Rx: Treating Bugs As Allergies - A Safe Method to Survive Software Failures

Natalia Khokhlova

March 5, 2007

Abstract

This paper [QTSZ05] proposed an interesting technique, called Rx, which can quickly recover programs from many types of software bugs, both deterministic and non-deterministic. The idea is to rollback the program to a recent checkpoint upon a software failure, and then to re-execute the program in a *modified* environment.

Contents

1 Introduction **1**

2 Motivation **2**

3 Main Idea of Rx **2**

3.1 Rx Design 3
3.2 Sensors 3
3.3 Checkpoint and Rollback 4
3.4 Environment Wrappers 4
3.5 Proxy 4
3.6 Control Unit 5
3.7 Advantages of Rx 5
3.8 Disadvantages of Rx 5

4 Evaluation **6**

5 Related Work **7**

6 Discussion **7**

7 Conclusion **8**

1 Introduction

Many applications require high availability. Unfortunately, software failures greatly reduce system availability. Prior work on surviving software failures suffers from one or more of the following limitations: required application restructuring, inability to address deterministic software bugs, unsafe speculation on program execution, and long recovery time. For this reason, software companies invest enormous effort and resources on software testing and bug detection prior to releasing software. However, software failures still occur during production runs since some bugs inevitably slip through even the strictest testing. Therefore, to achieve higher system availability, mechanisms must be devised to allow systems to survive the effects of uneliminated software bugs to the largest extent possible.

The authors of the paper, OPERAting Systems Research Group of University of Illinois at Urbana Champaign, proposed a safe (not speculatively fixing the bug) technique, called Rx, to quickly recover from many types of software failures caused by common software defects, both deterministic and non-deterministic. It requires few to no changes to applications source code, and provides diagnostic information for postmortem bug analysis. Their idea is to rollback the program to a recent checkpoint when a bug is detected, dynamically change the execution environment based on the failure symptoms, and then re-execute the buggy code region in the new environment. If the re-execution successfully pass through the problematic period, the new environmental changes are disabled to avoid imposing time and space overheads.

2 Motivation

To understand the main idea and mechanism of the proposed Rx technique, the authors compare it with a real-life example. When a person suffers from an allergy, the most common treatment is to remove the allergens from their living environment. For example, if patients are allergic to milk, they should remove diary products from the diet. If patients are allergic to pollen, they may install air filters to remove pollen from the air. Additionally, when removing a candidate allergen from the environment successfully treats the symptoms, it allows diagnosis of the cause of the symptoms. Obviously, such treatment cannot and also should not start before patients shows allergic symptoms since changing living environment requires special effort and may also be unhealthy.

In software, many bugs resemble allergies. That is, their manifestation can be avoided by changing the execution environment. Therefore, by removing the allergen from the execution environment, it is possible to avoid such bugs. For example, a memory corruption bug may disappear if the memory allocator delays the recycling of recently freed buffers or allocates buffers non-consecutively in isolated locations. A buffer overrun may not manifest itself if the memory allocator pads the ends of every buffer with extra space. Uninitialized reads may be avoided if every newly allocated buffer is all filled with zeros. Data races can be avoided by changing timing related events such as thread-scheduling, asynchronous events, etc. Bugs that are exploited by malicious users can be avoided by dropping such requests during program re-execution. Even though dropping requests may make a few users (hopefully the malicious ones) unhappy, they do not introduce incorrect behavior to program execution as the failure-oblivious approaches do. Furthermore, given a spectrum of possible environmental changes, the least intrusive changes can be tried first, reserving the more extreme one as a last resort for when all other changes have failed. Finally, the specific environmental change which cures the problem gives diagnostic information as to what the bug is.

3 Main Idea of Rx

The main idea of Rx is to, upon a software failure, rollback the program to a recent checkpoint and re-execute it in a new environment that has been modified based on the failure symptoms. If the bugs allergen is removed from the new environment, the bug will not occur during re-execution, and thus the program will survive this software failure. After the re-execution safely passes through the problematic code region, the environmental changes are disabled to reduce time and space overhead imposed by the environmental changes. Figure 1 shows the process by which Rx survives software failures. Rx periodically takes light-weight checkpoints that are specially designed to survive software failures instead of hardware failures or OS crashes. When a bug is detected, either by an exception or by integrated dynamic software bug detection tools called as the Rx sensors, the program is rolled back to a recent checkpoint. Rx then analyzes the occurring failure based on the failure symptoms and experiences accumulated from previous failures, and determines how to apply environmental changes to avoid this failure. Finally, the program re-executes from the checkpoint in the modified environment. This process will repeat by

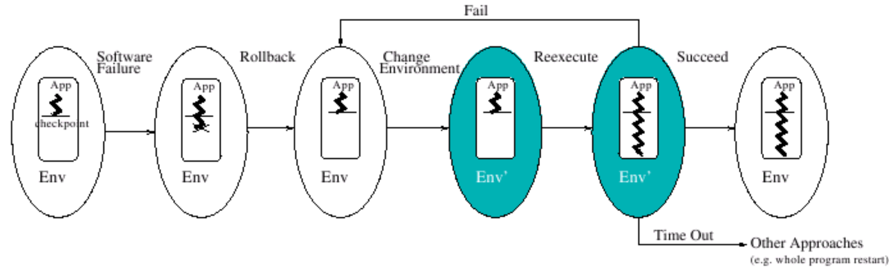

Figure 1: The main idea of Rx

re-executing from different checkpoints and applying different environmental changes until either the failure does not recur or Rx times out, resorting to alternate solution. If the failure does not occur during re-execution, the environmental changes are disabled to avoid the overhead associated with these changes. In this idea, the execution environment can include almost everything that is external to the target application but can affect the execution of the target application. At the lowest level, it includes the hardware such as processor architectures, devices, etc. At the middle level, it includes the OS kernel such as scheduling, virtual memory management, device drivers, file systems, network protocols, etc. At the highest level, it includes standard libraries, thirdparty libraries, etc. Obviously, the execution environment cannot be arbitrarily modified for re-execution. A useful re-execution environmental change should satisfy two properties.

3.1 Rx Design

Rx is composed of a set of user-level and kernel-level components that monitor and control the execution environment. The five primary components are seen in Figure 2: (1) sensors for detecting and identifying software failures or software defects at run time, (2) a Checkpoint-and-Rollback (CR) component for taking checkpoints of the target server application and rolling back the application to a previous checkpoint upon failure, (3) environment wrappers for changing execution environments during re-execution, (4) a proxy for making server recovery process transparent to clients, and (5) a control unit for maintaining checkpoints during normal execution, and devising a recovery strategy once software failures are reported by sensors.

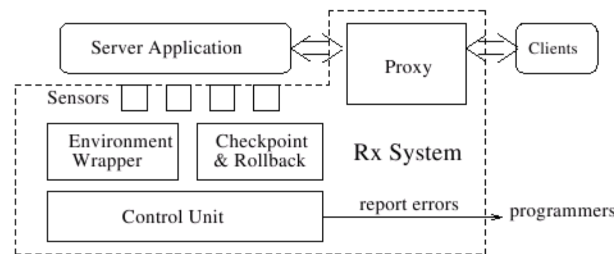

Figure 2: Rx architecture

3.2 Sensors

Sensors detect software failures by dynamically monitoring applications execution. There are two types of sensors. The first type detects software errors such as assertion failures, access violations, divide-by-zero exceptions, etc. This type of sensor can be implemented by taking over OS-raised exceptions. The second type of sensor detects software bugs such as buffer overflows, accesses to freed memory etc., before they cause the program to crash. Sensors notify the control unit

upon software failures with information to help identify the occurring bug for recovery and also for postmortem bug diagnosis.

3.3 Checkpoint and Rollback

The Checkpoint-and-Rollback component takes checkpoints of the target server application, and automatically and transparently rolls back the application to a previous checkpoint upon a software failure. At a checkpoint, CR stores a snapshot of the application into main memory. Similar to the fork operation, CR copies application memory in a copy-on-write fashion to minimize overhead. By preserving checkpoint states in memory, the overhead associated with slow disk accesses in most previous checkpointing solutions is avoided. This method is also used in previous work. Performing a rollback operation is straightforward: simply reinstate the program from the snapshot associated with the specified checkpoint. But in contrast to previous work on rollback and replay, Rx does not require deterministic replay. On the contrary, Rx purposely introduces nondeterminism into servers re-execution to avoid the bug that occurred during the first execution. Therefore, the underlying implementation of Rx can be simplified because it does not need to remember when an asynchronous event is delivered to the application in the first execution, how shared memory accesses from multiple threads are interleaved in a multi-processor machine, etc. The CR also supports multiple checkpoints and rollback to any of these checkpoints in case Rx needs to roll back further than the most recent checkpoint in order to avoid the occurring software bug. After rolling back to a checkpoint CPi, all checkpoints which were taken after CPi are deleted. This ensures that we do not rollback to a checkpoint which has been rendered obsolete by the rollback process. During a re-execution attempt, new checkpoints may be taken for future recovery needs in case this re-execution attempt successfully avoids the occurring software bug.

3.4 Environment Wrappers

The environment wrappers perform environmental changes during re-execution for averting failures. Some of the wrappers, such as the memory wrappers, are implemented at user level by intercepting library calls. Others, such as the message wrappers, are implemented in the proxy. Finally, still others, such as the scheduling wrappers, are implemented in the kernel.

3.5 Proxy

The proxy helps a failed server re-execute and makes server-side failure and recovery oblivious to its clients. When a server fails and rolls back to a previous checkpoint, the proxy replays all the messages received from this checkpoint. As shown in the figure 3, the Rx proxy can be in one of the two modes: normal mode for the servers normal execution and recovery mode during the servers re- execution. In the *normal* mode, the proxy simply bridges between the server and its clients. It keeps track of network connections and buffers the request messages between the server and its clients in order to replay them during the servers re-execution. It forwards client messages at request granularity. In other words, the proxy does not forward a partial request to the server. At a checkpoint, the proxy marks the next wait-for-forwarding request in its request buffer. When the server needs to roll back to this checkpoint, the mark indicates the place from which the proxy should replay the requests to the server. The proxy does not buffer any response in the normal mode except for those partially received responses. In the *recovery* mode, the proxy performs three functions to help server recovery: (1) it replays to the server those requests received since the checkpoint where the server is rolled back; (2) the proxy introduces message-related environmental changes to avoid some concurrency bugs; (3) the proxy buffers any incoming requests from clients without forwarding them to the server until the server successfully survives the software failure. Doing such makes the servers failure and recovery transparent to clients, especially since Rx has very fast recovery time. The proxy stays in the recovery mode until the server survives the software failure after one or multiple iterations of rollback and re-execution.

3.6 Control Unit

The control unit coordinates all the other components in the Rx. It performs three functions: (1) directs the CR to checkpoint the server periodically and requests the CR to roll back the server upon failures; (2) diagnoses an occurring failure based on the failure symptoms and its accumulated experiences, then decides what environmental changes should be applied and where the server should be rolled back to; (3) provides programmers useful failure-related information for postmortem bug analysis.

3.7 Advantages of Rx

Compared with previous solutions, Rx has the following unique advantages:

Comprehensive: Rx can survive many common software defects. Besides non-deterministic bugs, Rx can also survive deterministic bugs.

Safe: Rx does not speculatively fix bugs at run time. Instead, it prevents bugs from manifesting themselves by changing only the programs execution environment. It does not introduce uncertainty or misbehavior into a programs execution, which is usually very difficult for programmers to diagnose.

Noninvasive: Rx requires few to no modifications to applications source code. Therefore, it can be easily applied to legacy software.

Efficient: Because Rx requires no rebooting or warm-up, it significantly reduces system down time and provides reasonably good performance during recovery. Additionally, Rx is quite efficient. The technology imposes little overhead on server throughput and average response time and also has small space overhead.

Informative: Rx does not hide software bugs. Instead, bugs are still exposed. Furthermore, besides the usual bug report package (including core dumps, checkpoints and event logs), Rx provides programmers with additional diagnostic information for postmortem analysis, including what conditions triggered the bug and which environmental changes can or cannot avoid the bug. Based on such information, programmers can more efficiently find the root cause of the bug.

3.8 Disadvantages of Rx

In spite of the fact that Rx proposes the above advantages, there are also several limitations:

First, the authors are trying to evaluate Rx with more server applications containing real bugs under various workloads. *Second*, currently the Rxs proxy is implemented at the user level. To improve performance, it is planned to move it into the kernel, thereby avoiding context switches and memory copying. *Third*, the authors plan to extend Rx to support multi-tier server hierarchy. This is relative easy since Rx already works with a database server (MySQL), a web server (Apache), and aWeb proxy server (Squid). *Fourth*, the experiments so far have evaluated only I/O bound applications such as network servers whose availability is of critical importance. The authors of the technique plan to evaluate the Rxs overheads on computation intensive applications, and expect the overheads are likely to be higher. *Finally*, they have only compared with two alternative approaches: the whole program restart approach and a simple rollback and re-execution without environmental changes. This is because many other alternate approaches require substantial efforts to restructure/redesign applications.

While Rx can effectively and efficiently recover from many software failures caused by common software defects, Rx is certainly not a panacea. Rx cannot guarantee recovery from all software failures. Also, in some rare cases, it is possible that a bug still occurs during re-execution but its symptoms are not detected in-time by the sensors. In this case, Rx will claim a false recovery success. Additionally, Rx cannot deal with latent bugs bugs in which the fault is introduced at a time long before any obvious symptoms. As discussed in [18], this problem is general to all checkpoint-based recovery solutions.

Application	Version	Bug	# LOC	Application Description
MySQL	4.1.1.a	data race	588K	a database server
Squid	2.3.s5	buffer overflow	93K	a Web proxy cache server
Squid-ui	2.3.s5	uninitialized read		
Squid-dp	2.3.s5	dangling pointer		
Apache	2.0.47	stack overflow	283K	a Web server
CVS	1.11.4	double free	114K	a version control server

Table 1: Applications and Bugs

Apps	Bugs	Failure Symptoms	Environmental Changes	Clients Experience Failure?		Recoverable?		Average Recovery Time (s)	
				Alternatives	Rx	Alternatives	Rx	Restart	Rx
Squid	Buffer Overflow	SEGV	Padding	Yes	No	No	Yes	5.113	0.095
MySQL	Data Race	SEGV	Schedule Change	Yes	No	40% probablity	Yes*	3.500	0.161
Apache	Stack Overflow	Assert	Drop User Request	Yes	No	No	Yes	1.115	0.026
CVS	Double Free	SEGV	Delay Free	Yes	No	No	Yes	0.010	0.017
Squid-ui	Uninit Read	SEGV	Zero All	Yes	No	No	Yes	5.000	0.126
Squid-dp	Dangling Pointer	SEGV	Delay Free	Yes	No	No	Yes	5.006	0.113

Table 2: Overall results: comparison of Rx and alternative approaches

4 Evaluation

The authors evaluated four different real-world server applications as shown in Table 1 including a web server (Apache httpd), a web cache and proxy server (Squid), a database server (MySQL), and a concurrent version control server (CVS). The servers contained various types of bugs, including buffer overow, data race, double free, dangling pointer, uninitialized read, and stack overow bugs. Four of them were introduced by the original programmers.

As shown in Table 2, Rx can successfully avoid various types of common software defects, including 5 deterministic memory bugs and 1 concurrency bug. These bugs are avoided during re-execution because of Rxs environmental changes. For example, by padding buffers allocated during re-execution, Rx can successfully avoid the buffer overow bug in Squid. Apache survives the stack overow bug because Rx drops the bug-exposing user request during re-execution. Squid-ui survives the uninitialized read bug because Rx zero-fills all buffers allocated during re-execution. These results indicate that Rx is a viable solution to increase the availability of server applications.

Table 2 also shows that Rx provides a significantly better (21-53 times faster) recovery time than restart except for CVS. This is because rollback is a lightweight and fine-grained action due to the in-memory checkpoints. Also, as most faults were detected promptly (usually by crashing), the authors rarely needed to roll back further than the recent checkpoint. This minimized the amount of re-execution necessary. Furthermore, since the OPERAting Group was starting from a recent execution state, it was unnecessary to initialize data structures or to warm up buffer caches from disks. In contrast, restart is much slower. This is because restart requires the program to be reloaded and reinitialized from the beginning. Any memory state such as buffer caches and data structures need to be warmed up or initialized. Squid is a particularly clear example. For Squid, restart required 5.113 seconds to recover from a crash, whereas Rx took only 0.095 seconds. Since they used only a small workload, they expected that, with a real world workload, it would take an even longer time for the whole program restart approach to recovery from failures because it required a long time to warm up caches and other memory data structures. This result indicated that Rx enables servers to provide highly available services despite common software defects.

5 Related Work

The idea and work of Rx is built on much previous work. In this section I would like to briefly describe those works.

The idea of using checkpointing (accounted for Rx) to provide fault tolerance is old. It was already proposed in 1983 in proceedings of the 9th Symposium on Operating Systems Principles. It is also well known from the works [EAW] and [RLT78]. These checkpoints may be done to disk, remote memory, or non-volatile or persistent memory. These checkpoints can be provided with relatively low overhead. If there are messages and operations in flight, logging is also needed. After failure, many, but not all, errors can be avoided by reattempting the failed computation. In some related works, great care was taken to ensure deterministic replay. However, unlike deterministic replay used by other techniques, the authors of the Rx technique are purposely and systematically perturbing the re-execution environment to avoid determinism. As such, they have requirement and can use more lightweight checkpoints. Additionally, by changing environments, there is a possibility to tolerate faults which simple re-execution cannot.

Another related work is Failure-Oblivious Computing [RCD+04] proposes modifying the behavior of what it detects to be incorrect memory accesses. It discards or redirects incorrect writes to a separate hash table and manufactures values for incorrect reads. It has shortcomings in that it is restricted to memory related bugs, imposes high overheads and may introduce unpredictable behavior due to its potentially unsafe modifications to the memory interface. Also, the recently proposed reactive immune system [SLBK05] has similar limitations since it also speculatively fixes defects on-the-fly. As a result, unlike Rx, these approaches can be unsafe for correctness-critical server applications.

Recovery-Oriented Computing (ROC) [PBB+02] proposes restructuring the entire software platform to focus on and allow recovery. System components are to be isolated and failure aware. However, this requires not only restructuring individual servers, but all of the programs in the entire system. Micro-rebootable [CKF+04] software advocates software whose components are fail-stop, and individually recoverable, thereby making it easier to build fault-tolerant systems. Again, this requires reengineering of existing software. Rx can make use of dynamic bug detectors as sensors to determine when a bug has occurred. For memory bugs, dynamic checkers include Purify [HJ92], and StackGuard [CPM+98]. Many of these use instrumentation to monitor memory accesses, and hence impose high overhead. Some techniques can perform such checks with lower overhead, such as CCured [CHM+03] and SafeMem [QLZ05]. Beyond memory bugs, it is also possible to detect deadlock and races. Rx proxy is similar to the shadow drivers used in [SABL04], in that it interposes itself between the user of a service and the actual provider of that service in order to mask failures. However, rather than being between a kernel driver and applications calling on that driver, the authors of Rx are between a server process and the client processes. Furthermore the proxy does not replicate the original server during failure, but merely acts as a standin until the server recovers. The environmental changes they make are similar to noisemakers [**?**][51], except that, instead of trying to spur non-deterministic bugs into occurring, they were attempting to prevent deterministic and non-deterministic bugs by finding a legitimate execution path in which they simply do not arise.

6 Discussion

During the discussion session the following questions were discussed:

1. *What is part of a checkpoint?* At a checkpoint, CR stores a snapshot of the application into main memory. Similar to the fork operation, CR copies application memory in a copy-on-write fashion to minimize overhead.

2. *Checkpointing frequency?* For Rx, the checkpoint intervals in most cases are 200ms except for MySQL and CVS. For MySQL, they use a checkpoint interval of 750ms because too frequent checkpointing causes its data race bug to disappear in the normal mode.

3. *What are the policies to select/do the right environment changes?* The control unit diagnoses

an occurring failure based on the failure symptoms and its accumulated experiences, then decides what environmental changes should be applied and where the server should be rolled back to. After several failures, the control unit gradually builds up a failure table to capture the recovery experience for future reference (this experience represents the recovery policy).

7 Conclusion

In summary, Rx is a safe, non-invasive and informative method for quickly surviving software failures caused by common software defects such as memory corruptions and concurrency bugs and thus providing highly available services. It does so by re-executing the buggy program region in a modied execution environment. It can deal with both deterministic and non-deterministic bugs, and requires few to no modications to applications source code. Because Rx does not forcefully change programs execution by returning speculative values, it introduces no uncertainty or misbehavior into programs execution. Moreover, it provides additional feedback to programmers for their bug diagnosis. The experimental studies of four server applications that contain six bugs of different types show that Rx can successfully avoid software defects during re-execution and thus provide non-stop services. There are several limitations that the authors of Rx technique wish to address in their future work. First, they are trying to evaluate Rx with more server applications containing real bugs under various workloads. Second, currently the Rxs proxy is implemented at the user level. To improve performance, they plan to move it into the kernel, thereby avoiding context switches and memory copying. Third, they plan to extend Rx to support multi-tier server hierarchy. This is relative easy since Rx already works with a database server (MySQL), a web server (Apache), and a Web proxy server (Squid). Fourth, their experiments so far have evaluated only I/O bound applications such as network servers whose availability is of critical importance. They plan to evaluate the Rxs overheads on computation intensive applications, and they expect the overheads are likely to be higher. Finally, they have only compared with two alternative approaches: the whole program restart approach and a simple rollback and re-execution without environmental changes. This is because many other alternate approaches require substantial efforts to restructure/redesign applications.

References

[CHM+03] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. Ccured in the real world. *In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003. 5

[CKF+04] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. *In Proceedings of the 6th Symposium on Operating System Design and Implementation*, 2004. 5

[CPM+98] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. *In Proceedings of the 7th USENIX Security Symposium*, 1998. 5

[EAW] A survey of rollback-recovery protocols in message-passing systems. 5

[HJ92] R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. *In Proceedings of the USENIX Winter 1992 Technical Conference*, 1992. 5

[PBB+02] D. Patterson, A. Brown, A. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (roc): Motivation, definition, techniques, and case studies. *Technical report, U.C.Berkley*, 2002. 5

[QLZ05] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. *In Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005. 5

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies - a safe method to survive software failures. *SOSP05, Brighton, United Kingdom*, October 2005. (document)

[RCD+04] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. *In Proseedings of the 6th Symposium on OPerating System Design and Implementation*, 2004. 5

[RLT78] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computer Surveys*, 1978. 5

[SABL04] M. M. Swift, M. Annamalai, B. N. Bershad, and H.M. Levy. Recovering device drivers. *In Proceedings of the 6th Symposium on Operating System Design and Implementation*, 2004. 5

[SLBK05] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. *In Proceedings of the USENIX 2005 Annual Technical Conference*, 2005. 5