Exploring Recovery from Operating System Lockups

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

Abstract

Operating system lockup errors can render a computer unusable by preventing the execution other programs. Watchdog timers can be used to recover from a lockup by resetting the processor and rebooting the system when a lockup is detected. This results in a loss of unsaved data in running programs. Based on the observation that volatile memory is not affected when a processor a reset occurs, we present an approach to recover from a watchdog reset with minimal or zero loss of application state. We study the resolution of lockup conditions using thread termination and using exception dispatch. Thread termination can still result in a usable system and is already used as a recovery strategy for other errors in Linux. Using exceptions allows developers to write code to handle a lockup within the erroneous thread and attempt application transparent recovery. Fault injection experiments show that a significant percentage of lockups can be recovered by thread termination. Exception handling further improves the recoverability of the operating system.

1 Introduction

While many techniques have been invented over the years to create software that is resilient to faults [1], errors due to hardware and software faults still remain a serious problem in today's world. Some errors can cause the processor to lock up in an infinite loop of useless computation. In this case, the error can only be detected by an external entity. Lockup errors that happen in user programs can be detected by other programs [2] and can usually be handled without affecting unrelated programs. On the other hand, lockups that occur inside the operating system (OS) can render the computer unusable by not allowing any other programs to execute. Lockup causing bugs are common in OS code. More than 30% of the bugs in Linux discovered by Chou et al. [3] were bugs that could potentially cause a lockup.

Watchdog timers have been traditionally used to detect lockups in OS code and are usually configured to reset the processor when the timer expires. To prevent a processor reset and a consequent reboot, the OS must periodically reset the watchdog timer. It is common to refer to watchdog expiration as a bite and to the act of resetting the watchdog as a kick. While rebooting after a lockup improves availability, it results in a loss of all running user programs and data.

In this paper, we demonstrate that this reboot behavior can be replaced by an approach where the reset signal to the processor is used to recover the system with minimal or zero loss of application state. The key observation that motivates our approach to OS recovery is that the reset signal only affects the processor and leaves volatile memory intact. Information loss is limited to the contents of the processor at the time of the reset and the contents of volatile memory can be used for recovery.

We have implemented watchdog based recovery in Linux and in the Choices object-oriented OS [4]. We explore recovery after a watchdog bite using two methods: by terminating the locked up thread (in Linux and Choices), and by dispatching a C++ exception to the thread (in Choices).

Using thread termination is a simple approach to recovery. There is no attempt to prevent or fix possible kernel data structure inconsistencies. Therefore, there are no guarantees that the system is successfully recovered. However, attempting to recover a crashed system by terminating a thread is not uncommon. Operating systems such as Linux already respond to kernel space errors like invalid pointer dereferencing by terminating the erroneous thread. We, therefore, apply this approach to lockup errors as well. Experiments with Choices and with Linux demonstrate that recovery is possible from a wide variety of OS lockups when using thread termi-

Part of this research was made possible by grants from DoCoMo Labs USA and generous support from Texas Instruments.

nation. We consider a recovery attempt to be successful if the OS continues to schedule and run other existing threads and provides some minimal functionality like filesystem and console access.

In Choices, we are exploring the use of the C++ exception mechanism as a unified framework for notification of all errors that occur within the OS. We create exceptions from errors like memory faults, invalid instructions and hardware aborts and allow threads to respond to them using exception handlers [5]. We were therefore motivated to add OS lockups to the set of exceptions already handled by the Choices kernel.

Existing techniques such as Nooks [6] and SafeDrive [7] do not attempt to recover from lockup errors within extensions. Our watchdog timer based recovery approach complements both these techniques and enables them to recover from a larger class of errors. OKE [8] can detect and recover lockup errors in extensions compiled with a safety enforcing trusted C compiler. Our recovery approach does not require a special compiler and works with existing code. Also, these systems only consider errors in device drivers. Thread termination and exception dispatch can be used to recover from lockup errors in other parts of the OS as well. For example, a lockup in a non-preemptable system call handler is not detected by Nooks, SafeDrive, or OKE; but is detected and potentially recovered by our techniques.

Our recovery implementations have been evaluated on two ARM processor based platforms: the Texas Instruments OMAP1610 H2 hardware development kit and the QEMU [9] system emulator.

While the traditional action of a watchdog has been to reset the system on a watchdog bite, an alternative would be to raise a non-maskable interrupt (NMI). Current ARM processors do not support non-maskable interrupts. Nevertheless, we still examine the advantages of using a non-maskable interrupt to notify the processor of a watchdog bite.

2 Watchdog Recovery Design

A careful analysis of OS behavior is required when deciding when to execute watchdog kicks. Placing the kick code in the timer interrupt handler ensures that, as long as interrupts are enabled, and there is no lockup in the interrupt handler, the watchdog will not bite. However, it is possible that a lockup can occur with interrupts still enabled. If the lockup is in a non-preemptable section of code, the OS is unusable because it does not schedule any other threads. In Linux, watchdog timers are exported as devices to userspace and the kicks are issued periodically by a userspace thread. If the userspace thread does not get scheduled because of an OS lockup, the watchdog reboots the system. Kicks issued from threads are a more effective indication of the system being alive than kicks issued from timer interrupts.

When the ARM processor is reset, it switches to a privileged execution mode and sets its program counter (PC) to address 0. The signal also resets the interrupt controller and all interrupts are turned off. The memory management unit (MMU) is also turned off and only physical addressing is possible. Address 0 is normally the start address of the bootloader. The bootloader's job is to initialize the memory hardware and load the OS kernel into RAM from flash memory or secondary storage. It then relinquishes control to the OS. The bootloader usually does not differentiate between resets attributed to watchdog timers and power-on resets. Thus, the OS is always reloaded and rebooted, causing a loss of all running programs and data in memory.

In order to ensure that memory contents are preserved, the bootloader needs be modified to treat the watchdog bite differently. When the watchdog bites, the bootloader should not reload the kernel and should instead directly transfer control to the OS start address in memory. This is a reasonable approach because, once it is up and running, the OS core is never paged out and resides in the same physical memory area into which it was first loaded.

Once control is back in the OS, a recovery routine can take over. The recovery process involves switching the MMU back on, initializing the interrupt controller, re-enabling interrupts and performing an appropriate action to eliminate the lockup condition before starting to schedule threads again.

There are a couple of issues that arise when attempting to recover from watchdog bites that reset the processor. This requires that the processor cache is configured as write-through instead of write-back in order to avoid loss of cached data. Thus, when using this technique, we gain increased reliability at the expense of some decreased performance. Also, part of the processor context at the time the watchdog bites is lost forever. For example, the program counter is instantaneously overwritten by 0. This makes it difficult to accurately pinpoint the location of the lockup and debug the error. Both these issues cease to exist if the watchdog timer is wired to a non-maskable interrupt. This would enable the OS to respond to the lockup without any loss of information in the processor or the cache. Additionally, using an NMI simplifies the recovery implementation because the MMU and interrupt controllers are not disturbed.

3 Recovering Linux

Soft Lockup Detector: A soft lockup is an error condition where a thread is locked up in kernel mode with

interrupts enabled. Some soft lockups can render the system unusable by permanently preempting all other threads. The Linux kernel includes code that detects these kinds of soft lockup errors. A low priority thread updates a timestamp every second. This timestamp is checked during a timer interrupt to see if it was updated within the last ten seconds. This ensures that the system is usable by confirming that the watchdog thread is periodically scheduled. If the check fails, the detector displays a message reporting the lockup error and records it in the system logs. The detector does not attempt to fix the error.

In order to study the recoverability of the Linux kernel from a lockup detected by the soft lockup detector, we added code to terminate the thread which has locked up in kernel mode. Linux already handles most errors that are encountered within the kernel by terminating the thread. These are usually called "Oops" errors. However, if an "Oops" occurs in interrupt mode, the error is deemed to be serious and the "Oops" handler calls `panic()` which halts the system. Kernel code can also directly call `panic()` on detecting a serious error. We do not attempt to recover from Linux kernel panics.

The soft lockup detector cannot detect lockups that occur when interrupts are disabled because the detector code is not executed. These "hard" lockups can only be detected using an external hardware watchdog timer.

Hardware Watchdog: We added a new kernel thread that wakes up periodically and kicks the watchdog timer. If this thread is not scheduled periodically, the processor is reset. A normal power on reset causes the bootloader to load a compressed kernel image into RAM and transfer control to the header in the compressed image. The header then runs a decompression routine which places the kernel at some platform dependent physical address. The kernel is always resident at this physical address. We modified the small bootloader built into QEMU so that it does not reload Linux and instead directly jumps to the start address of the existing uncompressed kernel when a reset is generated by the watchdog timer.

We modified the first few instructions in the Linux boot up code to check for the reset reason. If the reset was due to the watchdog timer, a recovery routine is executed. The MMU is turned on first with the page tables configured for kernel tasks. Switching on virtual memory ensures that all kernel data structures are visible again. The task that was running at the time of the watchdog bite is then terminated. In the next stage, peripheral interrupts and the watchdog timer are re-enabled. The code then enters the processor idle loop which works as a dispatcher for runnable threads. Recovery is completed once the idle loop begins picking up runnable threads and scheduling them on the processor.

We do not need to worry about locks held by the thread when it is terminated because our target platform is a uniprocessor. Linux implements spin locks on uniprocessors by disabling interrupts for the duration that the lock is held. Thus, if a thread locks up when holding a spin lock, it can only be detected by a watchdog timer. The lock is implicitly released after recovering from a watchdog bite. On multi-processor hardware, lock usage tracking functionality is required in order to release all locks held by the thread when it is terminated. This can be implemented easily by modifying the spin lock functions or by using a code rewriting approach [7]. Usage of semaphores in the locked up thread can present some problems with recovery. We expect tracking semaphore usage to improve chances of successful recovery, but we have not yet explored this direction.

It is also possible that kernel data structures are left in an inconsistent state after a thread is abruptly terminated. This might be unacceptable in high integrity systems. Data structure usage tracking techniques such as those used in Nooks can help mitigate this issue. Fixing or preventing kernel data structure corruption is itself a significant challenge and we do not address it in this work.

These issues with locking, semaphore usage and data structure corruption are identical to those that occur when Linux encounters "Oops" errors. The default response in Linux is to terminate the thread without worrying about any of these issues. Thus, lock and semaphore tracking can also improve recoverability in this case.

As described in section 2, the use of an NMI allows for improved performance and improved debugging support. Some recent x86 interrupt controllers can be configured to generate periodic non-maskable interrupts to the processor. The x86 version of the Linux kernel includes support for lockup detection which exploits this functionality. Similar to the soft lockup detector, the NMI driven detector displays an error message when it detects a lockup. This support is however not yet available on the ARM platform.

4 Recovering Choices

Hardware Watchdog: Recovery from a processor reset issued by a hardware watchdog has also been implemented in Choices. Choices does not yet support soft lockup detection. Soft lockups do not result in an unusable system because the kernel is fully preemptable.

The watchdog is kicked at every timer interrupt. If timer interrupts are not received because of a hard kernel lockup, the watchdog bites. Just as with Linux, Choices invokes a recovery routine instead of proceeding with normal boot if the reset reason was a watchdog timeout.

The recovery routine pretends to be the idle thread and switches the MMU on and restores interrupts. It then pretends to be the locked up thread and calls `die()` directly.

The recovery procedure differs from our Linux implementation. In Linux, we restored the idle thread and it picks up and kills the locked thread. In Choices, we directly kill the locked thread and this automatically restores the next runnable thread on the processor. Both these approaches are valid and either one can be chosen depending on ease of implementation.

While a thread termination approach might help the kernel to continue scheduling other threads, it might still render the OS unusable because the terminated thread might be a critical kernel thread. We have previously explored the use of C++ exceptions to notify threads of errors they encounter in kernel space [5]. Using exceptions allows threads to attempt local recovery strategies in exception handlers. We were therefore motivated to explore converting a thread lockup condition into a C++ exception.

An exception can only be properly dispatched by the C++ exception handling libraries if the context in which the exception is thrown is correct. Thus, simply writing a C++ throw statement in the recovery routine will not work. We needed a way to recover the context of the locked up thread at the time of the watchdog bite. After some experimentation, we discovered that the processor does not lose the contents of most of its registers when it is reset. The PC is lost because it is reset to 0x0, and the value of the processor status register is also lost. But the contents of all the other registers are preserved.

We modified the bootloader to respond to a watchdog bite by storing the contents of the reset preserved registers before they are clobbered by running the recovery routine. A valid value of PC needs to be recovered for exception dispatch to work. We choose to approximate the value of the PC as the first instruction of the function in which the lockup occurred. In machine code generated by the GNU C++ compiler, the PC is saved on the stack frame in the preamble of every function. We can read the last saved PC from the stack using the recovered stack frame pointer register and use this value. The context is now usable for dispatching an exception. This context is modified so that when it is restored on the processor, it enters a helper function which uses the C++ throw keyword to raise an exception.

Standard C++ try-catch syntax can be used to handle these exceptions. We believe that this is an elegant approach to handling lockup conditions within an erroneous thread in kernel mode. A developer can write an exception handler to try thread-specific recovery strategies if the thread ever locked up. Also, unlike the thread termination approach, lockup exception handling can be used within kernel contexts like the initial interrupt processing code which is not a part of any thread.

Table 1: Lockup detection and recovery for Non-Preemptable and Preemptable Linux (*-Y with enhancement)

Lockup Location	**Software**				**Watchdog**			
	Det?		**Rec?**		**Det?**		**Rec?**	
Interruptible thread	Y	N*	Y	N*	Y	N	Y	N
Non-interruptible thread	N	N	N	N	Y	Y	Y	Y
Interrupt handler	Y	Y	N	N	Y	Y	N	N
Syscall handler	Y	N*	Y	N*	Y	N	Y	N

5 Evaluation

Linux: The 2.6 series of kernels have experimental support for kernel-mode preemption and this affects lockup detection. We, therefore, evaluate our implementations with both a non-preemptable and a preemptable kernel.

We introduced artificial infinite loop bugs into different types of kernel contexts and studied the detection and recovery properties of the kernel software detector and a hardware watchdog with kicks issued by a kernel thread. The thread that is terminated for recovery is a non-critical dummy thread and there is no memory corruption.

Table 1 catalogs our experiences with both a non-preemptable and a preemptable version of the kernel. As expected, the soft lockup detector is unable to detect lockups when interrupts are disabled. In these cases, the watchdog timer is able to detect and recover the system. Linux allows nested interrupts and therefore interrupts are enabled when running an interrupt service routine (ISR). A lockup in an ISR, which is non preemptable, is therefore detectable by the soft lockup detector. Recovery is not possible because Linux does not support termination of a thread executing in interrupt context.

Lockup detection effectiveness is reduced when experimental kernel mode preemption support is turned on. This ensures that the watchdog thread is always scheduled even when a higher priority preemptable thread enters a lockup in kernel mode. This is an unfavorable situation because, even though the system is usable, the locked up thread keeps the processor busy. It is possible to detect such situations by measuring the time spent by a thread in kernel space without yielding. A kernel developer has posted a patch for the x86 architecture that enables the soft lockup code to detect these lockups[1], but this has not yet been included in the mainstream kernel. In the table, entries marked with an asterisk can be changed to "Y" with such an enhancement.

[1] `http://lkml.org/lkml/2005/8/2/216`

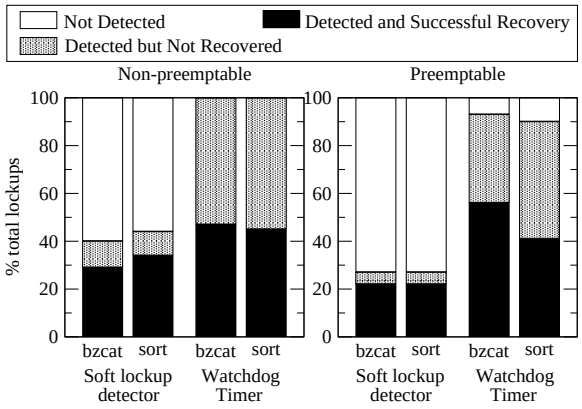

Figure 1: Linux lockup detection and recovery efficiency

When kernel preemption is turned on, the existing soft lockup detector can still detect lockups that occur during a period when preemption is temporarily disabled, resulting in the system being unable to schedule other threads. For example, lockups in interrupt handlers can be detected because these cannot be preempted.

We also performed automated lockup fault injection experiments into various parts of the kernel using a modified QEMU. We randomly pick instruction addresses into which faults are to be injected. A fault is injected by changing the chosen instruction to a self-loop. The fault is transient and is not re-encountered if the instruction is executed again. We inject only one lockup in each experiment. In one set of experiments, faults are injected when running a bzip2 decompression task (bzcat). In another set, faults are injected when running a sort task. Our goal is to examine if lockups in random parts of the kernel affect the successful completion of these user tasks.

We measure the number of lockup detections (using both the soft lockup detector and the watchdog timer) and the number of successful completions of these tasks after recovery is attempted. A successful completion is defined as a run that produces output identical to a run without fault injection. In all our experiments, there are several running background tasks; some of which are standard Linux kernel threads.

The results of our experiments are shown in figure 1. For the non-preemptable version of the kernel, the soft lockup detector detects less than 40% of lockups because most of them occur when interrupts are disabled. The system does not recover when the lockup goes undetected. The watchdog timer detects all the lockups. But, in spite of this increased detection efficiency, the user task only completes correctly in around 50% of the lockups. The reasons for unsuccessful recovery (after detection) vary. Our analysis reveals that between 80-100% of these were because the detection occurred when the kernel was in interrupt context. Since the kernel calls `panic()` when a thread is terminated in interrupt context, the system does not recover.

In the preemptable kernel experiments, there are several lockups that do not cause a complete system crash because they are preempted (not shown in the figure). These are not detected by either the soft lockup detector or the watchdog timer. In 7-10% of the lockups, the tasks complete successfully in spite of the lockup not being detected. These represent the cases in table 1 marked with an asterisk.

For the preemptable kernel, the watchdog has an edge over the soft lockup detector because it can detect lockups when interrupts are disabled.

Choices: The Choices kernel is designed to be preemptable and watchdog timers are only used to detect hard lockups. In order to test our watchdog recovery implementation in Choices, we first inserted artificial lockup bugs into a dummy kernel thread in Choices. Choices is able to recover from bugs in interruptible, non-interruptible and system call handlers by terminating the dummy thread. Unlike Linux, the design of Choices allows it to be recovered from lockups in interrupt handlers as well. Dummy threads and transient lockups were also used to test correct operation of the lockup exception dispatch mechanism in Choices.

We also performed fault injection experiments with Choices in a manner similar to that described for the Linux experiments. We use user tasks represented by a sort program and a gunzip decompression program. Hard lockup errors (infinite loop with interrupts disabled) are injected into the Choices kernel. The watchdog detects all the hard lockups errors that are encountered. Experiments are performed for both the thread termination and the exception handling approaches to recovery. C++ "catch" statements are used in several top level objects to handle exceptions by retrying the request.

Figure 2 compares the recovery capabilities of the

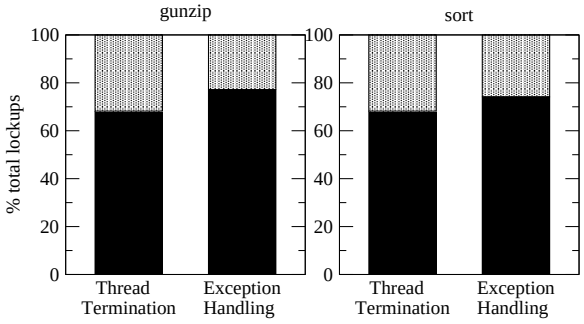

Figure 2: Choices hard lockup recovery comparison

exception based approach with the thread termination based approach. Handling errors using exceptions results in the user task completing successfully from about 6-9% more lockups than when using thread termination. This is because exception handlers in various OS objects attempt to recover the locked thread by retrying the method call that failed and some of these retries are successful. Non-recovered lockups are mostly due to data structures left in an inconsistent state. These results are based on only a few exception handlers in top level objects. We expect that a more thorough deployment of exception handlers throughout the object hierarchies will reduce data structure inconsistency issues that prevent recovery and result in improved recoverability.

6 Discussion and Related Work

In addition to watchdog timers, lockups can also be detected by hardware such as the RSE [10]. RMK [11] detects an OS lockup by counting the number of instructions between two consecutive context switches.

Our approaches to recovery deviate from a fail-stop model of computation in a manner similar to failure oblivious computing [12]. A full system reboot may have to be performed in order to completely recover the system. Our approaches can be combined with techniques like isolation containers [13, 6], microreboots [14] and data structure repair [15] to improve recoverability.

There is some directly related research in application recovery after OS crashes. The recovery box approach [16] uses non-volatile memory to store application state that is restored when the system is restarted after a crash. Remote-DMA can be used to access the memory of a crashed system and recover application state [17]. In the Rio filesystem [18], the buffer cache is recovered from volatile memory after a reset. In contrast to these approaches, we attempt to recover the entire system.

7 Summary and Conclusions

We have discussed detection of operating system lockup errors using software detectors and watchdog timers. We have shown that it is possible to recover from a significant number of such lockup errors by terminating the locked up thread. We have also shown that the use of simplistic retry based exception handling to attempt recovery provides up to a 9% improvement in recoverability. We expect this number to increase with increased deployment of exception handling within the OS.

Additional details and code are available online at `http://choices.cs.uiuc.edu/`.

References

[1] Wilfredo Torres-Pomales. Software Fault Tolerance: A Tutorial. Technical Report NASA/TM-2000-210616, NASA, 2000.

[2] Keith Whisnant, Ravishankar K. Iyer, Zbigniew T. Kalbarczyk, Phillip H. Jones III, David A. Rennels, and Raphael Some. The Effects of an ARMOR-Based SIFT Environment on the Performance and Dependability of User Applications. *IEEE Trans. Softw. Eng.*, 30(4):257–277, 2004.

[3] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An Empirical Study of Operating System Errors. In *SOSP*, pages 73–88, 2001.

[4] 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.

[5] Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, David K. Raila, and Roy H. Campbell. *Exception Handling in the Choices Operating System*, volume 4119 of *LNCS*. Springer-Verlag Inc., 2006.

[6] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. In *OSDI*, pages 1–16, 2004.

[7] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harre, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *OSDI*, Nov 2006.

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

[9] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC, FREENIX Track*, April 2005.

[10] Nithin Nakka, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Jun Xu. An Architectural Framework for Providing Reliability and Security Support. In *DSN*, pages 585–594. IEEE Computer Society, 2004.

[11] Long Wang, Zbigniew Kalbarczyk, Weining Gu, and Ravishankar K. Iyer. An OS-level Framework for Providing Application-Aware Reliability. In *12th IEEE Pacific Rim International Symposium on Dependable Computing*, 2006.

[12] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, pages 303–316, 2004.

[13] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP*, pages 207–222, New York, NY, USA, 2003. ACM Press.

[14] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *OSDI*, San Francisco, CA, December 2004.

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

[17] Florin Sultan, Aniruddha Bohra, Stephen Smaldone, Yufei Pan, Pascal Gallard, Iulian Neamtiu, and Liviu Iftode. Recovering Internet Service Sessions from Operating System Failures. *IEEE Internet Computing*, 9(2):17–27, 2005.

[18] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.