Building a Self-Healing Operating System

Francis M. David, Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N Goodwin Ave, Urbana, IL 61801
{fdavid,rhc}@uiuc.edu

Abstract

User applications and data in volatile memory are usually lost when an operating system crashes because of errors caused by either hardware or software faults. This is because most operating systems are designed to stop working when some internal errors are detected despite the possibility that user data and applications might still be intact and recoverable. Techniques like exception handling, code reloading, operating system component isolation, micro-rebooting, automatic system service restarts, watchdog timer based recovery and transactional components can be applied to attempt self-healing of an operating system from a wide variety of errors. Fault injection experiments show that these techniques can be used to continue running user applications after transparently recovering the operating system in a large percentage of cases. In cases where transparent recovery is not possible, individual process recovery can be attempted as a last resort.

1 Introduction

The reliability of computer systems is an increasingly important issue in the modern world. Complex computer systems govern most of our daily lives. The operating systems that manage critical applications on these computers need to cope with a growing number of software bugs, malicious attacks and hardware faults. Resilience to errors is an important requirement of modern operating systems. The serious nature of this problem has fueled significant amounts of recent research [33, 32, 34].

When most operating systems encounter critical errors in hardware or software, they immediately stop operation, resulting in a loss of currently running user applications and data. Windows blue screen errors [5] and kernel panics in UNIX systems are well known examples of such behavior. We argue that this is unacceptable because the user is only concerned with applications and associated data, which might still be intact and recoverable. This paper explores changing this fail-stop behavior to allow for the system to heal itself, at least partially, in order to prevent loss of user data.

Our goal is to recover an operating system (OS) transparently to user applications after an internal failure. In this paper, we present and evaluate a set of error detection and recovery techniques that can be used to provide self-healing support to an OS.

Several approaches may be used to detect OS errors. Some errors in an OS are detected and signaled [2] by the processor. Processor exceptions normally signal errors such as incorrect memory alignment, invalid opcodes and virtual memory access control violations. These errors can be due to hardware or software faults. OS errors such as entering infinite loops with interrupts disabled result in the system continuing to run without performing any useful work. Such lockup errors can only be detected using external hardware such as watchdog timers. Lockup causing bugs also occur often in OS code. More than 30% of the bugs in Linux discovered by Chou et al. [10] were bugs that could potentially cause a lockup.

Increased developer control over error handling using language supported exception handling, code reloading, OS component isolation, component micro-reboots, automatic system service restarts, watchdog based recovery and transactional components allow an OS to recover from a wide variety of errors. A process-level recovery approach which recovers individual user process state can be used as a last resort when all other attempts at transparent recovery fail. A classification of the error management techniques described in this paper is shown in figure 1.

Is the state of the system correct after recovering from an OS error? Some types of errors are simple, easily detected and fixed by techniques such as code reloading. The system is restored to a correct state in these cases. Because of the nature of some complex errors, and unknown extents of error propagation, it is not possible to guarantee correctness of the system after recovering from them. But this

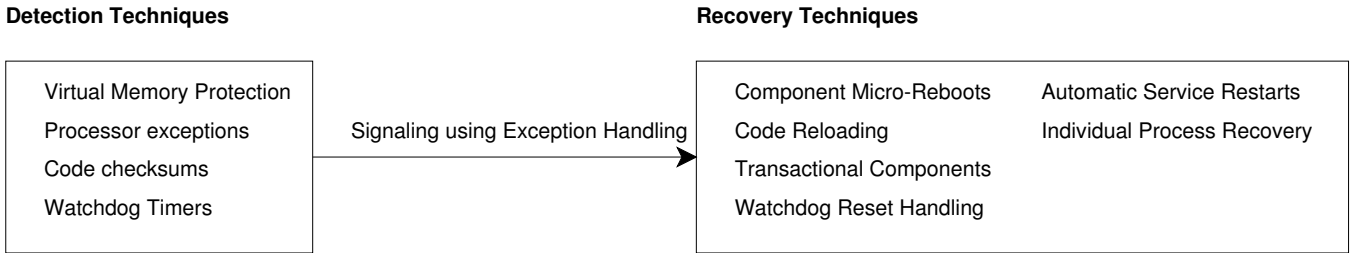

Figure 1. Operating System Error Management Techniques

should not be a deterrent to attempting recovery. For example, in the Linux kernel, an error condition known as an "Oops" is usually resolved by terminating the process that encountered the error. An "Oops" error only halts the system by causing a kernel panic if the error occurred when an interrupt was being serviced. The Linux kernel thus assumes process-level fault containment. This is not enforced within the kernel and may not always be true. Our recovery techniques make similar reasonable assumptions about errors and only recover correctly for the fault models they are designed to handle. Nevertheless, after recovering from an error, we advocate notification to the user that the system might be unstable and should be restarted after saving work.

Our research is currently targeted at the reliability of operating systems that power mobile cellular devices. Our ideas have been implemented on prototype cellphone hardware based on the ARM architecture; however, our recovery solutions are not architecture specific and are generic and widely applicable. We have implemented and evaluated our recovery techniques in the Choices objected-oriented research OS [7]. We have also implemented watchdog based recovery and process-level recovery in Linux.

Information about our research into exception handling, automatic restarts, micro-reboots and watchdog recovery can also be found in previously published work [13, 12] and we only briefly discuss these topics. Our new contributions in this paper include

1. A survey of techniques that can be applied to provide self-healing functionality to an OS and the corresponding fault models that they address.

2. Exploring the use of code-reloading as a reactive recovery strategy when errors are detected in OS code.

3. Exploring transactional objects and exception handling as a mechanism to roll-back erroneous state within OS components.

4. Outlining individual process checkpointing and restoration as a last resort when all other recovery approaches fail.

The remainder of this paper is organized as follows. In section 2, we describe error signaling in Choices using exceptions and the component isolation support for error confinement. Section 3 presents the techniques we have explored for OS error detection and recovery. We evaluate some of these techniques in section 4. We discuss related work in section 5 and conclude in section 6. This paper is an extended version of an internal technical report [11].

2 Error Signaling and Confinement

2.1 Exception handling

Exception handling is commonly used to signal error conditions in application code. The use of exception handling has also been explored in the Linux kernel [19]. This provides the ability to raise and handle C++ exception objects within the OS. In addition to supporting standard C++ exceptions, Choices has support for mapping processor exceptions to C++ language exceptions [13]. This allows system developers to write code to handle errors like null pointer dereferences and illegal opcodes in the OS using the C++ "catch" construct. Converting processor exceptions into language exceptions and allowing them to be handled by system code results in a uniform framework for error signaling. This provides developers a flexible and powerful technique to manage errors. Instead of providing generic handlers that just print out an error message and halt the system, local exception handlers can provide a more appropriate response and attempt to recover the system.

Performance concerns are no longer an issue when using exception handling. Our previous research has shown that if the compiler implements exception handling using modern table-based techniques instead of older context saving techniques, there is no noticeable impact on performance [13]. The need for the compiler to store exception dispatching tables results in some space overhead. But with the large amount of memory available in modern systems, this is not a significant concern either.

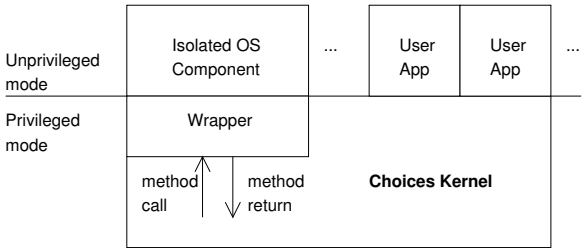

Figure 2. OS component isolation using wrappers

2.2 Component isolation

Component isolation helps contain the propagation of an error. If the error is contained within a component, recovery can be targeted toward the affected component. This technique has been investigated for monolithic operating systems by the Nooks project [33, 32] using virtual memory based isolation and by the OKE project [6] using a safety enforcing trusted C compiler. This property is inherent for micro-kernel operating systems such as Minix3 [20] and L4 [23].

We have implemented support for component isolation using virtual memory protection in Choices. Isolated components are provided with read-write access to defined memory regions which include a private stack and a private heap. If the component encapsulates a device driver, it is additionally granted access to its associated memory-mapped hardware. The rest of the kernel is marked read-only and is therefore protected from corruption caused by errors in the component. Unlike the Nooks approach, we execute isolated components in a reduced processor privilege mode for increased effectiveness.

Components are encapsulated by classes in Choices and our implementation uses wrapper objects to manage switching in and out of isolation mode during a method call on an isolated component. This is is illustrated in figure 2. This minimizes the code that needs to be changed in the components. Wrapper classes use multiple inheritance; they inherit generic isolation code from a wrapper base class for code reuse and they also inherit from the wrapped class in order to impersonate it to the rest of the OS. Exception handling also works with isolated components. Wrappers catch all unhandled exceptions in isolated components and re-raise them in the domains of callers if component recovery fails.

We have implemented component isolation for several drivers in Choices. The console driver in Choices has been replaced with a wrapper that provides isolation and delegates work to the real console driver. This required no changes to the original driver. Some filesystem objects and the watchdog timer driver have also been implemented as isolated components. The isolation properties have been verified using various common hand-written programming errors.

Component isolation only provides some error-containment. Any error detected while executing the isolated component causes an exception to be raised. Recovery would require that the exception is appropriately handled.

3 Error Detection and Recovery

3.1 Code reloading

Transient memory faults (bit-flips) or memory corruption because of faulty code can cause errors such as invalid instructions in system code. While ECC memory can help detect and fix some transient hardware bit-flip errors, it cannot handle memory corruption errors caused by incorrect program execution.

Code reloading is a simple and effective technique that can be used to fix such errors in OS code. The recovery strategy involves reloading the erroneous memory word from stable storage such as disk or other non-volatile memory such as flash. If the error is permanent (this can be discovered by testing), it might still be possible to recover by remapping the affected hardware page using virtual memory support.

In Choices, if the processor signals an undefined instruction exception, the handler reloads the instruction from a copy of the code in memory-mapped flash storage and the newly loaded instruction is executed. This recovery strategy is simple to implement; but, it cannot detect memory corruption that results in an opcode changing to another valid opcode.

Periodic code checking can be used to improve detection of memory faults. Hashing and checksums can easily be used to verify signatures of running code and trigger a reload if a fault is detected. This is a preemptive approach and can detect faults before they cause errors. This approach can also detect memory faults that cause an opcode to change to another valid opcode. Choices computes periodic CRC-32 checksums of critical kernel code and ensures that instruction memory has not been corrupted. If the checksum changes due to memory corruption, the affected memory block is reloaded from flash. The instruction cache is then flushed to ensure that any cached corrupted instructions are discarded. A code checksum may also be performed immediately after an OS error is detected in order to ensure that system and recovery code is intact.

Recent ARM based microprocessor designs [22] for mobile phones include Run Time Integrity Checker (RTIC)

hardware which can be configured by the OS to periodically compute and verify SHA-1 hashes of specified code sections. Any error that is detected is communicated to the processor through an interrupt. This design significantly reduces the performance cost of performing periodic checksums as the external hardware only utilizes the memory bus when it is idle. We do not currently have access to this latest hardware and are therefore constrained to work with checksums performed by the processor.

A similar checksum verification approach can be used to check for the integrity of static data. It is difficult to check the integrity of pages containing changing data. This would require semantic knowledge about the data and therefore require checker routines written by the OS developers. The EROS system [29] uses such an approach to verify the consistency of periodic system-wide checkpoints. We are currently exploring similar support with Choices.

A limitation of this technique is that it cannot be transparently applied to code that is generated at run-time or self-modifying code. In these cases, special care needs to be taken to ensure that a copy of the generated code is saved to stable storage.

3.2 Component micro-rebooting

Micro-reboot has been shown to be an effective recovery technique for application programs [8]. Applying this technique to operating systems is also feasible and can help recover from transient hardware faults and some software bugs. In the Nooks project, micro-reboots in the form of extension restarts were originally used to recover the Linux kernel. In Choices, a micro-reboot involves reinitializing the affected component or destroying and re-creating it and then retrying the request to the component. Micro-rebooting in Choices is supported by the exception handling framework. While code reloading only fixes errors in processor instructions, a micro-reboot fixes errors in kernel data structures. Micro-reboots work together with isolated components and the wrapper objects that provide isolation are also used to manage recovery.

The fault model addressed by this technique is component level fault containment which can be partially enforced by component isolation.

3.3 Automatic service restarts

When a critical OS service such as the paging daemon fails, it grinds the system to a halt. If the failure of such an important process is detected, a simple restart of the service may ensure the continued operation of the OS. The fault model addressed by this technique is single process failure with no external state corruption.

In micro-kernel operating systems, this essentially involves detecting and restarting failed system services which are run as user processes. For example, in Minix3, this job is performed by the reincarnation server [34]. In Choices, a system process can be created so that it is automatically restarted if it encounters an unhandled exception.

The process dispatcher is a special system process that loops continuously waiting for a ready process and yields to the new process. If the process dispatcher crashes, the system is rendered unusable. Therefore, in Choices the process dispatcher is implemented as a restartable process that is always recovered if it crashes.

Simple process restarts may not always work if the process uses locks to access shared data structures. This happens when the process dies when holding one or more locks. Assuming that the shared data structures are not corrupted or that they can be checked for correctness and fixed [15], system recovery is only possible if all held locks are released. For this reason, Choices tracks all locks held by a process and forcibly releases any held locks when a process is terminated. We have also implemented lock tracking and forced unlocking for some types of Linux locks.

3.4 Watchdog-based recovery

External hardware watchdog timers are used to detect errors where the OS is not performing any useful work and is in an infinite loop. A watchdog timer has to be periodically reset (kicked) by the OS and will signal the processor (bite) if the timer expires. Watchdog timers are normally wired to the reset pin on the processor and cause a full reboot of the system for recovery. Unfortunately, a reboot of the system results in a loss of user data and applications currently in memory.

By taking advantage of the fact that volatile memory is still preserved after a processor reset, we can reconstruct both OS and user state and continue to execute even after the reset. This novel approach avoids complete loss of user data and results in increased reliability.

We have implemented watchdog-based recovery in both Linux and Choices [12]. When the watchdog bites, the processor, the memory management unit (MMU) and interrupt subsystem are reset. Our modified reset handler skips the normal boot sequence if the reset is initiated by a watchdog timer. The handler turns the MMU back on, deactivates the process that was running when the reset was issued, reinitializes the interrupts and jumps to the operating system's process dispatch loop, which picks up the next ready process and runs it. The only information that is lost is the state of the process that was running when the processor was reset. This process cannot be scheduled again and is removed from the process queue. We have also explored resolution of the lockup condition by delivering an exception to the

locked up thread. This allows for the thread to attempt local recovery instead of being forcibly terminated.

The fault model assumed by this technique is also single process failure with no external state corruption. Watchdog recovery makes use of the lock tracking code described in the previous section to release shared resources held by the process that is deactivated.

Deadlocks are a special type of lockups that can also trigger a watchdog timeout. Cycle detection in resource wait-for graphs can help reduce deadlock error detection latency. This support is expected to be added to Choices soon. Recovery may be attempted by restarting some components in order to break cycles in the graph.

3.5 Transactional Components

We are currently exploring the use of software transactional memory techniques to provide transaction semantics to objects representing OS components in Choices. When an error causes an exception to be raised during a transactional operation on the component, the state of the component can be rolled back by aborting the transaction. The operation is then re-tried.

We have integrated parts of the RSTM [24] C++ transactional object library into Choices in order to provide transaction support. This is designed to work in conjunction with isolated components in order to minimize error propagation. Transaction management is performed by the same wrapper objects that provide isolation. The wrapper aborts the transaction and rolls-back component state if there is an unhandled exception. We are currently only using the roll-back capabilities of the RSTM library. Additional benefits such as multi-threaded and non-blocking execution provided by RSTM can be potentially used to improve performance, but are not yet exploited by Choices.

Supporting a transactional model on components incurs overheads in terms of both space and time. Space overheads are due to the need to store backup copies of component state before transactions commit. Time overheads arise from the need to perform memory copies and memory management tasks when setting up and committing a transaction.

This approach is different from component micro-rebooting because only the current transaction is rolled back. Micro-rebooting re-initializes the complete internal state of the component. Depending on the type of component, either micro-rebooting or transaction roll-back can be deployed. In particular, if the component includes significant state information that will be lost through micro-rebooting, transactions can be used to ensure that this state is recovered. Micro-rebooting can be used when the component can tolerate state reinitialization and has negligible space and time overheads.

The fault model addressed by this approach is arbitrary memory corruption within an OS component that is detected and signaled by an exception before a transaction on the component commits.

3.6 Process-level recovery

If transparent recovery is not possible, or if the recovery process itself encounters errors, individual process state can be saved to stable storage as a last resort. After user processes are saved, a normal full reboot may be attempted and the state of the processes can be selectively restored on the computer. All OS state is re-initialized after the reboot, potentially eliminating transient errors.

This technique ensures that all user application state is not lost when the error only affects a few applications or irrelevant OS state. This can also be combined with filesystem snapshots to ensure that file integrity is not compromised after recovery by continuing to run user processes that may have errors.

This only requires minimal support from the OS: a functioning non-volatile storage driver and user process state management code. These can be reloaded from stable storage if their integrity is in doubt. This can be easily implemented in Linux with process state checkpointing software [27, 18]. We have included support from the CRAK project [36] for checkpointing all user processes in the Linux kernel. The processes can be selectively restored after a reboot. Currently, the code requires the user to issue an explicit process save request. Ideally, this should be automatically done after attempts at transparent recovery have failed. Choices does not yet include support for process-level recovery.

The fault model addressed by this recovery technique is arbitrary OS corruption not affecting user process state and process recovery code.

4 Evaluation

All of the proposed recovery extensions have been implemented on the Texas Instruments OMAP1610 H2 prototype cellphone hardware [35] and also on a virtual hardware platform based on the ARM Integrator [1] board emulated by the QEMU [4] software. In order to perform some fault injection studies, we built a fault injector based on QEMU capable of injecting faults into memory, hardware registers and raising processor exceptions. In this section, we describe our experiences with code-reloading, automatic restarts, watchdog based recovery and transactional components.

4.1 Code Reloading

To test the effectiveness of code-reloading, we injected 100 random memory corruption faults into CRC monitored regions holding Choices interrupt vectors and handling code. This simulates errors due to transient memory bit-flips or software bugs in drivers. Only one fault is injected in every experiment. 85 faults were corrected by the periodic CRC checking support, avoiding a possible future failure. 4 of the faults caused an undefined instruction interrupt and were automatically corrected. Only 11 faults caused a kernel crash. These crashes occurred because errors were encountered before the periodic (5 seconds) CRC check could detect and fix the faults. This shows that code-reloading is quite effective in reducing the number of faults that can be attributed to corrupted OS code.

The effectiveness of this technique also depends on the check interval. A smaller interval between checks can catch more faults before they cause errors. But, frequent checking results in reduced performance. It may be possible to adaptively change the period based on the current error rate and a given performance degradation threshold. This is a topic for further research.

4.2 Automatic Restarts

Automatic process restarts, especially when applied to critical kernel processes also provide significant improvements in reliability. In a fault injection experiment that was performed 1000 times, a random processor exception was introduced while the process dispatcher was running. We found that automatically restarting the dispatcher resulted in recovery 78.9% of the time. The failures are due to exceptions being raised during updates to critical data structures, thus causing irreparable corruption.

4.3 Watchdog-based Recovery

Our Linux watchdog recovery implementation was tested by manually writing a device driver that spawns a buggy kernel thread. The introduced bug causes the thread to eventually enter into a state in which it enters an infinite loop with interrupts turned off. Without an external watchdog, this causes the kernel to lock up and hangs the system. One of our tests consists of a script that runs the bzip2 decompression algorithm on a compressed file as a user process and instructs the device driver to spawn the buggy kernel thread. The decompression is interrupted by the buggy thread which crashes the kernel. With watchdog based recovery support turned on, the kernel recovers as soon as the processor is reset and the decompression runs to completion. In all of our experiments, the decompression was verified to be correct. In another test, we cause the kernel to crash after allowing a user to open a text editor and start to type text into it. With watchdog based recovery, the kernel is able to recover after the processor reset and the user is able to continue editing the text and is eventually able to save the file to stable storage. It should be noted that the recovery is perfect in these cases because the bug does not corrupt external kernel state. Similar experiments in Choices also result in complete recovery after a kernel hang.

We also performed automated fault-injection experiments on Linux and Choices to study their recoverability from watchdog detected lockups. We measured recovery rates of about 70% in these experiments. A more detailed analysis of this study is available in an earlier paper [12].

4.4 Transactional Components

In order to study the effectiveness of transaction support, we performed random register bit-flip fault injection experiments into isolated test components. Component state was rolled back correctly in all (100%) of the experiments whenever an exception was raised.

Execution time overhead when compared to using components not using transactions is around 0.05-0.10 milliseconds at a ARM CPU frequency of 96MHz when the size of the component is less then 1024 bytes. This overhead is due to memory allocation and copying. Hardware acceleration can be used to alleviate this overhead [30]. A more comprehensive deployment and study of transactional memory support is currently in progress.

5 Related Work

There is a plethora of work in hardware and software fault-tolerance. Fault-tolerance in operating systems has been studied over several decades [28, 16]. Our recovery approaches are complementary to a large body of work in OS error detection using both hardware and software. Language based techniques are used by SafeDrive [37] to detect errors in the Linux kernel. Such techniques are complementary to our approaches and can help reduce error detection latencies. There is also some work in detecting infinite loop errors in OS code using experimental processor extensions [25]; however, recovery is not addressed. Self-checking code has been used to detect changes to running user applications [21].

Arjuna [26] is a middleware system that supports transaction like semantics on objects and thus provides failure atomicity. Exception handling was however not used in Arjuna as it was not supported by the language at that time. We support transaction semantics on objects within the OS kernel.

There is also some directly related work in application recovery after OS crashes. The recovery box approach [3] uses non-volatile memory to store application state that is restored when the system is restarted after a crash. Researchers at Rutgers have used remote-DMA in order to access the memory of a crashed system and recover application state [31]. In the Rio filesystem, filesystem buffer cache state is recovered after an OS crash [9]. In contrast to all these approaches, we try to self-heal the entire operating system after an error.

Checkpointing can be used to recover from crashed systems running in virtual machines. VMWare and Xen [17] provide mechanisms to checkpoint running operating systems and restore them. When the OS crashes, the checkpoint can be restored, thus providing limited recovery. EROS [29] provides similar support with a built-in checkpointing framework. Compared to these approaches which lose all information after the checkpoint, our recovery techniques attempt to recover currently running processes.

6 Conclusions and Future Work

Our experiments demonstrate that it is possible to build self-healing operating systems through simple and effective techniques such as code reloading, component isolation and automatic restarts. While micro-kernels are well suited for fault-tolerant operation because of their architecture with inherent isolation, it is also possible to reap similar benefits in a monolithic kernel built from isolated components. With the addition of external watchdog hardware support, it is also possible to detect and attempt recovery from system hangs that would otherwise remain undetected.

The generic recovery techniques described in this paper can be improved by incorporating support for a framework that allows the use of developer specified policies that govern recovery actions on a case-by-case basis. We have experimented with some rudimentary support for specifying simple policies like retry counts for micro-reboots and automatically restartable processes. But there is a need for the ability to specify more complex recovery actions that can take into account dependencies and OS state checking routines. This may increase the chances of a successful recovery.

Micro-reboots and service restarts can lose state information which might be necessary for successful recovery. While transactional components provide some form of state restoration, we are working on obtaining further improvements in dependability through re-organization of OS state [14].

More details, reports and code related to our implementations of the recovery techniques described in this paper and the code for the QEMU based fault injector are available online at `http://choices.cs.uiuc.edu/`.

Acknowledgments

We thank the anonymous reviewers for their invaluable feedback. Part of this research was made possible by grants from DoCoMo Labs USA and generous support from Texas Instruments. We would also like to thank Professor Ravishankar K. Iyer for helpful discussions during his class on fault-tolerant systems. Daniel Chen helped implement part of the Linux recovery code. Ganesh Bikshandi, Jia Guo and Justin Trobec helped implement parts of the component isolation code in Choices. Transactional component support was added to Choices by Winson Chan, Jin Heo and Changyoung Jung. Ramkumar Vadali and Shankar Kalyanaraman helped integrate code from CRAK for checkpointing user processes in the Linux kernel. This work also benefited significantly from discussions with Ellick Chan and Jeffrey Carlyle.

References

[1] ARM Integrator Family. `http://www.arm.com/miscPDFs/8877.pdf`.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[3] M. Baker and M. Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *USENIX*, pages 31–44, Summer 1992.

[4] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, April 2005.

[5] Blue Screen. `http://support.microsoft.com/kb/q129845/`.

[6] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *IEEE Open Architectures and Network Programming*, 2002.

[7] R. H. Campbell, G. M. Johnston, and V. Russo. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)". *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[8] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[9] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[10] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An Empirical Study of Operating System Errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[11] F. M. David and R. H. Campbell. Recovering from Operating System Errors. Technical Report UIUCDCS-R-2007-2831, University of Illinois at Urbana-Champaign, March 2007.

[12] F. M. David, J. C. Carlyle, and R. H. Campbell. Exploring Recovery from Operating System Lockups. In *USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[13] F. M. David, J. C. Carlyle, E. M. Chan, D. K. Raila, and R. H. Campbell. *Exception Handling in the Choices Operating System*, volume 4119 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 2006.

[14] F. M. David, J. C. Carlyle, E. M. Chan, P. A. Reames, and R. H. Campbell. Improving Dependability by Revisiting Operating System Design. In *Workshop on Hot Topics in Dependability*, Edinburgh, UK, June 2007.

[15] B. Demsky and M. Rinard. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the First Workshop on Algorithms and Architectures for Self-Managed Systems*, San Diego, California, June 2003.

[16] P. J. Denning. Fault Tolerant Operating Systems. *ACM Comput. Surv.*, 8(4):359–389, 1976.

[17] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[18] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, 2003.

[19] H. I. Glyfason and G. Hjalmtysson. Exceptional Kernel: Using C++ Exceptions in the Linux Kernel, October 2004. `http://netlab.ru.is/exception/KernelExceptions.pdf`.

[20] J. N. Herder. Towards a True Microkernel Operating System. Master's thesis, Vrije Universiteit Amsterdam, 2005.

[21] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. In *Digital Rights Management Workshop*, pages 141–159, 2001.

[22] i.MX31 Multimedia Applications Processor. `http://www.freescale.com/files/32bit/doc/ref_manual/MCIMX31RM.pdf`.

[23] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM Press.

[24] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006.

[25] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu. An Architectural Framework for Providing Reliability and Security Support. In *DSN*, pages 585–594. IEEE Computer Society, 2004.

[26] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna. *Computing Systems*, 8(2):255–308, 1995.

[27] E. Pinheiro. Truly-Transparent Checkpointing of Parallel Applications, 1998. `http://www.research.rutgers.edu/~edpin/epckpt/`.

[28] B. Randell. Operating Systems: The Problems of Performance and Reliability. In *Proceedings of IFIP Congress 71 Volume 1*, pages 281–290, 1971.

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

[30] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware Acceleration of Software Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006.

[31] F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I. Neamtiu, and L. Iftode. Recovering Internet Service Sessions from Operating System Failures. *IEEE Internet Computing*, 9(2):17–27, 2005.

[32] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Symposium on Operating Systems Design and Implementation*, pages 1–16, 2004.

[33] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 207–222, New York, NY, USA, 2003. ACM Press.

[34] A. S. Tanenbaum, J. N. Herder, and H. Bos. Can We Make Operating Systems Reliable and Secure? *Computer*, 39(5):44–51, 2006.

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

[36] H. Zhong and J. Nieh. CRAK: Linux Checkpoint/Restart as a Kernel Module. Technical Report CUCS-014-01, Columbia University, November 2002.

[37] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harre, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Symposium on Operating Systems Design and Implementation*, Nov 2006.