory allocator (spacebank)	System unusable.	✓ Restore from a previous checkpoint may fix this error.
	Process Creator	System cannot create new processes.	✓ Restore from a previous checkpoint may fix this error.

2.4 EROS

EROS [43] is a capability-based system which saves periodic snapshots of the entire machine state to disk. When the system recovers after a crash, the last written snapshot is reloaded. This approach only works when the error is not present in the snapshot. Though the system performs some consistency checks on snapshots, correctness cannot be assured and several previous snapshots may have to be reloaded before a working version is obtained. Minix3's approach of restarting an erroneous server results in a re-creation of all internal state and has better chances of eliminating errors. Another drawback is that snapshots of large systems and device state (not currently performed by EROS) can be expensive in terms of memory and performance. Reverting to a previous system snapshot on a failure also results in a loss of all work done since the snapshot. This may be undesirable in some situations. For example all user input since the last snapshot is lost.

2.5 Other Systems

The Exokernel OS architecture [21] places most operating system abstractions in an application library and securely multiplexes machine resources. Similar to a monolithic kernel, error propagation is possible throughout the library OS and the application. There is no mechanism that provides transparent recovery for an application when errors occur in the associated library OS. An important advantage of the exokernel approach is that errors only affect the process in which they occur. This benefit is at the cost of a complex design for multiplexing shared resources like the storage subsystem. Four design iterations were required to build the XN storage system [31]. The Nemesis OS [29] also adopted a vertical structure similar to the exokernel architecture while providing explicit low-level guarantees for reserved resources. Error propagation was limited by enforcing isolation between device driver, system and application domains. The design of Nemesis was driven by QoS considerations and not surprisingly, does not include recov-

ery support for arbitrary errors in components. However, Nemesis provides QoS isolation between the clients of a system service. Services are designed to prevent one client from adversely affecting the QoS observed by others.

The Singularity system [28] adopts a radically different approach to security and reliability by using software enforcement of address spaces. CuriOS relies on hardware support to enforce memory protection.

3 Observations

From our study of the operating systems in the previous Section, we are able to make several observations about how the design of an operating system can impact its ability to transparently recover in the event of the failure and restart of an OS service.

Transparency of addressing: Clients should be able to use the same address to access the OS service after it is restarted. In EROS, since the whole system is restored to a previous checkpoint, this property is true. This is not supported by Chorus, whose hot restart algorithm restarts servers with a new address. Nor is this supported by L4 or Minix3 since a restarted server would be assigned a different address. A name server can be used to ameliorate this problem by maintaining a consistent name for the server across a restart. The restarted server would register its new address with the name server to provide continued availability.

Minix3 achieves transparency of addressing to some degree by using the file system server as a name server. A server can register itself as the handler for a device entry on the file system. For instance, the Minix3 *random* server mentioned in table 1 handles requests for the */dev/random* file system entry. In our experiment, we opened */dev/random* using the *open* system call and used the returned file handle to read a stream of random numbers from the server. If the *random* server crashed, reads using this file handle failed; however, once the server was restarted, reads using the same handle began to work once again.

Suspension of clients for duration of recovery: Clients should not time out or initiate new requests during the recovery phase. This property is supported by Chorus. In Minix3, clients are allowed to run when the server is restarting, and this results in errors when a client attempts to communicate with it. This is also the case in L4; the client will receive an error when it tries to communicate with a server that may be restarting. The whole system is restored to a previous checkpoint in EROS and therefore, this property is not applicable.

Persistence of client-related state: When a service is restarted, requests from clients must not fail because the server lost client-related state. Client-related state must be preserved and made available to the restarted server. Chorus and Minix3 have some support for in-memory state preservation, but this is not exploited by any of the OS services they support. An alternative is to save this information to stable storage. In EROS, all computation since the last saved checkpoint is lost.

Isolation of client-related state: Designs of existing microkernel operating systems provide unrestricted access to client-related state within a server. An error that occurs in the server can potentially corrupt state related to all clients. This intra-component error propagation problem exists in a large number of important microkernel OS services. In EROS, error propagation may lead to inconsistent data being checkpointed.

In the next Section, we describe how CuriOS fulfills all of these requirements and enables transparently-restartable OS components.

4 CuriOS Design

4.1 Structure and Overview

CuriOS is structured as a collection of interacting objects that represent various components and services. An object can be confined to an isolated memory protection domain in order to reduce error propagation. We refer to such an object as a *protected object* (PO). All methods on a protected object are executed with reduced privileges and run with hardware enforced memory protection. CuriOS applies the principle of least privilege to protected objects and only grants them access to memory regions that are required for correct operation. This prevents an error that occurs while running code in a protected object from corrupting other parts of the system by overwriting memory outside of the protected object. The reduced privilege execution mode also prevents protected object code from executing privileged processor instructions. Devices can be made accessible from within protected objects in order to encapsulate device drivers.

A protected object in CuriOS is analogous to a "server" in a traditional microkernel system. Our implementation of protected objects on the ARM platform only enforces restrictions on memory access. Implementations of protected objects on other platforms such as the x86 can additionally exploit architectural features to provide access control for other resources such as IO ports.

Protected objects work together with a small kernel, known as *CuiK*, in order to provide standard OS services as shown in figure 2. CuiK is a thin layer of the OS that

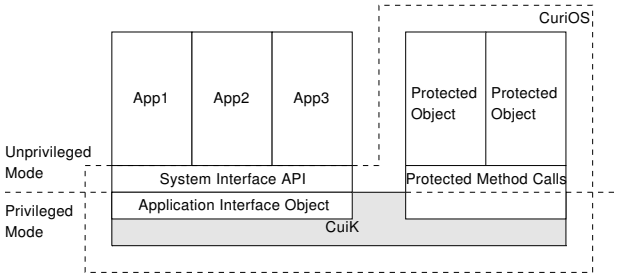

Figure 2: CuriOS Organization

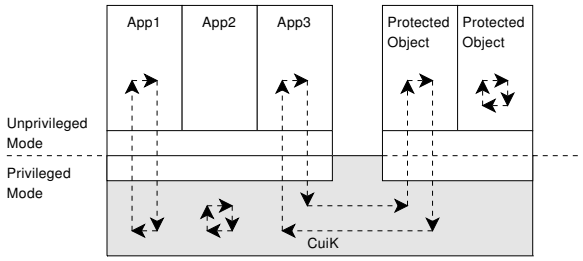

Figure 3: CuriOS Threads

runs with the highest privileges. It is composed of a small set of objects that manage low level architecture specific functionality such as interrupt dispatching and context switching. Communication between protected objects is managed by CuiK.

CuiK uses *protected method calls* to invoke operations on protected objects. Each protected object is assigned a private heap. A private stack is reserved for every thread that accesses the protected object. This stack is allocated at the first invocation of a protected method, contributing to a small delay in processing the first call to a protected object. Subsequent invocations of protected methods on the same protected object by the same thread reuse this stack. A protected method call results in a switch to a reduced privilege execution mode and constrained access rights to memory. The private stack and the heap are mapped in with read-write privileges. The rest of CuriOS is mapped in with read-only privileges. Permissions to write to any additional memory has to be explicitly granted by CuiK.

Our current implementation of protected method calls uses a wrapper object that intercepts method calls to a protected object and manages memory access control, processor mode switching and recovery. The combination of protected objects and CuiK results in a single address space operating system [11], where virtual addresses are identical across various components, but access permissions differ.

Threads in CuriOS are managed by CuiK. Using defined interfaces, a thread executing in CuriOS can cross user-space application, kernel, and protected object boundaries. For example, a system call in an application causes the thread to cross from user-space into the CuiK kernel. This same thread can cross from CuiK into a protected object using a protected method call. Some example threads are illustrated in figure 3.

CuriOS is written in C++ and uses object-oriented techniques to minimize code duplication and improve portability. Wrapper classes, for example, inherit from a common base class that provides the support functions used to switch protection domains and manage private heaps and stacks. In our current implementation, wrapper classes use multiple inheritance and also inherit from the class representing the object being wrapped. This allows the wrapper to exploit polymorphism and substitute the protected object anywhere in the system.

C++ exception handling is used as the error signaling mechanism in CuriOS [17]. Exceptions are raised for both processor signaled errors such as invalid memory accesses and for externally signaled errors such as OS infinite loop lockups (signaled by a watchdog timer) [16].

Exceptions are raised when errors are detected while executing code within a protected object. Exceptions that are not handled within the object are intercepted at the wrapper which attempts to destroy and re-create the protected object. The wrapper maintains a copy of the constructor arguments (if any) in order to re-create the protected object. This is similar to microrebooting or server restarts in Minix3 and can be used to fix transient hardware or software faults. The protected object is re-created in-place in memory ensuring that external references to it remain valid. This provides *transparency of addressing*. The method call is immediately retried on the newly constructed protected object. Multiple retry failures cause an exception to be returned to the caller. All normal system activity is suspended until the recovery is completed. Thus *clients are suspended for the duration of recovery*.

4.2 Server State Management

A server providing an OS service is implemented using a protected object. Clients are either user applications, or other protected objects. A protected object that represents an OS service can in turn operate as a client to another server.

A server that needs to maintain state information about clients uses state management functionality provided by CuiK to distribute, isolate, and persist client-related state. Servers that are completely stateless can be easily restarted and do not require this functionality.

A *Server State Region* (SSR) is an object represent-

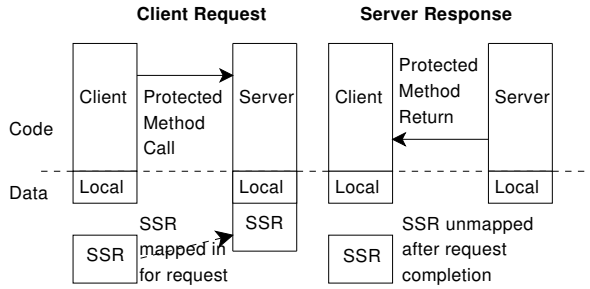

Figure 4: Request Processing

ing a region of memory that is allocated to store an OS server's client-related information. An SSR is created when a client establishes a connection to the server. For accounting purposes, the memory associated with the SSR is charged to the client. SSRs are protected from both the server and the client through hardware-supported virtual memory protection mechanisms. A client is never granted access to its SSR. A server is only granted write access to a client's SSR when it is processing a request from that client (see figure 4). The SSR is passed as an argument to the server's protected method call. The server can then use the SSR to store client-related information. It has full control over management of the memory within the SSR. Write permissions to the SSR are revoked when the protected method call returns. SSRs are implemented using a C++ object that holds a pointer to a hardware-protectable region of memory.

All SSRs in CuriOS are managed by a singleton object called the `SSRManager`. The `SSRManager` provides the functions used to register a new server, bind a client to a server (resulting in the creation of an SSR), undo a client-server binding (deletion of the associated SSR) and enumerate all the SSRs associated with a server. Each server using SSRs is required to provide a recovery routine that is invoked immediately after the server object is re-created upon a failure. This routine can query the `SSRManager` to obtain all associated SSRs in order to re-create the internal state of the restarted server.

The SSR-based state management framework provides *persistence and isolation of client-related state*.

4.3 OS Service Construction

How should the state of a generic OS server be structured in order to use the state management support provided by CuiK? There are two types of stateful servers. The first type is a server that does not require collective information about all of its clients in order to service a request. A server that provides pseudo random numbers based on a per-client seed is one such example. It only needs to know one client's seed in order to service a request from that client. Such servers can store client information in SSRs and can be transparently restarted upon a failure. All future client requests will continue to work correctly because its SSRs and the information stored in them is not lost.

The second type is a server that requires knowledge about all of its clients in order to service a request. Examples of this type are OS services like the scheduler and timer managers. Such servers can store client related information in SSRs and can redundantly cache this information locally to process requests. Upon a restart, such a server should be able to re-create its internal state from its distributed SSRs.

Many CuriOS components and drivers are stateless or structured as one of these two types of servers. When restarting, the server's recovery routine re-creates internal state from all SSRs. It is possible that the SSR that was in use at the time an error occurs is corrupted. The recovery routine can check the consistency of the objects in SSRs using simple heuristics before using them. CuriOS uses magic numbers in objects and these can be checked for corruption. We also use server-specific checks to ensure that pointers and numbers are within expected ranges. Unlike EROS which does consistency checks of all state during normal running time, these SSR consistency checks in CuriOS are only performed on exceptional conditions that require server recovery.

CuriOS servers can be multi-threaded and our current implementation shares the same virtual memory mappings for all threads. Thus, it is possible that multiple SSRs are mapped in when an error occurs. In this case, the error propagation is limited to the SSRs that are currently mapped in. This can be further improved by including support for thread-level protection.

4.4 Recoverable Errors

Protected method calls to servers are designed to retry the request after a server fails and restarts during the processing of the request. SSR-based recovery addresses a large class of errors that result in corrupted local OS service state. Complete reconstruction of service local state during recovery can remedy any such corruption. When an SSR is corrupted and multiple attempts at recovery fail, only the client associated with the corrupted SSR is affected and will need to be notified of a failure. The other clients of the service can continue to function normally. Thus, SSR-based recovery can minimize the impact of a software bug that is triggered by a specific client request and a consequent failure. When repeated attempts at processing the request fail, the client can be notified and the service can continue processing other requests that do not trigger the bug.

Restarting a service that has visible external effects may not always result in correct behavior. For example, restarting a printer driver due to a failure may cause another copy of the print job to be dispatched. This problem may be ameliorated to some degree by writing code that is restart-aware. This is achieved by incorporating some means of recording the progress made in servicing a request. This limitation has also been acknowledged for device driver restarts in Minix3 [26]. Similar to the approach taken by Minix3, we advocate notification of possible non-transparent recovery to applications or users.

5 CuriOS Services

Timer Management: A `PeriodicTimerManager` service provided by CuriOS allows user applications to access timer functionality. There is only one instance of this class in the system and it is created as a PO. Clients can start a timer by placing a request to be notified periodically. The job of the `PeriodicTimerManager` is to periodically signal a semaphore that the client waits upon. In order to support recovery from a restart of the `PeriodicTimerManager` service, SSRs are used to persist and distribute information regarding each client. The timer period, starting time and semaphore are stored in every client's SSR. The `PeriodicTimerManager` is implemented using a linked list of pending client notifications. Upon a failure-induced restart, the `PeriodicTimerManager` can re-create this complete internal linked list from the timer period and starting time information in the distributed SSRs.

Scheduling: CuriOS schedulers are modeled as process containers which manage a collection of processes and provide a scheduling strategy by presenting a method to pick the next process to run. A FIFO scheduler, for example, is implemented using a linked list of processes. The scheduler is created as a PO with clients as individual processes. A client SSR includes the pointer to the corresponding `Process` object and scheduling strategy-specific information such as priorities. If the scheduler is restarted after a failure, it queries the `SSRManager` for all its clients and re-creates its internal list.

Networking: The recovery mechanisms in CuriOS allow for the construction of an extremely reliable network stack. CuriOS uses the LWIP networking stack [20] encapsulated in two restartable protected objects: one for managing TCP connections and the other for UDP. LWIP creates a *tcp_pcb* or a *udp_pcb* data structure to manage state information for every connection. We refactored LWIP code to place these data structures within SSRs. In the case of TCP, for example, this includes all information necessary to service the incoming and outgoing packets of a connection. This includes the network addresses, ports, windows, sequence numbers and so on. If the TCP service crashes and is restarted, this information is used to resume the processing of packets. If this state information is not preserved during a restart, the unfortunate consequence is that all network connections in progress will be terminated.

Each SSR is associated with a client `Socket` object and is mapped into the TCP PO's address space when interacting with it. When there is an incoming packet, the corresponding SSR is located and mapped in before sending it through the stack. A similar approach is used to provide access to SSRs for the TCP PO's timer driven events.

All the assert code in LWIP was converted to throw exceptions instead of halting the stack. This comprehensive error detection in LWIP helps reduce error propagation and improves recovery rates.

File Systems: CuriOS currently supports two different file systems. `CramFSFileObject` is a class that provides access to a compressed file on the read-only CramFS file system [14]. When a file is opened, an instance of this class is created as a PO. This instance only has information about its backing storage and does not maintain any state regarding clients. Hence it does not require usage of the server state management functionality. The method call to read a file provides both the offset into the file and the required number of bytes. The PO is only granted privileges to modify its own data and the destination buffer. Calls to other objects like the backing storage are mediated by CuiK. Using a PO for each file has several reliability benefits. An error that occurs when processing one file is contained within the PO and cannot corrupt arbitrary memory in the system. If the error were transient, a restarted PO can continue serving clients. If there is an error in a compressed file stored on the disk that causes the decompression routines to fail, it only causes an error in the clients that were reading that particular file.

CuriOS also includes support for the Linux ext2 file system. An `Ext2Container` PO is created for every ext2 file system on disk. This manages the inode and free space bitmaps. If this PO crashes and restarts, it can re-read this information from disk. An `Ext2Inode` PO is tasked with managing all interaction with a file. This PO only has privileges to modify the inode it represents, which, in turn, has all the pointers to disk blocks comprising the file. This has similar reliability benefits as the `CramFSFileObject` protected object. Since POs are re-created in-place, the same objects can be used to access the file after the service is restarted.

It is important to realize that when we refer to file system service recovery, we are not dealing with recovering corrupted file system state on disk. There are many other programs that are designed to handle corrupted data on disk and this is an orthogonal problem with recovering the state information maintained in a live file system server.

Device Drivers: The serial port driver in CuriOS is implemented as a PO with complete access to the memory mapped registers of the serial port controller. This PO is stateless and only has one client: the CuriOS console object. Errors that occur when reading or writing to the serial port are handled by restarting this PO and retrying the request. CuriOS has a NOR flash driver that is implemented as a stateless PO. This PO is created with read/write access rights to physical memory regions that map to NOR flash chips. An error that occurs in this PO can lead to potential corruption of arbitrary data stored in NOR flash, but cannot easily corrupt read-only mapped system memory. Protected objects are also used to encapsulate drivers for interacting with the hardware timers. These are used to start, query and stop the hardware timers. Interrupts from the hardware timers are also dispatched to them. These are currently stateless and can be restarted. The `PeriodicTimerManager` service depends on the correct functioning of these driver objects.

6 Evaluation

CuriOS has been implemented and runs on the Texas Instruments OMAP1610 H2 mobile device development platform [48] and the QEMU [7] emulated Integrator/CP platform [2]. In this Section we evaluate the CuriOS implementation in terms of error recovery capabilities as well as performance and memory overheads. We also present a brief analysis of the refactoring effort involved in constructing CuriOS services.

6.1 Error Recovery

In order to evaluate the error recovery capabilities of the CuriOS implementation, we resort to fault injection experiments using a modified version of the QEMU emulator. We have verified that error recovery works equally well on real hardware and we only use the emulator in order to enable non-intrusive and large-scale automated fault injection. Our QEMU-based fault injection tool picks a random instruction in pre-specified functions and injects a fault just before that instruction is executed. In each experiment run, we inject exactly one fault and observe the behavior of the system.

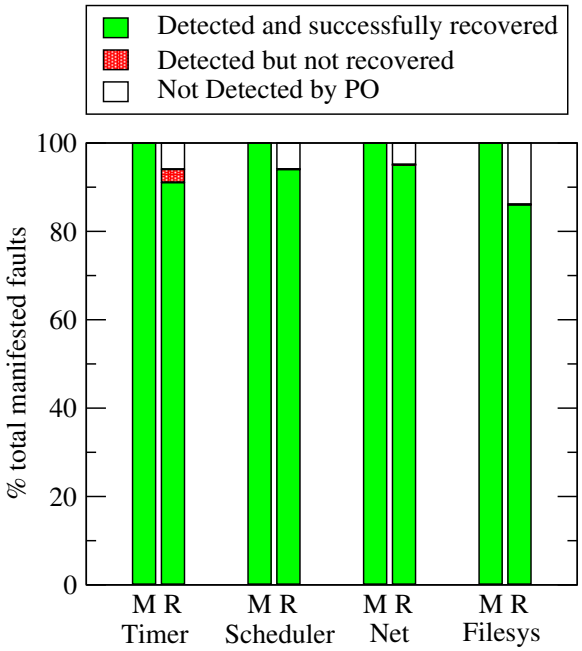

Figure 5: Error Recovery after Fault Injection

Our fault injection tool is used to inject two types of faults. The first type of fault is a memory access fault (an ARM processor "data abort"). These virtual memory faults are instantaneously detected at the injected instruction and immediately cause an error. The fault latency is zero in this case and error propagation is limited. The other type of fault that we inject is a register bit-flip. The tool randomly flips a bit in one of the register operands for the selected instruction. Register bit-flips do not always lead to errors (a corrupted register can be overwritten). They can, however, lead to latent errors which may not be detected immediately. This has been used to emulate several common programming errors such as incorrect assignment statements and pointer corruptions [38]. After recovery, we terminate experiments once we are able to verify that the OS is still functional (ability to schedule processes and access the disk).

We present error recovery results for the timer manager, system scheduler, networking stack and file system. In all of our experiments, we consider the system to be usable and successfully recovered if, at the end of the experiment, CuriOS can schedule new processes and can access the disk. We perform between 250 and 600 fault injection experiments per server and the results are shown in figure 5. The "M" columns present results for the memory access fault experiments and the "R" columns present results for the register bit-flip experiments. The Y axis represents the percentage of faults that had some visible manifestation (errors). Note that

all of these manifested faults would result in a service or system failure in most existing operating systems, which do not implement restart recovery with state management as used in CuriOS. An error is reported as successfully recovered if the system is usable after recovery. In some cases, a client's connection to a server is terminated because of a corrupted SSR or repeated errors while the system, as well as other clients using the restarted service, still remain usable. We count such cases as successful system recovery in the figure shown. In the face of arbitrary errors that corrupt a client's SSR or request, there is no possibility of maintaining the unfortunate client's connection.

Timer Manager: This experiment is set up so that a couple of processes that use the timer are started before faults are injected into the server. These are applications like a clock that displays the time of day. While the memory abort errors are fully recovered by reconstructing the internal timer queue, in the register bit-flip injections we see a few cases (3%) where errors are detected but are not recovered. These happen when the recovery procedure itself encounters an error and is unable to complete. We are working on eliminating such cases by improving the robustness of the recovery routines. 6% of the register-bit flips evade detection by the protected object mechanism and cause unrecoverable errors in other parts of CuriOS. This is because register bit-flip errors can propagate to other CuriOS subsystems through invalid method call arguments and results. We do not yet perform an exhaustive check of the validity of all method call arguments and results. This is a work in progress and we expect it to significantly reduce this inter-component error propagation.

System Scheduler: This experiment is set up similar to the timer manager experiment with several processes in the system. The goal of this experiment is to examine if a failure in the scheduler can be recovered and if CuiK can continue scheduling processes. For the memory abort experiments, re-creation of the internal linked list is always successful. In the case of the register bit-flip experiments, 6% of the errors are not detected by the PO mechanism and cause CuriOS to crash.

Network: We run a simple web server and an echo server in CuriOS while also running an HTTP client that fetches a half-megabyte file from an external host. Faults are injected into the LWIP code for TCP processing of IP packets on both the send and receive paths. If an error is detected, the TCP stack PO is restarted and the request is retried. If multiple attempts at executing a protected method fail, an exception is thrown. If this exception is thrown when processing an incoming IP packet, the packet is silently dropped by the IP layer. This has no effect on the correctness of TCP because this is similar to packet loss on the network and is recovered by the TCP stack. If an exception is thrown back to a client with a TCP connection handle, the TCP connection for that client is terminated. We verify that the network stack is still usable and other connections are unaffected in spite of a single connection failure. For the network stack, 5% of the manifested register-bit flip faults are not detected and consequently, not recovered. While the system is recovered in 95% of the cases, 45% of these recoveries were at the cost of a single client's TCP connection termination due to state corruption.

File System: We inject faults into the code for the ext2 file system that is used when accessing a file on disk (in `Ext2Inode`). The memory abort faults are always completely recovered after a retry. 13% of the manifested register bit-flip faults are not detected by the protected object mechanism and are therefore not recovered. Again, this is due to error propagation via corrupted method call arguments and results.

6.2 Performance

A protected method call incurs additional processing overhead in comparison to a normal C++ method call. We made use of both of CuriOS' supported platforms to measure the overhead associated with protected method calls. On the OMAP1610 hardware platform we measured the overhead in terms of microseconds of execution, and on the QEMU emulator we measured the overhead in terms of instructions executed. The OMAP1610 was clocked at 96MHz, and the same test source code was used for both platforms. Table 2 shows the overheads for the two types of protected method calls: a protected call into a stateless server that does not require the mapping of an SSR and a protected call info a server that uses an SSR to manage state information. In the second case, additional processing is required to map the SSR into memory. The time overhead for switching into and out of a protected object domain is comparable to the cost of performing two context switches (148 microseconds for two switches) in CuriOS. Since a protected method call is analogous to switching between two microkernel domains, we believe that this represents acceptable performance. The numbers reported here are the average

Table 2: Protected Method Call Performance

Protected Call	**Instruction Overhead**	**Time Overhead (microseconds)**
Without SSR	1594 ± 4	195.7 ± 0.5
With SSR	4893 ± 3	378.9 ± 0.9

of 100 trials with error estimates provided by the sample standard deviation. We believe that these overheads may be further reduced with careful code optimization.

Apart from the extra code implementing the protected object mechanism, a major source of overhead is the need to flush the TLB when switching between page tables. While the ARM architecture allows for selective flushing of TLB entries, our current implementation does not support this feature. The single address space design of CuriOS helps to keep the costs of protected method calls down by obviating the need to flush the virtually tagged caches on the OMAP1610 ARM processor.

How fast does recovery happen? When an error is detected, the exception handling framework signals the error and the C++ library unwinds the stack and destroys stack objects. Restarting the server requires re-running the constructor for the PO and code to recover information from SSRs (if required). Altogether, the time from error detection to a recovered system is usually on the order of a few hundred microseconds.

6.3 Memory Overheads

Protected objects, like user applications, require additional page tables to enforce memory protection and this results in some memory overhead. Each PO also has an associated heap and a stack for each thread that can execute within the protected domain. The memory overhead due to stacks depends on the number of threads that use the PO. The use of SSRs also results in some memory overheads. We use hardware protection to isolate SSRs. However, hardware protection is not always available for small memory regions. Thus the minimum size of an SSR is determined by the smallest hardware-protectable region of memory. For example, on the ARM platform, this is a 1 KB page. Our current implementation uses a page for the minimum size of an SSR. This results in some memory waste. If this is a concern for small embedded devices, our design can be extended so that multiple SSRs share the same protected area. This saves space at the cost of better isolation between the SSRs. This problem may be mitigated by future architectural support for finer granularity of access control such as Mondriaan Memory Protection [54]. Nevertheless, the total memory overhead per protected object in CuriOS is only on the order of tens of kilobytes when there are a small number of clients. This includes 20 KB for a minimal set of page tables plus memory pages for the heap, per-thread stack and per-client SSR (at least one page for each).

6.4 Refactoring Effort

Our proposed OS design requires writing OS service code to encapsulate objects in protected domains and to utilize our state management framework. The protected object support in CuriOS is implemented through wrapper objects. Wrappers are currently written by hand and consist of a one line statement per object method. The statement is a C++ preprocessor macro that expands to the code required to switch into and out of the associated protection domain. This additional complexity may also be avoided by using an automated wrapper generation tool. Code-changes are also required to refactor OS services so that they can make use of the state management framework. The use of SSR-based state management in the file system, scheduler and the timer manager required less than 50 additional lines of code in each component. In order to convert the LWIP networking stack to use SSRs, we had to change around 100 lines of code. This mostly involved replacing calls to its internal allocator with the SSR-based state management code.

7 Discussion

7.1 Security

Our security model relies primarily on address space isolation. We only map in memory that is necessary for a protected object to execute. This includes the unprivileged code and stack for the object as well as the SSR region for the request. Our model is most closely related to Nooks, which uses similar protection policies for kernel memory. We differ from Nooks in that protected objects execute in an unprivileged processor mode. This prevents a malfunctioning or compromised server from affecting the integrity or confidentiality of information used by inactive clients. Although we restrict the scope of possible damage, our current implementation does not consider intentionally malicious modules. We are working on fortifying the protected method call and server state management mechanisms by borrowing ideas from systems like EROS.

7.2 Fault-Tolerance

A number of standard fault tolerance techniques are available in literature. These include redundancy in hardware and software, transactions, error correction codes for memory, majority or Byzantine voting, and other software fault tolerance approaches [49]. Some of these techniques can be directly applied to CuriOS to further improve its fault tolerance. These techniques may be used to ensure that the core of the system (CuiK and recovery code) itself is protected from failure.

VINO [42] used transactions to roll-back changes made by misbehaving kernel extensions. We have also investigated the use of software transactional memory techniques to protect component state in Choices [15].

The use of transactional semantics alone to recover complete component state is only effective when errors are detected before commits. When this property cannot be enforced, there are no constraints on error propagation within the component. However, when used together with our SSR-based approach that reduces error propagation, transactions can provide an additional layer of protection to SSRs while they are being manipulated by a service.

In addition to some of the operating systems discussed in Section 2, many other system designs incorporate virtual memory protection to improve reliability. In the Rio project [12], virtual memory was used to protect the file cache from corruption by errors occurring elsewhere in the system. The protected object concept is similar to a virtual memory protected region in Nooks [46]. However, unlike Nooks, a protected object executes in an unprivileged processor mode. More importantly, while Nooks is designed to wrap OS extensions such as device drivers, a protected object can encapsulate core OS components. Unlike the shadow driver mechanism [45] used by Nooks, the SSR-based recovery mechanisms can isolate requests that cause crashes because of a software bug and continue servicing requests that do not trigger the bug. This is possible because of the rigorous partitioning of per-client state in CuriOS. When using the shadow driver approach, the bug will be triggered in the shadow driver just as it was in the original driver since the same code is used.

OS service design using SSRs is closely related to the principle of crash-only software [9]. Similar to crash-only components, recovery involves a component restart and component crashes are masked from end users using transparent component-level retries.

7.3 Applicability to Other Systems

The state separation approach described in this work may also be applied to other microkernel systems which provide isolation for OS services such as L4 and Minix3. This would require some modifications to these kernels to incorporate SSR management and changes to server APIs. These systems would need to also be augmented to support the other requirements for transparent recovery detailed in Section 3. The benefits of state partitioning for operating systems that do not use inter-component isolation is debatable. Since there are no constraints on error propagation, it is difficult to determine which OS subsystem needs to be restarted.

7.4 Additional Benefits

There are several additional benefits of our design. Since memory usage of SSRs can be attributed to clients, they cannot cause a denial of service problem at a server by creating a large number of connections to it. Our design also makes it possible to transparently upgrade a server by simply terminating the old server and starting a newer version while preserving the SSRs. If the new server can interpret the existing SSRs (backwards compatible), it can continue serving existing clients.

7.5 Drawbacks

While there are several advantages of adopting our approach to OS design, there are also several drawbacks. Apart from the performance and memory overheads quantified in Section 6, there is still the added complexity involved in separating state from services and hopefully not introducing new software faults (bugs) in the process. We have tried to quantify this additional complexity in terms of lines of code in Section 6. Our observations indicate that it requires about 12-24 person-hours to design and refactor an OS service to work with our framework. This includes the time spent in fixing most bugs uncovered using fault injection.

8 Concluding Remarks

In this paper, we have analyzed some of the reasons why current designs for reliable microkernel operating systems struggle with client-transparent recovery. Through simple fault injection experiments with various systems, we gain insights into properties that are essential for successful client-transparent recovery of OS services. We have described a design for structuring an OS that preserves these properties. CuriOS minimizes error propagation and persists client information using distributed and isolated OS service state to enhance the transparent restartability of several system components. Restricted memory access permissions prevent erroneous OS services from corrupting arbitrary memory locations. Our experimental results show that it is possible to isolate and recover core OS services from a significant percentage of errors with acceptable performance.

The source code for our CuriOS implementation and the code for the QEMU-based fault injector can be found on our website at `http://choices.cs.uiuc.edu/`.

Acknowledgments

We are very grateful for the insights and feedback from Galen Hunt (our shepherd) and the anonymous reviewers. Part of this research was made possible by grants from DoCoMo Labs USA and Motorola as well as generous equipment support from Texas Instruments.

References

[1] 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.

[2] ARM™Integrator Family. `http://www.arm.com/products/DevTools/Hardware_Platforms.html`.

[3] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. E. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 11–33.

[4] BARTLETT, J., GRAY, J., AND HORST, B. Fault Tolerance in Tandem Computer Systems. In *The Evolution of Fault-Tolerant Systems*, A. Avizienis, H. Kopetz, and J.-C. Laprie, Eds. Springer-Verlag, Vienna, Austria, 1987, pp. 55–76.

[5] BARTLETT, J. F. A NonStop Kernel. In *Symposium on Operating Systems Principles* (New York, NY, USA, 1981), ACM Press, pp. 22–29.

[6] BARTLETT, W., AND SPAINHOWER, L. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 87–96.

[7] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track* (April 2005).

[8] CAMPBELL, R. H., JOHNSTON, G. M., AND RUSSO, V. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)". *ACM Operating Systems Review 21*, 3 (July 1987), 9–17.

[9] CANDEA, G., AND FOX, A. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (Lihue, HI, May 2003).

[10] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot – A Technique for Cheap Recovery. In *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 31–44.

[11] CHASE, J. S., LEVY, H. M., FEELEY, M. J., AND LAZOWSKA, E. D. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems 12*, 4 (1994), 271–307.

[12] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The Rio File Cache: Surviving Operating System Crashes. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), pp. 74–83.

[13] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. R. An Empirical Study of Operating System Errors. In *Symposium on Operating Systems Principles* (2001), pp. 73–88.

[14] Compressed ROM filesystem. `http://sourceforge.net/projects/cramfs/`.

[15] DAVID, F. M., AND CAMPBELL, R. H. Building a Self-Healing Operating System. In *Symposium on Dependable, Autonomic and Secure Computing* (Columbia, MD, Sep 2007), pp. 3–17.

[16] DAVID, F. M., CARLYLE, J. C., AND CAMPBELL, R. H. Exploring Recovery from Operating System Lockups. In *USENIX Annual Technical Conference* (Santa Clara, CA, June 2007), pp. 351–356.

[17] DAVID, F. M., CARLYLE, J. C., CHAN, E. M., RAILA, D. K., AND CAMPBELL, R. H. *Exception Handling in the Choices Operating System*, vol. 4119 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 2006.

[18] DAVID, F. M., CARLYLE, J. C., CHAN, E. M., REAMES, P. A., AND CAMPBELL, R. H. Improving Dependability by Revisiting Operating System Design. In *Workshop on Hot Topics in Dependability* (Edinburgh, UK, June 2007), pp. 58–63.

[19] DENNING, P. J. Fault Tolerant Operating Systems. *ACM Computing Survey 8*, 4 (1976), 359–389.

[20] DUNKELS, A. Full TCP/IP for 8-bit Architectures. In *International Conference on Mobile Systems, Applications and Services* (New York, NY, USA, 2003), ACM, pp. 85–98.

[21] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Symposium on Operating Systems Principles* (1995), pp. 251–266.

[22] GU, W., KALBARCZYK, Z., AND IYER, R. K. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 887–896.

[23] HARTIG, H., HOHMUTH, M., AND WOLTER, J. Taming Linux. In *Australasian Conference on Parallel And Real-Time Systems* (Adelaide, Australia, Sept 1998).

[24] HERDER, J., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Roadmap to a Failure-Resilient Operating System. *USENIX ;login 32* (February 2007), 14–20.

[25] 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.

[26] HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P., AND TANENBAUM, A. S. Failure Resilience for Device Drivers. In *International Conference on Dependable Systems and Networks* (2007), pp. 41–50.

[27] HILLER, M., JHUMKA, A., AND SURI, N. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Symposium on Software Testing and Analysis* (New York, NY, USA, 2002), ACM Press, pp. 81–85.

[28] HUNT, G. C., LARUS, J. R., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, 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, 2005.

[29] HYDEN, E. A. *Operating System Support for Quality of Service*. PhD thesis, University of Cambridge, 1994.

[30] JOHANSSON, R. On Single Event Upset Error Manifestation. In *European Dependable Computing Conference* (London, UK, 1994), Springer-Verlag, pp. 217–231.

[31] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., NO, H. M. B., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application Performance and Flexibility on Exokernel Systems. In *Symposium on Operating Systems Principles* (New York, NY, USA, 1997), ACM Press, pp. 52–65.

[32] KENNELL, R., AND JAMIESON, L. H. Establishing the Genuinity of Remote Computer Systems. In *USENIX Security Symposium* (2003), pp. 295–308.

[33] KITTUR, S., ARMAND, F., STEEL, D., AND LIPKIS, J. Fault Tolerance in a Distributed CHORUS/MiX System. In *USENIX Annual Technical Conference* (1996), pp. 219–228.

[34] LEE, I., AND IYER, R. K. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In *International Symposium on Fault-Tolerant Computing* (1993), pp. 20–29.

[35] LESLIE, B., VAN SCHAIK, C., AND HEISER, G. Wombat: A portable user-mode Linux for embedded systems. In *Linux.Conf.Au, (Canberra)* (April 2005).

[36] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GÖTZ, S. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Symposium on Operating Systems Design and Implementation* (San Francisco, CA, Dec. 2004), pp. 17–30.

[37] LIEDTKE, J. On μ-Kernel Construction. In *Symposium on Operating Systems Principles* (New York, NY, USA, 1995), ACM Press, pp. 237–250.

[38] NG, W. T., AND CHEN, P. M. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *International Symposium on Fault-Tolerant Computing* (1999), pp. 76–83.

[39] RANDELL, B. Operating Systems: The Problems of Performance and Reliability. In *IFIP Congress 71 Volume 1* (1971), pp. 281–290.

[40] ROZIER, M., ABROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMAN, F., KAISER, C., LANGLOIS, S., LÉONARD, P., AND NEUHAUSER, W. Overview of the Chorus Distributed Operating System. In *Workshop on Micro-Kernels and Other Kernel Architectures* (Seattle WA (USA), 1992), pp. 39–70.

[41] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and Implementation of the Sun Network Filesystem. In *USENIX Conference* (Portland, OR, USA, 1985), pp. 119–130.

[42] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Symposium on Operating Systems Design and Implementation* (New York, NY, USA, 1996), ACM, pp. 213–227.

[43] SHAPIRO, J. S. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[44] The Standish Group. TCO in the Trenches 2002. `http://www.himalaya.compaq.com/object/TCO.html`.

[45] SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering Device Drivers. In *Symposium on Operating Systems Design and Implementation* (2004), pp. 1–16.

[46] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *Symposium on Operating Systems Principles* (New York, NY, USA, 2003), ACM Press, pp. 207–222.

[47] TANENBAUM, A. S., HERDER, J. N., AND BOS, H. Can We Make Operating Systems Reliable and Secure? *IEEE Computer 39*, 5 (2006), 44–51.

[48] Texas Instruments OMAP Platform. `http://focus.ti.com/omap/docs/omaphomepage.tsp`.

[49] TORRES-POMALES, W. Software Fault Tolerance: A Tutorial. Tech. Rep. NASA/TM-2000-210616, NASA Langley Research Center, 2000.

[50] ÚLFAR ERLINGSSON, VALLEY, S., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software Guards for System Address Spaces. In *Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 75–88.

[51] VENEMA, W. Murphy's law and computer security. *USENIX Security Symposium* (1996), 187.

[52] WANG, L., KALBARCZYK, Z., GU, W., AND IYER, R. K. An OS-level Framework for Providing Application-Aware Reliability. In *IEEE Pacific Rim International Symposium on Dependable Computing* (2006).

[53] WELCH, B. B. *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System*. PhD thesis, University of California, Berkeley, CA 94720, Feb. 1990. Technical Report UCB/CSD 90/567.

[54] WITCHEL, E., CATES, J., AND ASANOVIĆ, K. Mondrian Memory Protection. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ACM Press, pp. 304–316.

[55] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARRE, M., NECULA, G., AND BREWER, E. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Symposium on Operating Systems Design and Implementation* (Nov 2006), pp. 45–60.