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

Feng Qin, Joseph Tucek, Jagadeesan Sundaresan and Yuanyuan Zhou
Department of Computer Science
University of Illinois at Urbana Champaign
{fengqin, tucek, sundaresan, yyzhou}@cs.uiuc.edu

ABSTRACT

Many applications demand 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.

This paper proposes an innovative *safe* technique, called Rx, which can quickly recover programs from many types of software bugs, both deterministic and non-deterministic. Our idea, inspired from allergy treatment in real life, is to rollback the program to a recent checkpoint upon a software failure, and then to re-execute the program in a *modified* environment. We base this idea on the observation that many bugs are correlated with the execution environment, and therefore can be avoided by removing the "allergen" from the environment. Rx requires few to no modifications to applications and provides programmers with additional feedback for bug diagnosis.

We have implemented Rx on Linux. Our experiments with four server applications that contain six bugs of various types show that Rx can survive all the six software failures and provide transparent fast recovery within 0.017-0.16 seconds, 21-53 times faster than the whole program restart approach for all but one case (CVS). In contrast, the two tested alternatives, a whole program restart approach and a simple rollback and re-execution without environmental changes, cannot successfully recover the three servers (Squid, Apache, and CVS) that contain deterministic bugs, and have only a 40% recovery rate for the server (MySQL) that contains a non-deterministic concurrency bug. Additionally, Rx's checkpointing system is lightweight , imposing small time and space overheads.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability

General Terms

Design, Experimentation, Reliability

SOSP'05, October 23–26, 2005, Brighton, United Kingdom.
Copyright 2005 ACM 1-59593-079-5/05/0010 ...$5.00.

Keywords

Availability, Bug, Reliability, Software Failure

1. INTRODUCTION

1.1 Motivation

Many applications, especially critical ones such as process control or on-line transaction monitoring, require high availability [26]. For server applications, downtime leads to lost productivity and lost business. According to a report by Gartner Group [47] the average cost of an hour of downtime for a financial company exceeds six million US dollars. With the tremendous growth of e-commerce, almost every kind of organization is becoming dependent upon highly available systems.

Unfortunately, software failures greatly reduce system availability. A recent study showed that software defects account for up to 40% of system failures [36]. Memory-related bugs and concurrency bugs are common software defects, causing more than 60% of system vulnerabilities [16]. 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.

Previous work on surviving software failures can be classified into four categories. The first category encompasses various flavors of rebooting (restarting) techniques, including whole program restart [26, 53], micro-rebooting of partial system components [12, 13], and software rejuvenation [29, 25, 7]. Since many of these techniques were originally designed to handle *hardware* failures, most of them are ill-suited for surviving software failures. For example, they cannot deal with deterministic software bugs, a major cause of software failures [17], because these bugs will still occur even after rebooting. Another important limitation of these methods is service unavailability while restarting, which can take up to several seconds [56]. For servers that buffer significant amount of state in main memory (e.g. data buffer caches), it requires a long period to warm up to full service capacity [10, 57]. Micro-rebooting [13] addresses this problem to some extent by only rebooting the failed components. However, it requires legacy software to be reconstructed in a loosely-coupled fashion.

The second category includes general checkpointing and recovery. The most straightforward method in this category is to checkpoint, rollback upon failures, and then re-execute either on the same machine [23, 42] or on a different machine designated as the "backup server" (either active or passive) [26, 5, 10, 11, 57, 2, 60]. Similar to the whole program restart approach, these techniques were also proposed to deal with hardware failures, and therefore suffer from the same limitations in addressing software failures. In particular, they also cannot deal with failures caused by deterministic bugs. Progressive retry [58] is an interesting improvement over these approaches. It reorders messages to increase the degree of non-determinism. While this work proposes a promising direction, it limits the technique to message reordering. As a result, it cannot handle bugs unrelated to message order. For example, if a server receives a malicious request that exploits a buffer overflow bug, simply reordering messages will not solve the problem. The most aggressive approaches in the checkpointing/recovery category include recovery blocks [41] and n-version programming [4, 3, 44], both of which rely on different implementation versions upon failures. These approaches may be able to survive deterministic bugs under the assumption that different versions fail independently. But they are too expensive to be adopted by software companies because they double the software development costs and efforts.

The third category comprises application-specific recovery mechanisms, such as the multi-process model (MPM), exception handling, etc. Some multi-processed applications, such as the old version of the Apache HTTP Server and the CVS server, spawn a new process for each client connection and therefore can simply kill a failed process and start a new one to handle a failed request. While simple and capable of surviving certain software failures, this technique has several limitations. First, if the bug is deterministic, the new process will most likely fail again at the same place given the same request (e.g. a malicious request). Second, if a shared data structure is corrupted, simply killing the failed process and restarting a new one will not restore the shared data to a consistent state, therefore potentially causing subsequent failures in other processes. Other application-specific recovery mechanisms require software to be failure-aware, which adversely affects programming difficulty and code readability.

The fourth category includes several recent non-conventional proposals such as failure-oblivious computing [43] and the reactive immune system [48]. Failure-oblivious computing proposes to deal with buffer overflows by providing *artificial* values for out-of-bound reads, while the reactive immune system returns a *speculative* error code for functions that suffer software failures (e.g. crashes). While these approaches are fascinating and may work for certain types of applications or certain types of bugs, they are *unsafe* to use for correctness-critical applications (e.g. on-line banking systems) because they "speculate" on programmers' intentions, which can lead to program misbehavior. The problem becomes even more severe and harder to detect if the speculative "fix" introduces a silent error that does not manifest itself immediately. In addition, such problems, if they occur, are very hard for programmers to diagnose since the application's execution has been forcefully perturbed by those speculative "fixes".

Besides the above individual limitations, existing work provides insufficient feedback to developers for debugging. For example, the information provided to developers may include only a core dump, several recent checkpoints, and an event log for deterministic replay of a few seconds of recent execution. To save programmers' debugging effort, it is desirable if the run-time system can provide information regarding the bug type, under what conditions the bug is triggered, and how it can be avoided. Such diagnostic information can guide programmers during their debugging process and thereby enhance their efficiency.

1.2 Our Contributions

In this paper, we propose 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. Our 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.

Our idea is inspired from real life. 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*. According to a previous study by Chandra and Chen [17], around 56% of faults in Apache depend on execution environment [1]. 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.

Similar to an allergy, it is difficult and expensive to apply these execution environmental changes from the very beginning of the program execution because we do not know what bugs might occur later. For example, zero-filling newly allocated buffers imposes time overhead. Therefore, we should lazily apply environmental changes only when needed.

We have implemented Rx with Linux and evaluated it with four server applications that contain four real bugs (bugs introduced by the original programmers) and two injected bugs (bugs injected by us) of various types including buffer overflow, double free, stack overflow, data race, uninitialized read and dangling pointer bugs. Compared with previous solutions, Rx has the following unique advantages:

[1] Note that our definition of execution environment is different from theirs. In our work, the standard library calls, such as $malloc$, and system calls are also part of execution environment.

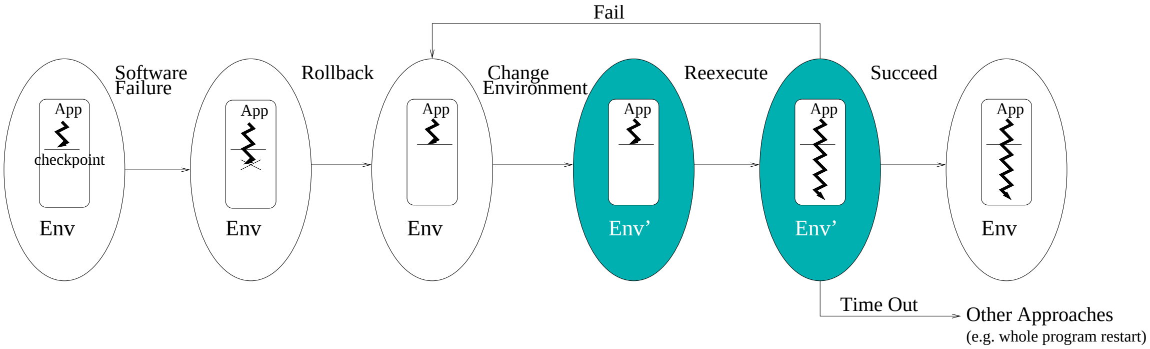


Figure 1: Rx: The main idea

- *Comprehensive:* Rx can survive many common software defects. Besides non-deterministic bugs, Rx can also survive deterministic bugs. Our experiments show that Rx can successfully survive the six bugs listed above. In contrast, the two tested alternatives, a whole program restart approach and a simple rollback and re-execution without environmental changes, cannot recover the three servers (Squid, Apache, and CVS) that contain deterministic bugs, and have only a 40% recovery rate for the server (MySQL) that contains a non-deterministic concurrency bug. Such results indicate that applying environmental changes during re-execution is the key reason for Rx's successful recovery of all tested cases.

- *Safe:* Rx does not speculatively "fix" bugs at run time. Instead, it prevents bugs from manifesting themselves by changing only the program's execution environment. Therefore, it does not introduce uncertainty or misbehavior into a program's 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. In our experiments, Rx successfully avoids software defects in the four tested server applications without modifying any of them.

- *Efficient:* Because Rx requires no rebooting or warm-up, it significantly reduces system down time and provides reasonably good performance during recovery. In our experiments, Rx recovers from software failure within 0.017-0.16 seconds, 21-53 times faster than the whole program restart approach for all but one case (CVS). Such efficiency enables servers to provide non-stop services despite software failures caused by common software defects. 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. For example, if Rx successfully avoids a bug by padding newly allocated buffers, the bug is likely to be a buffer overflow. Similarly, if Rx avoids a bug by delaying the recycling of freed buffers, the bug is likely to be caused by double free or dangling pointers.

2. 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 bug's "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 (See Section 3.2). 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 re-executing from different checkpoints and applying different environmental changes until either the failure does not recur or Rx times out, resorting to alternate solutions, such as whole-program rebooting [26, 53]. If the failure does not occur during re-execution, the environmental changes are disabled to avoid the overhead associated with these changes.

In our 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, third-party libraries, etc. Such definition of the execution environment is much broader than the one used in previous work [17].

Obviously, the execution environment cannot be arbitrarily modified for re-execution. A useful re-execution environmental change should satisfy two properties. First, it should be *correctness-preserving*, i.e., every step (e.g., instruction, library call and system call) of the program is executed according to the APIs. For example, in the $malloc()$ library call, we have the flexibility to decide where buffers should be allocated, but we cannot allocate a smaller buffer than requested. Second, a useful environmental change should be able to potentially avoid some software bugs. For example, padding every allocated buffer can prevent some buffer overflow bugs from manifesting during re-execution.

Category	Environmental Changes	Potentially-Avoided Bugs	Deterministic?
Memory Management	delayed recycling of freed buffer	double free, dangling pointer	YES
	padding allocated memory blocks	dynamic buffer overflow	YES
	allocating memory in an alternate location	memory corruption	YES
	zero-filling newly allocated memory buffers	uninitialized read	YES
Asynchronous	scheduling	data race	NO
	signal delivery	data race	NO
	message reordering	data race	NO
User-Related	dropping user requests	bugs related to the dropped request	Depends

Table 1: Possible environmental changes and their potentially-avoided bugs

Examples of useful execution environmental changes include, but are not limited to, the following categories:

(1)Memory management based: Many software bugs are memory related, such as buffer overflows, dangling pointers, etc. These bugs may not manifest themselves if memory management is performed slightly differently. For example, each buffer allocated during re-execution can have padding added to both ends to prevent some buffer overflows. Delaying the recycling of freed buffers can reduce the probability for a dangling pointer to cause memory corruption. In addition, buffers allocated during re-execution can be placed in isolated locations far away from existing memory buffers to avoid some memory corruption. Furthermore, zero-filling new buffers can avoid some uninitialized read bugs. *Since none of the above changes violate memory allocation or deallocation interface specifications, they are safe to apply*. Also note that these environmental changes affect only those memory allocations/deallocations made during re-execution.

(2)Timing based: Most non-deterministic software bugs, such as data races, are related to the timing of asynchronous events. These bugs will likely disappear under different timing conditions. Therefore, Rx can forcefully change the timing of these events to avoid these bugs during re-execution. For example, increasing the length of a scheduling time slot can avoid context switches during buggy critical sections. This is very useful for those concurrency bugs that have high probability of occurrences. For example, the data race bug in our tested MySQL server has a 40% occurrence rate on a uniprocessor machine.

(3)User request based: Since it is infeasible to test every possible user request before releasing software, many bugs occur due to unexpected user requests. For example, malicious users issue malformed requests to exploit buffer overflow bugs during stack smashing attacks [21]. These bugs can be avoided by dropping some users' requests during re-execution. Of course, since the user may not be malicious, this method should be used as a last resort after all other environmental changes fail.

Table 1 lists some environmental changes and the types of bugs that can be potentially avoided by them. Although there are many such changes, due to space limitations, we only list a few examples for demonstration.

If the failure disappears during a re-execution attempt, the failure symptoms and the effects of the environmental changes applied are recorded. This speeds up the process of dealing with future failures that have similar symptoms and code locations. Additionally, Rx provides all such diagnostic information to programmers together with core dumps and other basic postmortem bug analysis information. For example, if Rx reports that buffer padding does not avoid the occurring bug but zero-filling newly allocated buffers does, the programmer knows that the software failure is more likely to be caused by an uninitialized read instead of a buffer overflow.

After a re-execution attempt successfully passes the problematic program region for a threshold amount of time, all environmental changes applied during the successful re-execution are disabled to reduce space and time overheads. These changes are no longer necessary since the program has safely passed the "allergic seasons".

If the failure still occurs during a re-execution attempt, Rx will rollback and re-execute the program again, either with a different environmental change or from an older checkpoint. For example, if one change (e.g. padding buffers) cannot avoid the bug during the re-execution, Rx will rollback the program again and try another change (e.g. zero-filling new buffers) during the next re-execution. If none of the environmental changes work, Rx will rollback further and repeat the same process. If the failure still remains after a threshold number of iterations of rollback/re-execute, Rx will resort to previous solutions, such as whole program restart [26, 53], or micro-rebooting [12, 13] if supported by the application.

Upon a failure, Rx follows several rules to determine the order in which environmental changes should be applied during the recovery process. First, if a similar failure has been successfully avoided by Rx before, the environmental change that worked previously will be tried first. If this does not work, or if no information from previous failures exists, environmental changes with small overheads (e.g. padding buffers) are tried before those with large overheads (e.g. zero-filling new buffers). Changes with negative side effects (e.g. dropping requests) are tried last. Changes that do not conflict, such as padding buffers and changing event timing, can be applied simultaneously.

Although the situation never arose during our experiments, there is still the rare possibility that a bug still occurs during re-execution but is not detected in time by Rx's sensors. In this case, Rx will claim a recovery success while it is not. Addressing this problem requires using more rigorous on-the-fly software defect checkers as sensors. This is currently a hot research area that has attracted much attention. In addition, it is also important to note that, *unlike in failure oblivious computing, this problem is caused by the application's bug instead of Rx's environmental changes*. The environmental changes just make the bug manifest itself in a different way. Furthermore, since Rx logs its every action including what environmental changes are applied and what the results are, programmers can use this information (i.e. some environmental changes make the software crash much later) to analyze the occurring bug.

3. 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,

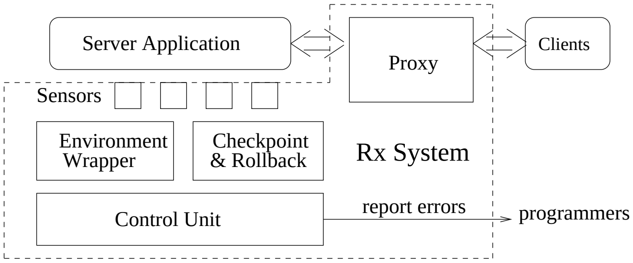


Figure 2: Rx architecture

and (5) a control unit for maintaining checkpoints during normal execution, and devising a recovery strategy once software failures are reported by sensors.

3.1 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. This type of sensors leverage existing low-overhead dynamic bug detection tools, such as CCured [20], StackGuard [21], and our previous work SafeMem [40], just to name a few. In our Rx prototype, we have only implemented the first type of sensors. However, we plan to integrate second type of sensors into Rx.

Sensors notify the control unit upon software failures with information to help identify the occurring bug for recovery and also for postmortem bug diagnosis. Such information includes the type of exception (Segmentation fault, Floating Point Exception, Bus Error, Abort, etc.), the address of the offending instruction, stack signature, etc.

3.2 Checkpoint and Rollback

3.2.1 Mechanism

The CR (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 [19, 30, 32, 59, 35, 39, 49]. Performing a rollback operation is straightforward: simply reinstate the program from the snapshot associated with the specified checkpoint.

Besides memory states, the CR also needs to take care of other system states such as file states during checkpointing and rollback to ensure correct re-execution. To handle file states, CR applies ideas similar to previous work [35, 49] by keeping a copy of each accessed files and file pointers in the beginning of a checkpoint interval and reinstate it for rollback. To simplify implementation, we can leverage a versioning file system which automatically takes a file version upon modifications. Similarly, copy-on-write is used to reduce space and time overheads. For some logs file that users may want the old content not to be overwritten during re-execution, Rx can easily provide a special interface that allows applications to indicate what files should not be rolled back. Other system states such as messages and signals will be described in the next subsection because they may need to be changed to avoid a software bug recurring during re-execution. More details about our lightweight checkpointing method can be found in our previous work [49], which uses checkpointing and logging to support deterministic replay for interactive debugging.

In contrast to previous work on rollback and replay, Rx does not require deterministic replay. On the contrary, Rx purposely introduces *nondeterminism* into server's 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., as we have done in our previous work [49].

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 CP_i, all checkpoints which were taken after CP_i 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.

3.2.2 Checkpoint Maintenance

A possible concern is that maintaining multiple checkpoints could impose a significant space overhead. To address this problem, Rx can write old checkpoints to disks on the background when disks are idle. But rolling back to a checkpoint, which is already stored in disks, is expensive due to slow disk accesses.

Fortunately, we do not need to keep too many checkpoints because Rx strives to bound its recovery time to be 2-competitive as the baseline solution: whole program restarting. In other words, in the worse case, Rx may take twice as much time as the whole program restarting solution (In reality, in most cases as shown in Section 6, Rx recovers much faster than the whole program restart). Therefore, if a whole program restart would take T seconds (This number can be measured by restarting immediately at the first software failure and then be used later), Rx can only repeat rollback/re-execute process for at most T seconds. As a result, Rx cannot rollback to a checkpoint which is too far back in the past, which implies that Rx does not need to keep such checkpoints any more.

More formally, suppose Rx takes checkpoints periodically, let τ_1,τ_2,$\cdots$,τ_n be the timestamps of the last n checkpoints that have been kept in the *reverse* chronological order. We can use two schemes to keep those checkpoints: one is to keep only recent checkpoints, and the other is to keep exponential landmark checkpoints (with β as the exponential factor) as in the Elephant file system [46]. In other words, the two schemes satisfy the following equations, respectively.

$$\tau_i - \tau_{i+1} = \tau_{i-1} - \tau_i \quad (2 \le i \le n-1)$$

$$\tau_i - \tau_{i+1} = \beta * (\tau_{i-1} - \tau_i) \quad (2 \le i \le n-1)$$

Note that time here refers to application execution time as opposed to elapse time. The latter can be significantly higher, especially when there are many idle periods.

After each checkpoint, Rx estimates whether it is still useful to keep the oldest checkpoint. If not, the oldest checkpoint taken at time τ_n is deleted from the system to save space. The estimation is done by calculating the worst-case recovery time that requires rolling back to this oldest checkpoint. Suppose after rolling back to a checkpoint, every ith re-execution ($1 \leq i \leq m$) with different environmental changes incurs the overhead p_i. Obviously, some environmental changes such as buffer padding impose little time overhead, whereas other changes such as zero-filling buffers incur large overhead. p_is can be measured at run time. Therefore the worst-case recovery time, $RTime$, that requires to roll back to the oldest checkpoint would be (let τ be the current timestamp):

$$RTime = \sum_{i=1}^{n}\sum_{j=1}^{m}(\tau - \tau_i)(1 + p_j) = \sum_{i=1}^{n}(\tau - \tau_i)\sum_{j=1}^{m}(1 + p_j)$$

If $RTime$ is greater than T, the oldest checkpoint taken at time τ_n is deleted.

3.3 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.

Memory Wrapper The memory wrapper is implemented by intercepting memory-related library calls such as *malloc(), realloc(), calloc(), free(),* etc to provide environmental changes. During the normal execution, the memory wrapper simply invokes the corresponding standard memory management library calls, which incurs little overhead. During re-execution, the memory wrapper activates the memory-related environmental changes instructed by the control unit. Note that the environmental changes only affect the memory allocation/deallocation made during re-execution.

Specifically, the memory wrapper supports four environmental changes:

(1) Delaying free, which delays recycling of any buffers freed during a re-execution attempt to avoid software bugs such as double free bugs and dangling pointer bugs. A freed buffer is reallocated only when there is no other free memory available or it has been delayed for a threshold of time (process execution time, not elapsed time). Freed buffers are recycled in the order of the time when they are freed. This memory allocation policy is not used in the normal mode because it can increase paging activities.

(2) Padding buffers, which adds two fixed-size paddings to both ends of any memory buffers allocated during re-execution to avoid buffer overflow bugs corrupting useful data. This memory allocation policy is only used in the recovery mode because it wastes memory space.

(3) Allocation isolation, which places all memory buffers allocated during re-execution in an isolated location to avoid corruption useful data due to severe buffer overflow or other general memory corruption bugs. Similar to padding, it is disabled in the normal mode because it has space overhead.

(4) Zero-filling, which zero-fills any buffers allocated during re-execution to reduce the probability of failures caused by uninitialized reads. Obviously, this environmental change needs to be disabled in the normal mode since it imposes time overhead.

Since none of the above changes violate memory allocation or deallocation interface specifications, they are safe to apply. At each memory allocation or free, the memory wrapper returns exactly what the application may expect. For example, when an application asks for a memory buffer of size N, the memory wrapper returns a buffer with at least size N, even though this buffer may have been padded at its both ends, allocated from an isolated location, or zero-filled.

Message Wrapper Many concurrency bugs are related to message delivery such as the message order across different connections, the size and number of network packets which comprise a message, etc. Therefore, changing these execution environments during re-execution may be able to avoid an occurring concurrency software bug. This is feasible because servers typically should *not* have any expectation regarding the order of messages from different connections (users), the size and the number of network packets that forms a message, especially the latter two which depend on the TCP/IP settings of both sides.

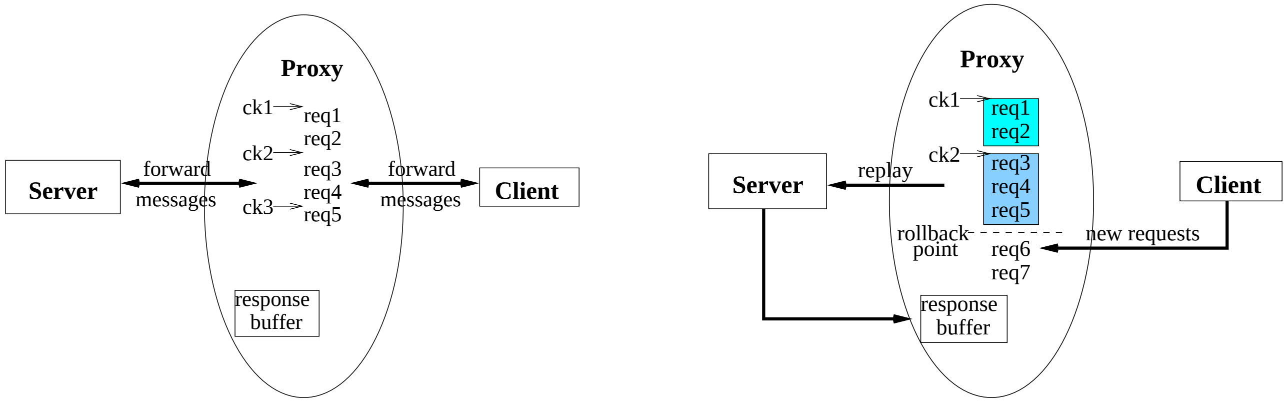
The message wrapper, which is implemented in the proxy (described in the next subsection), changes the message delivery environment in two ways: (1) It can randomly shuffle the order of the requests among different connections, but keep the order of the requests within each connection in order to maintain any possible dependency among them. (2) It can deliver messages in random-sized packets. Such environmental changes do not impose overhead. Therefore, this message delivery policy can be used in the normal mode, but it does not decrease the probability of the occurrence of a concurrency bug because there is no way to predict in what way a concurrency bug does not occur.

Process Scheduling Similarly, concurrency bugs are also related to process scheduling and are therefore prone to disappear if a different process scheduling is used during re-execution. Rx does this by changing the process' priority, and thus increasing the scheduling time quantum so a process is less likely to switched off in the middle of some unprotected critical region.

Signal Delivery Similar to process scheduling, the time when a signal is delivered may also affect the probability of a concurrency bug's occurrence rate. Therefore, Rx can record all signals in a kernel-level table before delivering them. For hardware interrupts, Rx delivers them at randomly selected times, but preserving their order to maintain any possible ordering semantics. For software timer signals, Rx ignores them because during rollback, the related software timer will also be restored. For software exception related signals such as segmentation faults, Rx's sensors receive them as indications of software failures.

Dropping User Requests Dropping user requests is a last environmental change before switching to the whole program restart solution. As described earlier, the rational for doing this is that some software failures are triggered by some malformed requests, either unintentionally or intentionally by malicious users. If Rx drops that request, the server will not experience failure. In this case, the server only denies those dropped requests, but does not affect other requests. The effectiveness of this environmental change is based on our assumption that the client and server use a request/response model, which is generally the case for large varieties of servers including Web Servers, DNS Servers, database servers, etc.

Rx does not need to look for the exact culprit user request. As long as the dropped requests include this request, the server can avoid the software bug and continue providing services. Of course, the percentage of dropped requests should be small (e.g. 10%) to avoid malicious users exploiting it to launch denial of service attacks. Rx can achieve this by performing a binary search on all recently received requests. First, it can drop half of them to see whether the bug still occurs during re-execution. If not, the problem request set becomes one half smaller. If the bug still occurs, it rolls back to drop the other half. If it still does not work, Rx resorts to the whole program restart solution. Otherwise, the binary search continues until the percentage of dropped requests becomes smaller



(a) Proxy behavior in normal mode

(b) Proxy behavior in recovery mode

Figure 3: (a) In normal mode, the proxy forward request/response messages between the server and the client, buffers requests, and marks the waiting-for-sending request for each checkpoint (e.g., req3 is marked by checkpoint 2). (b) After the server is rolled back from the rollback-point, as shown in the dashed line to checkpoint 2, the proxy discards the mark of checkpoint 3, replays the necessary requests (req3, req4 and req5) to the server and buffers the incoming requests (req6 and req7). The "unanswered" responses are buffered in the response buffer.

than the specified number. If the percentage upper bound is set to be 10%, it only takes 5 iterations of rollback and re-execution.

After Rx finds the small set of requests (less than the specified upper bound) that, once dropped, enable the server to survive the bug, Rx can remember each request's signatures such as the IP address, message size, message MD5 hash value, etc. In subsequent times when a similar bug recurs in the normal mode, Rx can record the signatures again. After several rounds, Rx accumulates enough sets of signatures so that it can use statistical methods to identify the characteristics of those bug-exposing requests. Afterward, if the same bug recurs, Rx can drop only those requests that match these characteristics to speed up the recovery process.

3.4 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, along with the message-based environmental changes described in the Section 3.3. The proxy runs as a stand-alone process in order to avoid being corrupted by the target server's software defects.

As Figure 3 shows, the Rx proxy can be in one of the two modes: *normal* mode for the server's normal execution and *recovery* mode during the server's re-execution. For simplicity, the proxy forwards and replays client messages in the granularity of user requests. Therefore, the proxy needs to separate different requests within a stream of network messages. The proxy does this by plugging in some simple information about the application's communication protocol (e.g. HTTP) so it can parse the header to separate one request from another. In addition, the proxy also uses the protocol information to match a response to the corresponding request to avoid delivering a response to the user twice during re-execution. In our experiments, we have evaluated four server applications, and the proxy uses only 509 lines of code to handle 3 different protocols: HTTP, MySQL message protocol and CVS message protocol.

As shown on Figure 3(a), 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 server's 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. This is because after a full response is received, the proxy sends it out to the corresponding client and mark the corresponding request as "answered". Keeping these committed responses is useless because during re-execution the proxy cannot send out another response for the same request. Similarly, the proxy also strives to forward messages to clients at response granularity to reduce the possibility of sending a self-conflicting response during re-execution, which may occur when the first part of the response is generated by the server's normal execution and the second part of the response is generated by re-execution that may take a different execution path.

However, if the response is too large to be buffered, a partial response is sent first to the corresponding client but the MD5 hash for this partial response is calculated and stored with the request. If a software failure is encountered before the proxy receives the entire response from the server, the proxy needs to check the MD5 hash of the same partial response generated during re-execution. If it does not match with the stored value, the proxy will drop the connection to the corresponding client to avoid sending a self-conflicting response. To handle the case where a checkpoint is taken in the middle of receiving a response from the server, the proxy also marks the exact position of the partially-received response.

As shown on Figure 3(b), in the *recovery* mode, the proxy performs three functions to help server recovery. First, it replays to the server those requests received since the checkpoint where the server is rolled back. Second, the proxy introduces message-related environmental changes as described in Section 3.3 to avoid some concurrency bugs. Third, 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 server's failure and recovery transparent to clients, especially since Rx has very fast recovery time as shown in Section 6. The proxy stays in the recovery mode until the server survives the software failure after one or multiple iterations of rollback and re-execution.

To deal with the output commit problem [52] (clients should perceive a consistent behavior of the server despite server failures), Rx first ensures that any previous responses sent to the client are not resent during re-execution. This is achieved by recording for each request whether it has been responded by the server or not. If so, a response made during re-execution is dropped silently. Otherwise, the response generated during re-execution will be temporally buffered until any of the three conditions is met: (1) the server successfully avoids the failure via rollback and re-execution in changed execution environments; (2) the buffer is full; or (3) this re-execution fails again. For the first two cases, the proxy sends the buffered responses to the corresponding clients and the corresponding requests are marked as “answered”. Thus, responses generated in subsequent re-execution attempts will be dropped to ensure that only one response for each request goes to the client. For the last case, the responses are thrown away.

For applications such as on-line shopping that require strict session consistency (i.e. later requests in the same session depend on previous responses), Rx can record the signatures (hash values) of all committed responses for each outstanding session, and perform MD5 hash-based consistency checks during re-execution. If a re-execution attempt generates a response that does not match with a committed response for the same request in an outstanding session, this session can be aborted to avoid confusing users.

The proxy also supports multiple checkpoints. When an old checkpoint is discarded, the proxy discards the marks associated with this checkpoint. If this checkpoint is the oldest one, the proxy also discards all the requests received before the second oldest checkpoint since the server can never roll back to the oldest checkpoint any more.

The space overhead incurred by the proxy is small. It mainly consists of two parts: (1) space used to buffer requests received since the undeleted oldest checkpoint, (2) a fixed size space used to buffer “unanswered” responses generated during re-execution in the recovery mode. The first part is small because usually requests are small, and the proxy can also discard the oldest checkpoint to save space as described in Section 3.2. The second part has a fixed size and can be specified by administrators.

3.5 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.

After several failures, the control unit gradually builds up a failure table to capture the recovery experience for future reference. More specifically, during each re-execution attempt, the control unit records the effects (success or failure) and the corresponding environmental changes into the table. The control unit assigns a score vector $\langle s_1, s_2, \cdots, s_m \rangle$ to each failure, where m is the number of possible environmental changes. Each element s_i in the vector is the score for each corresponding environmental change C_i for a certain failure. For a *successful* re-execution, the control unit adds one point to all the environmental changes that are applied during this re-execution. For a *failed* re-execution,the control unit subtracts one point from all the applied environmental changes. When a failure happens, the control unit searches the failure table based on failure symptoms, such as type of exceptions, instruction counters, call chains, etc, provided by the Rx sensors. If one table entry matches, it then applies those environmental changes whose scores are larger than a certain threshold T_s. Otherwise, it will follow the rules described in Section 2 to determine the order how environmental changes should be applied during re-execution. This failure table can be provided to programmers for postmortem bug analysis. It is possible to borrow ideas from machine learning (e.g., a Bayesian classifier) or use some statistical methods as a more “advanced” technique to learn what environmental changes are the best cure for a certain type of failures. Such optimization remains as our future work.

4. DESIGN AND IMPLEMENTATION ISSUES

Inter-Server Communication In many real systems, servers are tiered hierarchically to provide service. For example, a web server is usually linked to an application server, which is then linked to a backend database server. In this case, rolling back one failed server may not be enough to survive a failure because the failure may be caused by its front-end or back-end servers. To address this problem, Rx should be used for all servers in this hierarchy so that it is possible to rollback a subset or all of them in order to survive a failure. We can borrow many ideas, such as, coordinated checkpointing, asynchronous recovery, etc, from previous work on supporting fault tolerance in distributed systems [14, 15, 23, 44, 1], and also from recent work such as micro-reboot [13]. More specifically, during the normal execution, Rx in the tiered servers take checkpoints coordinately. Once a failure is detected, Rx rolls back the failed server and also broadcasts its rollback to other correlated servers, which then roll back correspondingly to recover the whole system. Currently, we have not implemented such support in the Rx and it remains a topic for future study.

Multi-threaded Process Checkpointing Taking a checkpoint on a multi-threaded process is particularly challenging because, when Rx needs to take a checkpoint, some threads may be executing system calls or could be blocked inside the kernel waiting for asynchronous events. Capturing the transient state of such threads could easily lead to state inconsistency upon rollback. For example, there can be some kernel locks which have been acquired during checkpoint, and rolling back to such state may cause two processes hold the same kernel locks. Therefore, it is essential that we force all the threads to stay at the user level before checkpointing. We implement this by sending a signal to all threads, which makes them exit from blocked system calls or waiting events with an EINTR return code. After the checkpoint, the library wrapper in Rx retries the prematurely returned system calls and thus hides the checkpointing process from the target application. This has a bearing on the checkpointing frequency, as a high checkpointing frequency will severely impair the performance of normal I/O system calls, which are likely to be retried multiple times (once at every checkpoint) before long I/Os finish. Therefore, we cannot set the checkpointing interval too small.

Unavoidable Bug/Failure for Rx Even though Rx should be able to help servers recover from most software failures caused by common software bugs such as memory corruptions and concurrency bugs, there are still some types of bugs that Rx cannot help the server to avoid via re-execution in changed execution environments. Resource leakage bugs, such as memory leaks, which have accumulative effects on system and may take hours or days to cause system to crash, cannot be avoided by only rolling the server back to a recent checkpoint. Therefore, for resource leaking, Rx resorts to the whole program restart approach because restart can refresh server with plenty of resources. For some of the semantic bugs, Rx may not be effective to avoid them since they may not be related to execution environments. Finally, Rx are not able to avoid the bugs or failures that sensors cannot detect. Solving this problem would require more rigorous dynamic checkers as sensors.

5. EVALUATION METHODOLOGY

The experiments described in this section were conducted on two machines with a 2.4GHz Pentium processor, 512KB L2 cache, 1GB of memory, and a 100Mbps Ethernet connection between them. We run servers on one machine and clients on the other. The operating system we modified is the Linux kernel 2.6.10. The Rx proxy is currently implemented at user level for easy debugging. In the future, we plan to move it to the kernel level to improve performance.

We evaluate four different real-world server applications as shown in Table 2, 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 contain various types of bugs, including buffer overflow, data race, double free, dangling pointer, uninitialized read, and stack overflow bugs. Four of them were introduced by the original programmers. We have not yet located server applications which contain uninitialized read or dangling pointer bugs. To evaluate Rx's functionality of handling these two types of bugs, we inject them into Squid separately, renaming the two Squids as Squid-ui (containing an uninitialized read bug) and Squid-dp (containing a dangling pointer bug), respectively.

App	Ver	Bug	#LOC	App 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 2: Applications and Bugs (App means Application. Ver means Version. LOC means lines of code).

In this paper, we design four sets of experiments to evaluate the key aspects of Rx:

- The first set evaluates the functionality of Rx in surviving software failures caused by common software defects by rollback and re-execution with environmental changes. We compare Rx with whole program restart in terms of client experiences during failure, and in terms of recovery time. In addition, we also compare Rx with the simple rollback and re-execute with *no* environmental changes. This approach is implemented by disabling environmental changes in Rx.
- The second set evaluates the performance overhead of Rx for both server throughput and average response time without bug occurrence. Additionally, we evaluate the space overhead caused by checkpoints and the proxy.
- The third set evaluates how Rx would behave under certain degree of malicious attacks that continuously send bug-exposing requests triggering buffer overflow or other software defects. We measure the throughput and average response time under different bug arrival rates. In this set of experiments, we also compare Rx with the whole program restart approach in terms of performance.
- The fourth set evaluates the benefits of Rx's mechanism of learning from previous failure experiences, which are stored in the failure table to speed up recovery.

For all the servers, we implement clients in a similar manner as previous work, such as httperf [37] or WebStone [55], sending continuous requests over concurrent connections. For Squid and Apache, the clients spawn 5 threads. Each thread sends out requests to fetch different files whose sizes range in 1KB, 2KB, ..., 512KB with uniform distribution. For CVS, the client exports a 30KB source file. For MySQL, we use two loads. To trigger the data race, the client spawns 5 threads, each of them sending out begin, select, and commit requests on a small table repeatedly. The size of individual requests must be as small as possible to maximize the probability of the race occurring. For the overhead experiments with MySQL, a more realistic load with updates is used. To demonstrate that Rx can avoid server failures, we use another client that sends bug-exposing requests to those servers.

6. EXPERIMENTAL RESULTS

6.1 Overall Results

Table 3 demonstrates the overall effectiveness of Rx in avoiding bugs. For each buggy application, the table shows the type of bug, what symptom was used to detect the bug, and what environmental change was eventually used to avoid the bug. The table also compares Rx to two alternative approaches: the ordinary whole program restart solution and a simple rollback and re-execution without environmental changes. For Rx, the checkpoint intervals in most cases are 200ms except for MySQL and CVS. For MySQL, we use a checkpoint interval of 750ms because too frequent checkpointing causes its data race bug to disappear in the normal mode. The reason for using 50ms as the checkpoint interval for CVS will be explained later when we discuss the recovery time. The average recovery time is the recovery time averaged across multiple bug occurrences in the same execution. Section 6.4 will discuss the difference in Rx recovery time between the first time bug occurrence and subsequent bug occurrences.

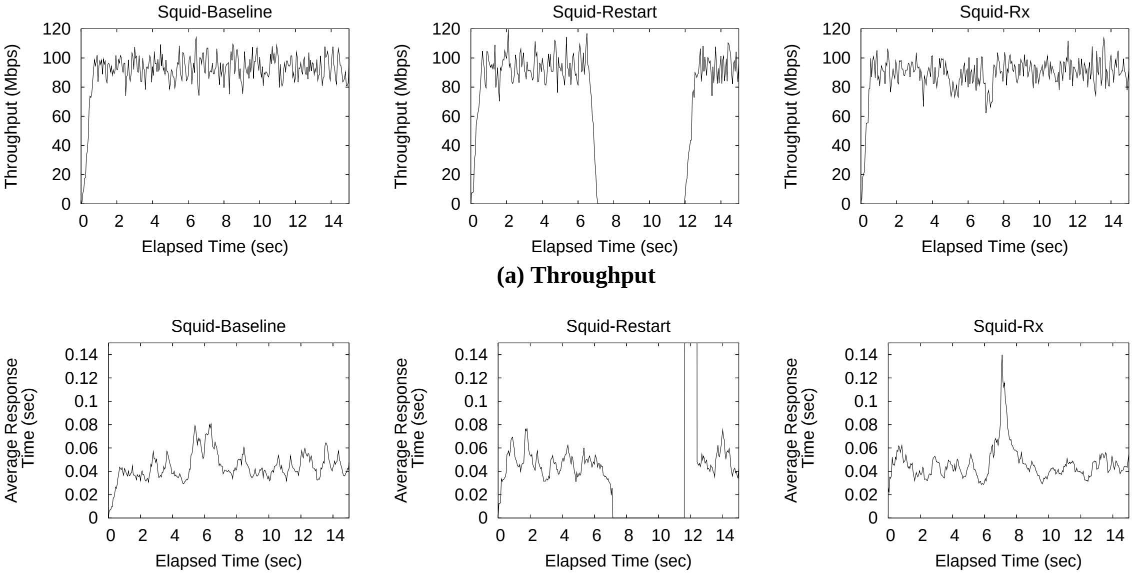
As shown in Table 3, Rx can successfully avoid various types of common software defects, including 5 deterministic memory bugs and 1 concurrency bug. These bugs are avoided during re-execution because of Rx's environmental changes. For example, by padding buffers allocated during re-execution, Rx can successfully avoid the buffer overflow bug in Squid. Apache survives the stack overflow 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.

In contrast, the two alternatives, restart and simple rollback/re-execution, cannot successfully recover the three servers (Squid, Apache and CVS) that contain deterministic bugs. For the restart approach, this is because the client notices a disconnection and tries to resend the same bug-exposing request, which causes the server to crash again. For the simple rollback and re-execution approach, once the server rolls back to a previous checkpoint and starts re-execution, the same deterministic bug will occur again, causing the server to crash immediately. These two alternatives have a 40% recovery rate for MySQL that contains a non-deterministic concurrency bug because in 60% cases the same bug-exposing interleaving is used again after restart or rollback. Such results show that these two alternative approaches, even though simple, cannot survive failures caused by many common software defects and thus cannot provide continuous services. The results also indicate that applying environmental changes is the key reason why Rx can survive software failures caused by common software defects, especially *deterministic* bugs.

Because the Rx's proxy hides the server failure and recovery process from its clients, clients do not experience any failures. In contrast, with restart, clients experience failures due to broken network connections. To be fault tolerant, clients need to reconnect to the server and reissue all unreplied requests. With the simple rollback and re-execution, since the server cannot recover from the failure, clients eventually time out and thus experience server failures.

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 3: Overall results: comparison of Rx and alternative approaches (whole program restart, and simple rollback and re-execution without environmental changes). The results are obtained by running each experiment 20 times. *The recovery time for the restart approach is measured by having the client not resend the bug-exposing request after reconnection.* **Otherwise, the server will crash again immediately after restart. *For MySQL, during the 20 runs, the data race bug never occur during re-execution in Rx after applying various timing-related environmental changes.**



(a) Throughput

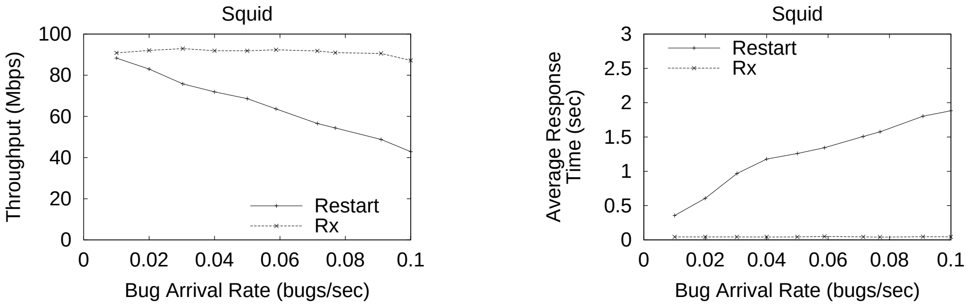
(b) Average Response Time

Figure 4: Throughput and average response time of Squid with Rx and Restart for one bug occurrence (Between time period (7,11.5), there are no measurements for restart because no requests are responded during this period.)

Table 3 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 we find that most faults are detected promptly (usually by crashing), we rarely need to roll back further than the recent checkpoint. This minimizes the amount of re-execution necessary. Furthermore, since we are starting from a recent execution state, it is 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 requires 5.113 seconds to recover from a crash, whereas Rx takes only 0.095 seconds. Since our experiments use only a small workload, we expect that, with a real world workload, it will take an even longer time for the whole program restart approach to recovery from failures because it requires a long time to warm up caches and other memory data structures. This result indicates that Rx enables servers to provide highly available services despite common software defects. Instead of experiencing a failure, clients experience an increased response time for a very short period. We expect that after the Rx's proxy is pushed into the kernel, the Rx results will be even better since such optimization will reduce the number of context switches and memory coping overhead.

If the bug-exposing request is not resent after failure, restart has similar recovery time for CVS (otherwise, restart cannot recover the failure for CVS). Restart takes only .01 seconds to recover for CVS, while Rx takes .017 seconds. This is because CVS is implemented using the xinetd daemon as its network monitor. Each connection to CVS causes xinetd to fork and exec a new instance of CVS. Therefore, CVS must have a very low startup time in order to provide adequate performance. Additionally, there is no state shared between different CVS processes except for that of the repository, which is persistently stored on disk. As such, CVS has only minimal state to initialize. Given such a simple application, ordinary restart technique are good enough. For the same reason, even when Rx takes a checkpoint every 50ms, the overhead is still small, less than 11%. But even with such frequent checkpoints, Rx's recovery time is still slightly higher than restart, which indicates for CVS-like servers, restart is a better alternative in terms of recovery time. But note that restart is not failure transparent to clients, and, if the bug-exposing request is resent again by the client after the failure, the same bug (especially deterministic one) is very likely to happen again.

Rx does not hide software defects. Instead, Rx reacts only after



(a) Throughput **(b) Average Response Time**

Figure 5: Throughput and average response time with different bug arrival rates

a defect is exposed. In addition, Rx's failure and recovery experiences provide programmers with extra information to diagnose the occurring or occurred software defects. For example, for CVS, Rx is able to avoid the bug by delaying the recycling of recently freed buffers during re-execution. Programmers then should investigate more in the direction of double-free or dangling pointers.

6.2 Recovery Performance

We have compared Rx with restart in terms of performance during recovery. As shown in Figure 4, Rx maintains throughput levels close to that of the baseline case. At the time of bug occurrence (at 7 seconds from the very beginning), the server throughput drops by 33% and the average response time increases by a factor of two for only a very short period of time (17-161 milliseconds). Therefore, a bug occurrence imposes only a small overhead, and has a minimal impact on overall throughput and response time. Restart, on the other hand, has a 5 second period of zero throughput. It services no requests during this period, so there are no measurements for response time. Once Squid has restarted, there is a spike in response time because all of the clients get their requests satisfied after a long queuing delay. Because Squid cannot service requests until it has completed the lengthy startup and initialization process, the whole program restart approach significantly degrades the performance upon a failure. Similarly, with a large real-world workload, we expect that the performance with restart will be even worse since the recovery time will become longer and many more requests will be queued, waiting to be serviced.

Figure 5 further illustrates the Rx's performance in the case of continuous attacks by malicious users who keep issuing bug-exposing requests. The throughput and response time of Rx remain constant as the rate of bug occurrences increases, whereas the performance of restart degrades rapidly. This is because Rx has very small recovery time, while restart spends a long time in recovery. Therefore, if such a bug were triggered by an Internet-wide worm [50] or a malicious user, restart cannot cope. However, since Rx can deal with higher bug arrival rates, Rx can tolerate such attacks much better.

6.3 Rx Time and Space Overhead

Figures 6 shows the overhead of Rx compared to the baseline (without Rx) for various frequencies of checkpointing. The performance of Rx degrades somewhat as the checkpoint interval decreases, but the amount of degradation is small. For squid, both throughput and response time are very close to baseline for all tested checkpoint rates. This is because the network remains the bottleneck for all cases. For MySQL, the performance degrades slightly at small checkpoint intervals. Since MySQL is more CPU bound, the additional memory-copying imposed by frequent checkpoints causes some degradation. It is expected that as checkpoints are taken extremely frequently, Rx's overhead will become dominant. However, there is no need for very frequent checkpointing. As shown earlier, even when Rx checkpoints every 200 milliseconds, it is able to provide very good recovery performance. With such a checkpoint interval, the overhead imposed by Rx is quite small, almost negligible for Squid and only 5% for MySQL.

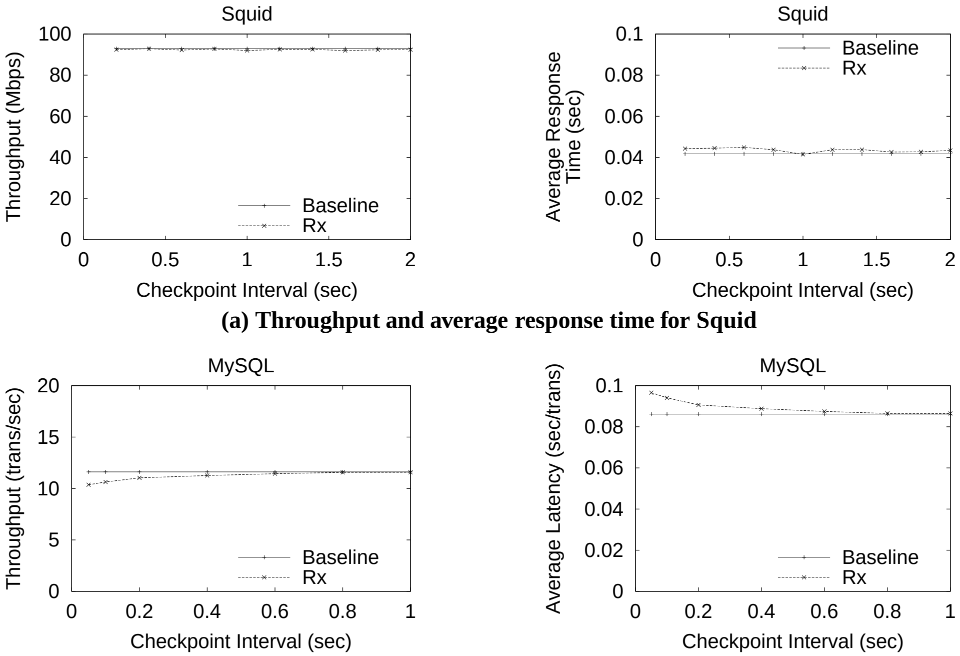
Apps	Rx Space Overhead (kB/checkpoint)		
	kernel	proxy	total
Squid	405.35	3.70	409.05
Mysql	300.00	0.16	300.16
Apache	460.00	3.60	463.60
CVS	42.22	2.89	45.11

Table 4: The average space overhead per checkpoint

Table 4 shows the average memory space overhead of Rx per checkpoint. The space overhead of Rx for each checkpoint is relatively small (45.11-463.60kB). It mainly comes in two parts: updates made during the checkpoint interval and the proxy message buffers. For the first part, Rx uses copy-on-write to reduce space overhead. For the second part, since Rx only records requests in the normal mode and request sizes are usually small, the proxy does not occupy much memory per checkpoint. Therefore, if 2-3MB of space can be used by Rx, Rx is able to maintain 5-20 checkpoints: enough for our recovery purpose.

6.4 Benefits of the Failure Table

Figure 7 reports the server recovery time with Rx when the server encounters the bug for the first time and for subsequent times in the same run. The results show that the failure table can effectively reduce the recovery time when the same bug/failure occurs again. For example, to deal with the first time occurrence of the buffer overflow bug in Squid, Rx applies message reordering, delaying free + message reordering, padding + message reordering sequentially in three consecutive re-execution trials, and finally avoid the bug at the third re-execution. The entire recovery process lasts around 216.7 milliseconds. However, for any subsequent occurrences of the same bug, which can be located in the failure table, Rx applies the correct environmental changes (padding + message reordering) in the first re-execution attempt, thus the recovery time is reduced to 94.7 milliseconds. For CVS, the failure table also helps to reduce the recovery time from 25 milliseconds to 16.9 milliseconds. For MySQL, the data race bug is avoided at the very first try with message reordering and therefore there is no difference between the first bug occurrence and subsequent ones.



(a) Throughput and average response time for Squid

(b) Throughput and average response time for MySQL

Figure 6: Rx overhead in terms of throughput and average response time for Squid and MySQL. In these experiments, we do not send the bug-exposing request since we want to compare the pure overhead of Rx with the baseline in normal cases.

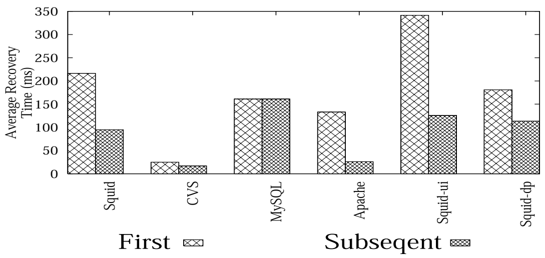


Figure 7: Rx recovery time for the first and subsequent bug occurrences

7. RELATED WORK

Our work builds on much previous work. Due to space limitations, this section only briefly describes those works that are not discussed in previous sections.

The idea of using checkpointing to provide fault tolerance is old [9], and well known [23, 42]. These checkpoints may be done to disk [19, 30, 32, 59], remote memory [2, 60, 39], or non-volatile or persistent memory [35]. These checkpoints can be provided with relatively low overhead. If there are messages and operations in flight, logging is also needed [6, 34, 33, 31]. After failure, many, but not all, errors can be avoided by reattempting the failed computation [26]. To deal with resource exhaustion or operating system crashes, monitoring, logging and recovery can be done remotely via support by special network interface cards [8]. In some cases, great care is taken to ensure *deterministic* replay [22, 45, 24]. However, unlike *deterministic* replay used by other techniques, we are purposely and systematically perturbing the re-execution environment to avoid determinism. As such, we have requirement and can use more lightweight checkpoints. Additionally, by changing environments, we can tolerate faults which simple re-execution cannot.

Failure-Oblivious Computing [43] 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 (1-8x slowdown [43],) and may introduce unpredictable behavior due to its potentially *unsafe* modifications to the memory interface. The recently proposed reactive immune system [48] 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) [38] 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 [13] 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 [28], and StackGuard [21]. 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 [20] and SafeMem [40]. Beyond memory bugs, it is also possible to detect deadlock and races.

Our proxy is similar to the shadow drivers used in [54], 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, we 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 we make are similar to noisemakers [51], except that, instead of trying to spur non-deterministic bugs into occurring, we are attempting to prevent deterministic and non-deterministic bugs by finding a legitimate execution path in which they simply do not arise.

8. CONCLUSIONS AND LIMITATIONS

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 modified execution environment. It can deal with both deterministic and non-deterministic bugs, and requires few to no modifications 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.

Our 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. In contrast, the two tested alternatives, a whole program restart approach and a simple rollback and re-execution without environmental changes, cannot recover the three servers (Squid, Apache and CVS) that contain deterministic bugs, and only have a 40% recovery rate for the server (MySQL) that contains a non-deterministic concurrency bug. These results indicate that applying environmental changes is crucial to survive software failures caused by common software defects, especially deterministic bugs. In addition, Rx also provides fast recovery within 0.017-0.16 seconds, 21-53 times faster than the whole program restart approach for all but one case (CVS). With Rx, clients do not experience any failures except a small increase in the average response time for a very short period of time. To provide such fast recovery, Rx imposes small time and small space overheads.

There are several limitations that we wish to address in our future work. First, we are trying to evaluate Rx with more server applications containing real bugs under various workloads. Second, currently the Rx's proxy is implemented at the user level. To improve performance, we plan to move it into the kernel, thereby avoiding context switches and memory copying. Third, we plan to extend Rx to support multi-tier server hierarchy as described in Section 4. This is relative easy since Rx already works with a database server (MySQL), a web server (Apache), and a Web proxy server (Squid). Fourth, our experiments so far have evaluated only I/O bound applications such as network servers whose availability is of critical importance. We plan to evaluate the Rx's overheads on computation intensive applications, and we expect the overheads are likely to be higher. Finally, we 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. Like almost all previous solutions, Rx cannot guarantee recovery from all software failures. For example, as we discuss in Section 4, neither semantic bugs nor resource leaks can be directly addressed by Rx. Also, as described in Section 2, 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. While similar rare cases can also appear in many previous solutions, it is still worthy addressing by using more rigorous dynamic integrity and correctness checkers as Rx's sensors. This is currently an active research area with many recent innovations. Additionally, Rx cannot deal with latent bugs–bugs in which the fault is introduced at a time long before any obvious symptoms. As discussed by Chandra and Chen [18], this problem is general to all checkpoint-based recovery solutions. Fortunately, this is a rare case, as a previous study [27] shows that most errors tend to cause quick crashes.

9. ACKNOWLEDGMENTS

The authors would like to thank the shepherd, Ken Birman, and the anonymous reviewers for their invaluable feedback. We appreciate useful discussion with the OPERA group members. This research is supported by IBM Faculty Award, NSF CNS-0347854 (career award), NSF CCR-0305854 grant and NSF CCR-0325603 grant.

10. REFERENCES

[1] L. Alvisi and K. Marzullo. Trade-offs in implementing optimal message logging protocols. In *Proceedings of the 15th ACM Symposium on the Principles of Distributed Computing*, May 1996.

[2] C. Amza, A. Cox, and W. Zwaenepoel. Data replication strategies for fault tolerance and availability on commodity clusters. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, Jun 2000.

[3] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12), 1985.

[4] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the 1st International Computer Software and Applications Conference*, Nov 1977.

[5] J. F. Bartlett. A NonStop kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles*, Dec 1981.

[6] K. P. Birman. *Building Secure and Reliable Network Applications*, chapter 19. Manning ISBN: 1-884777-29-5, 1996.

[7] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. In *Proceedings of the 1998 International Computer Performance and Dependability Symposium*, Sep 1998.

[8] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. Remote repair of operating system state using backdoors. In *Proceedings of the 2004 International Conference on Autonomic Computing*, May 2004.

[9] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the 9th Symposium on Operating Systems Principles*, Oct 1983.

[10] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1), 1989.

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

[12] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, Jun 2002.

[13] 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*, Dec 2004.

[14] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, Feb 1999.

[15] M. Castro and B. Liskov. Proactive recovery in a Byzantine-Fault-Tolerant system. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, Oct 2000.

[16] CERT/CC. Advisories. http://www.cert.org/advisories/.

[17] S. Chandra and P. M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, Jun 2000.

[18] S. Chandra and P. M. Chen. The impact of recovery mechanisms on the likelihood of saving corrupted state. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, Nov 2002.

[19] Y. Chen, J. S. Plank, and K. Li. CLIP: A checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE Supercomputing Conference*, Nov 1997.

[20] 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*, Jun 2003.

[21] 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*, Jan 1998.

[22] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Dec 2002.

[23] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Surveys*, 34(3):375–408, 2002.

[24] Y. A. Feldman and H. Schneider. Simulating reactive systems by deduction. *ACM Transactions on Software Engineering and Methodology*, 2(2):128–175, 1993.

[25] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. On the analysis of software rejuvenation policies. In *Proceedings of the Annual Conference on Computer Assurance*, Jun 1997.

[26] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliable Distributed Systems*, Jan 1986.

[27] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of Linux kernel behavior under errors. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, Jun 2003.

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

[29] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, Jun 1995.

[30] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, Aug 1988.

[31] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and check-pointing. *Journal of Algorithms*, 11(3):462–491, 1990.

[32] K. Li, J. Naughton, and J. Plank. Concurrent real-time checkpoint for parallel programs. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Princiles & Practice of Parallel Programming*, Mar 1990.

[33] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, Oct 2000.

[34] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *Proceedings of the 16th Symposium on Operating Systems Principles*, Oct 1997.

[35] D. E. Lowell and P. M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical report, CSE-TR-410-99, University of Michigan, Jul 1998.

[36] E. Marcus and H. Stern. *Blueprints for High Availability*. John Willey & Sons, 2000.

[37] D. Mosberger and T. Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[38] D. Patterson, A. Brown, P. 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, Technical Report UCB//CSD-02-1175, U.C.Berkeley, Mar 2002.

[39] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[40] 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*, Feb 2005.

[41] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[42] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computer Surveys*, 10(2):123–165, 1978.

[43] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, Dec 2004.

[44] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th Symposium on Operating Systems Principles*, Oct 2001.

[45] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, May 1996.

[46] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, Dec 1999.

[47] D. Scott. Assessing the costs of application downtime. Gartner Group, May 1998.

[48] 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*, Apr 2005.

[49] S. Srinivasan, C. Andrews, S. Kandula, and Y. Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX 2004 Annual Technical Conference*, Jun 2004.

[50] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, Aug 2002.

[51] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proceedings of the 2nd Workshop on Runtime Verification*, Jul 2002.

[52] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[53] M. Sullivan and R. Chillarege. Software defects and their impact on system availability – A study of field failures in operating systems. In *Proceedings of the 21th Annual International Symposium on Fault-Tolerant Computing*, Jun 1991.

[54] 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*, Dec 2004.

[55] G. Trent and M. Sake. Webstone: The first generation in http server benchmarking, 1995.

[56] W. Vogels, D. Dumitriu, A. Agrawal, T. Chia, and K. Guo. Scalability of the Microsoft Cluster Service. In *Proceedings of the 2nd USENIX Windows NT Symposium*, Aug 1998.

[57] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J. Barrera, and J. Gray. The design and architecture of the Microsoft Cluster Service. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, Jun 1998.

[58] Y.-M. Wang, Y. Huang, and W. K. Fuchs. Progressive retry for software error recovery in distributed systems. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing*, Jun 1993.

[59] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. M. R. Kintala. Checkpointing and its applications. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, Jun 1995.

[60] Y. Zhou, P. M. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Jun 1999.