FIRestarter: Practical Software Crash Recovery with Targeted Library-level Fault Injection

Koustubha Bhat* Erik van der Kouwe† Herbert Bos† Cristiano Giuffrida†
Department of Computer Science
Vrije Universiteit Amsterdam, The Netherlands.
*{k.bhat}@vu.nl, †{vdkouwe, herbertb, giuffrida}@cs.vu.nl

***Abstract*—Despite advances in software testing, many bugs still plague deployed software, leading to crashes and thus service disruption in high-availability production applications. Existing crash recovery solutions are either limited to transient faults or require manual annotations to target predetermined persistent bugs. Moreover, existing solutions are generally inefficient, hindering practical deployment.**

In this paper, we present FIRestarter (Fault Injection-based Restarter), an efficient and automatic crash recovery solution for commodity user applications. To eliminate the need for manual annotations, FIRestarter injects targeted software faults at the library interface to automatically trigger error handling code for standard library calls already part of the application. In particular, when a crash occurs, we roll back the application state before the last recoverable library call, inject a fault, and restart execution forcing the call to immediately return a predetermined error code. This strategy allows the application to automatically bypass the crashing code upon such a restart and exploits existing error-handling code to recover from even persistent bugs. Moreover, since library calls lie pervasively throughout the code, our design provides a large recovery surface despite the automated approach. Finally, FIRestarter's recovery windows are small and frequent compared to traditional checkpoint-restart, which enables new optimizations such as the ability to support rollback by means of hybrid hardware/software transactional memory instrumentation and improve performance. We apply FIRestarter to a number of event-driven server applications and show our solution achieves near-instantaneous, state-preserving crash recovery in the face of even persistent crashes. On popular web servers, our evaluation results show a recovery surface of at least 77%, with low performance overhead of at most 17%.

I. Introduction

Minutes of downtime can cost millions of dollars for high-availability software services [1]–[3]. Despite much research on software testing over the past several decades [4]–[9], deployed software is still plagued by pervasive bugs [10], [11]. While inherent limitations due to lack of dynamic characteristics of applications constrain static checkers, dynamic bug finders like sanitizers and fuzzers either depend on test coverage or must deal with path explosion challenges. Therefore, even rigorous testing efforts fail to eliminate all bugs in software before deployment. Many such *residual* bugs lead to application crashes, severely impacting high availability of production services that underpin our global critical infrastructure. Moreover, recovering from downtime may lead to additional recovery bugs [1], [12].

To tolerate failures induced by residual bugs, deployed services typically automatically restart after a crash or use some form of service replication (e.g., replicating worker processes or entire server applications). Unfortunately, all such simple solutions still result in loss of application state and have significant impact on several classes of applications. Applications that only retain short-lived state, such as web servers, experience service disruption. For example, a client request causing a target web server's worker process to crash will disrupt all the ongoing client connections handled by the same process. Applications with long-lived ephemeral state, such as caching servers, experience at least runtime performance loss. Applications with long-lived non-ephemeral state, such as database servers, experience data loss or, at best, prompt lengthy recovery actions to bring the service back to a sane state [13], [14]. Furthermore, crashes caused by software bugs keep occurring until they are thoroughly diagnosed and fixed, often requiring substantial manual effort.

Crash Recovery Techniques	*Persistent faults?*	*No annotation?*	*Recovery surface*	*Recovery latency (in seconds)*	*Perf. overhead*
Nooks [15]	✗	✓	Kernel extns.	-	<60%
Microreboot [16]	✗	✓	Managed code	<1	>2%
Shadow drivers [17]	✗	✓	Drivers	-	<3%
Recovery Domains [18]	✗	✓	Kernel:34-97%	-	8-560%
Rx [19]	✓	✗	ENV influenced	≈0.5	<5%
ASSURE [20]	✓	✗	Rescue-pointed	≈0.1	<7.6%
REASSURE [21]	✓	✗	Rescue-pointed	<1	<115%
HAFT [22]	✗	✓	90.2%	<1	200%
OSIRIS [23]	✓	✓	OS units: ≈60%	<1	≈5%
FIRestarter	✔	✔	**>77%**	**≈0.1**	**<17%**

TABLE I: Comparison of software crash recovery techniques.

Crash recovery solutions reduce service disruption by masking the impact of crashes for end users. Existing solutions either perform automatic patch generation [24]–[26] (which incurs high recovery latencies and still lose state) or employ some form of checkpoint-restart [27]–[29]. Solutions based on checkpoint-restart are promising, but, as shown in Table I, are currently limited in important ways. Several only target transient faults [16], [22], [30], while software bugs are often persistent (e.g., deterministic memory safety bugs). Others handle persistent bugs, but require manual annotations [20], [21], [31] or specific system design [23]. Moreover, prior solutions typically incur high performance overhead (even 2x

or more) due to large checkpointing windows [20], [21], [32].

In this paper, we present FIRestarter (Fault Injection-based Restarter), which offers *automatic, efficient, state-preserving, and instantaneous* crash recovery. Our key insight is that we repurpose library-level fault injection—a classic dependability assessment technique [33]—to automate crash recovery. Library interfaces report errors in interactions with the environment, which applications need to handle. After recovering a checkpoint, we inject targeted library-level software faults to trigger error handling code already in the application. Our key strategy is to piggyback on error-handling code *already* existing in the target application, for crash recovery. This allows us to recover even from persistent crashes in a manner that is specified by the application itself. By transparently converting application crashes into library error conditions that the application can already handle, we eliminate the need for manual annotations and also avoid recovery bugs [12]. Moreover, the crash recovery surface and its quality will be as good as the application's handling of errors from its environment. This also means that any developer effort to improve the quality of error-handling code automatically translates to higher-quality crash recovery (and vice versa).

We apply FIRestarter to event-driven server applications, which are key building blocks of modern production services and particularly amenable to efficient crash recovery [27]. Our results show that FIRestarter can effectively and instantaneously mitigate the service disruption caused by crashes in production. We are the first to efficiently and automatically enforce a large crash recovery surface against persistent bugs.

Contributions: We make the following contributions:

1) We propose a novel crash recovery model that exploits the contract between applications and standard libraries as well as targeted library-level software fault injection to automatically bypass crashing code paths.
2) We propose FIRestarter, a design and implementation of our recovery model decomposing a target application into a number of crash transactions. Our full FIRestarter prototype implementation is open source and available at https://github.com/vusec/firestarter.
3) We explore optimizations, with compiler-based instrumentation that can *automatically* and *adaptively* select checkpointing implementations (hardware-based and software-based) at runtime.
4) We apply our prototype to event-driven server applications. Our evaluation shows that FIRestarter can significantly increase the crash recovery surface against persistent bugs (as high as 84% on Nginx), with no need for annotations and with practical performance overheads (as low as 17% on Nginx).

Listing 1: Running example of a library call interval with error handling in Nginx.

```
1  ...
2  ret_s = setsockopt(s, SOL_SOCKET, SO_REUSEADDR,
3          (const void *) &reuseaddr, sizeof(int));
4  if (ret_s == -1) { // Error handling
5        LOG_ERROR("setsockopt() failed");
6        if (ngx_close_socket(s) == -1) {
7            LOG_ERROR("ngx_close_socket failed");
8        }
9        return NGX_ERROR;
10 }
11
12 ...
13 ret_b = bind(s, ls[i].sockaddr, ls[i].socklen);
14 if (ret_b == -1) { // Error handling
15   err = ngx_socket_errno;
16   ...
17   LOG_ERROR("bind() failed");
18
19   if (ngx_close_socket(s) == -1) {
20       LOG_ERROR("ngx_close_socket_n failed");
21   }
22   if (err != NGX_EADDRINUSE) {
23       return NGX_ERROR;
24   }
25   failed = 1;
26   continue;
27 }
28 ...
```

II. Fault Model

FIRestarter enables recovery from crashes due to both transient and persistent faults in high-availability event-driven server applications. Transient faults include issues such as temporary hardware failures and race conditions, while persistent faults include common software faults due to programming errors such as null pointer dereferences, buffer overflows, and dangling pointers. Generally, retrying the failed operation suffices to recover from transient faults whereas surviving persistent faults is much harder. Effective persistent fault recovery requires bypassing the faulty code and gracefully disabling the affected path without breaking the program, or in the worst case, gracefully exiting the program.

With our focus on survivability of applications, we assume a fail-stop fault model (even for persistent hardware faults) like most of the research in the area [19]–[21], where the fault crashes the application immediately and, rely on supplementary methods that can convert fail-silent bugs into fail-stop ones [34]–[36] for broader applicability. Such crashes form an important class of failure in common software, particularly in the context of high availability. Modern compilers such as GCC and LLVM provide options to instrument programs to detect spatial and temporal memory corruption bugs and crash the program immediately. Similarly, defensive coding practices, such as the use of assertions to verify application state invariants, also help to guard against fail-silent faults.

III. Overview

FIRestarter hardens unmodified applications against transient and persistent software faults that would otherwise be fatal. It uses compiler transformations to create a lightweight checkpoint at each library call, and rolls back to an earlier

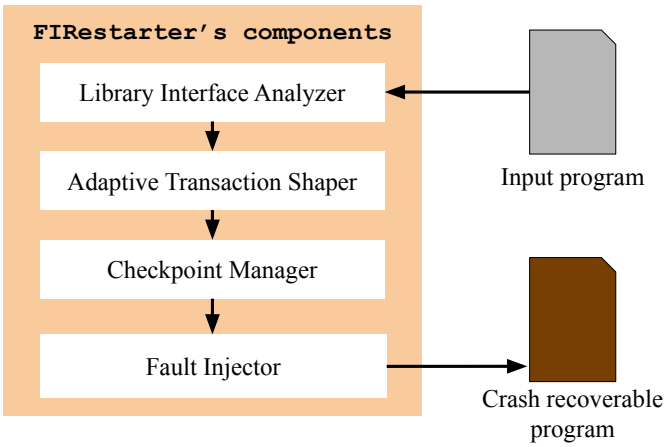

Fig. 1: Constituent components of FIRestarter.

library call in case of crashes. For transient faults, we roll back changes and re-execute the faulting code. If this fails, we assume a persistent fault and inject a fault into the library call to bypass the faulting code. Our target applications are event driven servers, that typically run in an infinite loop accepting incoming requests, invoking appropriate request handlers, and returning their results.

Listing 1 shows our running example, a code snippet from the Nginx web server, abridged and annotated for better readability. On Line 2, the server sets a socket option by calling the standard library function `setsockopt()`. It only invokes the `bind()` library function on Line 13 if the previous call was successful. Error handling code for both the calls, log the failed call and close the socket. In the remainder of this section, we explain at a high level how FIRestarter instruments the code to survive faults, filling in the details in later sections.

FIRestarter's transformation procedure consists of four distinct passes, as illustrated in Figure 1. First, the *Library Interface Analyzer* assigns appropriate pre-defined *compensation actions* to all library call sites found in the application. Based on operational and failure semantics easily obtainable from library documentation, compensation actions revert their effects and allow injecting library faults. In the example, we can compensate by closing the socket to revert the effects of `setsockopt()` and return the value of -1 to indicate an error. Next, the *Adaptive Transaction Shaper* statically places hooks in the code to allow the system to find suitable transaction boundaries at runtime. In the example, it places hooks to mark the start of transactions *right after* the `setsockopt()` and `bind()` library calls and places end markers *just before* them, encapsulating the code region between the two library calls in a transaction. After this pass, the *Checkpoint Manager* adds code to perform lightweight checkpointing at library calls, enabling transactions to be rolled back when necessary. FIRestarter supports checkpoints based on both hardware transactional memory (HTM) and software transactional memory (STM). For example, the Checkpoint Manager may add `XBEGIN` and `XEND` instructions to initiate and end hardware transactions respectively, or instrumentation to track all the `store` operations in an undo log for software transactions. Finally, the *Fault Injector* adds instrumentation for handling persistent crashes during the transactions. It uses information gathered by the Library Interface Analysis pass to find an alternative path to execute after the Checkpoint Manager rolls back the state after a crash. To detect crashes, it deploys signal handlers which handle and proxy fatal signals (like `SIGSEGV` on Unix systems) to invoke crash recovery. In the example, the Fault Injector adds instrumentation that alters the return value of `setsockopt()` to -1 to divert the execution path when a crash occurs, to the error handling code, which eventually tries closing the socket and returns `NGX_ERROR`. In summary, the FIRestarter transformations enable an application to roll back crashes and inject non-fatal library errors instead, relying on the application's error handling code to cleanly handle the error and skip the faulty code.

IV. Reverting Crashes

FIRestarter offers two key advances that together enable efficient stateful recovery from both transient and persistent faults: lightweight checkpointing using hybrid hardware/-software transactions to enable efficient crash recovery, and library-level fault injection to bypass persistently faulty code. We discuss our checkpointing mechanism in this section and fault injection in Section V.

To revert crashes, we need to make regular checkpoints, which allows us to revert to the last checkpoint whenever a crash occurs. Afterwards we can re-execute the code with no loss of state. Regular checkpointing incurs significant runtime overhead, so it is critical to do this as efficiently as possible. To this end, we use memory transactions, which do not store the entire state, but rather keep track of memory stores and revert them to restore the last checkpoint. This approach can efficiently achieve high checkpointing frequencies [27].

A. Hybrid Checkpointing

Logging each store in software is slow. Fortunately, the Intel TSX instruction set extensions allow us to leverage the L1 cache for this purpose, providing instructions to begin or commit a transaction to memory and automatically roll back if anything goes wrong. While originally developed for managing concurrency, we repurpose it here for efficiently recovering from software faults. We will refer to this mechanism as simply Hardware Transactional Memory (HTM). Since the size of the cache limits HTM, it will automatically roll back if a transaction grows too large. Moreover, unpredictable events such as conflicts between CPU cores within the same cache lines and interrupts also cause a transaction to abort. Therefore, not all code can be run with HTM.

While prior work has also used TSX for reliability [22], it falls back to unprotected execution after an abort. We instead propose a hybrid approach, which falls back to software transactional memory (STM) instead. To support STM semantics, we rely on a common undo log-based design [27], which instruments the specified code region to track all the stores to memory and save the old data in the undo log.

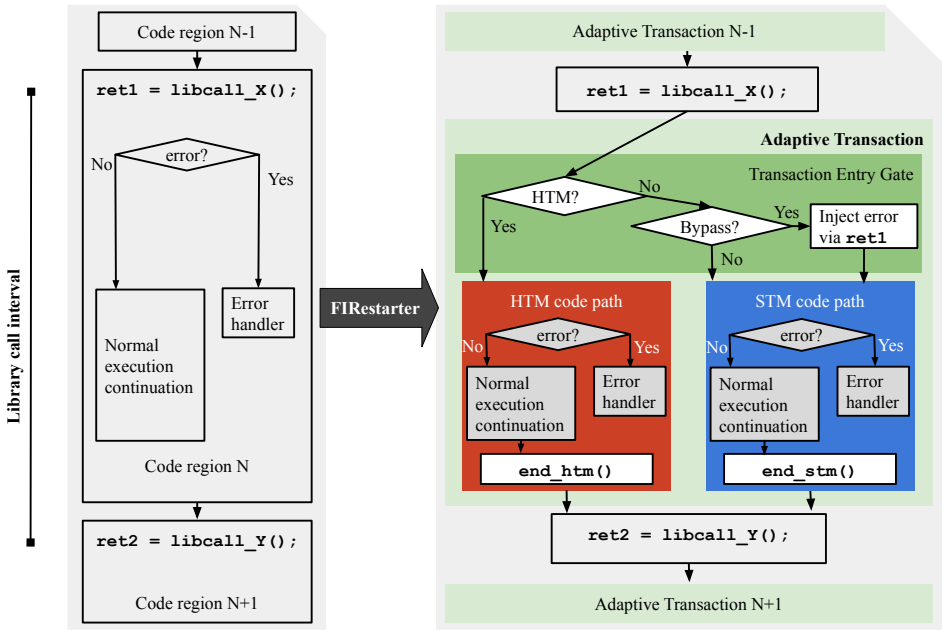

Fig. 2: FIRestarter transforming a program to introduce an adaptive transaction between two consecutive library calls.

To roll back, we walk the undo log in reverse order and restore each modified memory location to its original value. We additionally instrument the code region to save (and restore when necessary) the contents of all registers in memory, using a method akin to `glibc`'s `setjmp()` and `longjmp()` functions [37].

B. Switching between HTM and STM

We get the best of both worlds: where possible, HTM yields good performance, while where necessary, STM is available to maximize the recovery surface. However, static analysis cannot determine which transactions will work with HTM and which will be aborted prematurely, because: (1) it cannot determine the complete call graph due to the possibility of indirect function calls, (2) it cannot predict loop iteration counts, and (3) it cannot estimate how many page faults or other interrupts may occur at runtime. These issues stem from fundamental limitations of static analysis, including the inability to solve complex aliasing problems and the lack of knowledge about the input and control variables affected by the environment. We must decide which type of transaction to use, at run-time.

To switch between HTM and STM, we introduce *adaptive transactions*. At the locations where a transaction can start, we insert a *transaction entry gate* that allows us to switch between HTM and STM (see Figure 2). As we will explain in Section V-A, we start transactions right after a library call and commit them right before the next. We clone all code between consecutive library calls, instrumenting one copy with the `XBEGIN` and `XEND` instructions for HTM and the other copy with undo logging for STM. The transaction entry gate dynamically selects one of the variants.

Note that each runtime transaction must stick to the same selected checkpoint method even across function boundaries along its execution path, until it completes or aborts. So, function entry points must select the transaction type of their caller, and return sites need to apply the callee's transaction type at return time. At these locations we insert *flow switches*, which select the correct execution path based on a global variable, set to the current transaction type by the transaction entry gates. We ensure that the code uses the same local variables regardless of the selected clone so that data remains consistent even when switching inside a function.

C. Dynamic Transaction Adaptivity

Our design so far offers a mechanism for high-frequency checkpoints that can dynamically switch between HTM and STM, but also requires a policy to determine when to use which. This policy is critical for performance. Although HTM has much better performance than STM, if HTM aborts, it requires not just rollback, but also code re-execution using STM to determine whether HTM aborted due to resource constraints, or due to a real crash. A naive policy of attempting HTM first each time is inefficient. However, permanently switching to STM after the first abort is also inefficient, as any transaction can randomly abort, for example due to an interrupt.

Figure 3 shows the percentages of HTM transactions that abort and corresponding throughput degradation we observe in a FIRestarter-protected Nginx server. Using the naive policy of always attempting HTM, 20% of HTM transactions abort, resulting in substantial overhead. Specifically for the transactions after the library calls `malloc()`, `posix_memalign()` and `fcntl64()`, we observe 82%, 47% and 15% HTM aborts respectively. We attribute this behavior to library calls that are followed by large memory operations (hence exceeding HTM

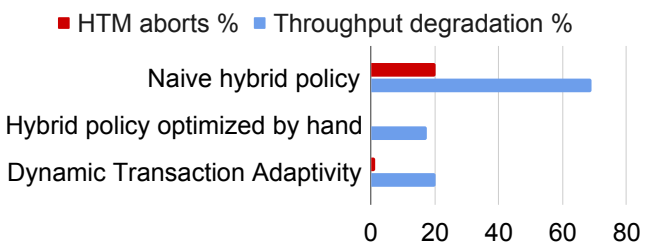

Fig. 3: Impact of adaptive transaction policies on HTM abort percentage and throughput degradation on Nginx.

limit), such as initializations following memory allocation. With this, throughput degradation of Nginx shoots up to 69%.

We could mark these specific code regions manually to let their transactions take the STM route directly without trying HTM at all and avoid those unnecessary HTM aborts. When optimizing by hand in such a way, the HTM aborts drop to 0.0002% (hence invisible in Figure 3) and the throughput degradation drops to 18%, less than a third of the original overhead. Clearly, avoiding HTM aborts helped reduce the overall performance impact significantly. We observe this behavior on other applications too and discuss this in Section VI.

Following this observation, we devise a dynamic adaptation policy and make the transactions automatically decide when to attempt HTM and when to directly switch over to STM. Each transaction monitors its HTM abort rates at runtime and switches over to STM permanently when exceeding an abort tolerance threshold. With this approach, transactions that slow down the system due to many aborts (such as large-scale memory initialization) switch to STM, while transactions where resource limits are rarely exceeded continue to try HTM before possibly falling back to STM on a case-by-case basis.

In Figure 3, we show the impact of FIRestarter with our dynamic transaction adaptation policy on throughput of Nginx server when we set the HTM abort tolerance threshold to 1% with a monitoring sample size of 128. In other words, for every 128 HTM aborts, the transaction checks the ratio of the number of HTM aborts to the total number of executions of the transaction, and decides to permanently switch to STM when the ratio exceeds 1%. With this policy, FIRestarter incurs a performance overhead of 21% on Nginx, which is very close to the best that we could achieve using manual decision marking. Our policy scores good performance without any manual annotations.

V. Preventing Further Crashes

So far we described a system that can efficiently roll back execution after a crash and try again. This is effective against transient faults, but, in the face of persistent faults, will result in an endless crash-recovery loop. To mitigate persistent faults in the absence of a patch or manual annotations, FIRestarter injects a fault in a suitable library call, piggybacking on existing error handling code to bypass the faulty code.

Library calls often interact with the environment, possibly resulting in errors, which are typically reported to the caller through well-documented mechanisms. A well-written program shall handle such errors appropriately. So, library calls are the ideal means to inject faults that will divert the protected program's execution away from the faulty code which caused the crash and prompted an execution rollback.

	Execution path diversion		
Recoverability	*possible*	*NOT possible*	***Total***
Operation reversible	23	0	23
No reversion needed	9	26	35
Operation deferrable	5	2	7
State restoration needed	12	8	20
Irrecoverable	12	4	16
Total	61	40	101

TABLE II: Library functions in Nginx, Apache, Lighttpd, Redis, and PostgreSQL classified based on recoverability and ability to divert (faulty) execution via fault injection.

To inject a fault, we need to perform several steps: (1) roll back the execution to a suitable library call, (2) revert the operation of the library call, and (3) make the library call report a suitable error according to its interface documentation. We roll back execution using the same mechanisms proposed for transient faults in Section IV and will describe how to determine which checkpoint to restore in Section V-A. Selecting the right restore point is tightly coupled with reverting operations and therefore discussed in the same section. We discuss how to inject the fault in Section V-B and finally discuss the impact on our running example in Section V-C.

A. Finding a Suitable Restore Point

The Library Interface Analyzer pass identifies which library calls are useful for recovery. We aim to minimize transaction sizes to reduce checkpointing overhead due to transaction aborts (for HTM) as well as undo log size (for STM) and because it is hard to roll back any operation in the transactions with externally visible effects. However, we need the ability to inject faults to divert execution away from a crash. Therefore, the shortest feasible (and therefore most desirable) transaction granularity is the interval between consecutive interactions with the environment (i.e., library calls).

We perform static and dynamic analysis on the target application to determine how it uses shared libraries. Learning the error handling semantics from their documentation, we classify library calls into several *recoverability classes*: (1) *Operation reversible*, when a revert operation for the call exists (e.g., `munmap()` can revert an `mmap()` library call), (2) *No reversion necessary*, when the call is idempotent and does not modify application state (e.g., `getpid()`), (3) *Operation deferrable*, when we can defer the effects of a successful call until after its transaction successfully commits (e.g., usually `free()`), (4) *State restoration necessary*, when the call is reversible if specific runtime state prior to the call execution can be restored (e.g., by checkpointing call arguments), and (5) *Irrecoverable*, when a call has side effects that go beyond the application process memory and have externally visible consequences (e.g., `write()`, `send()`).

Fig. 4: Transformation of our running example code snippet by FIRestarter's components; a: original code; b: add hook for fault injection; c: clone code, instrumenting results with transaction mechanisms; d: add fault injection code.

Table II provides statistics on recoverability classes of the standard library functions used by common server applications (Nginx, Apache, Lighttpd, Redis and PostgreSQL) and the ability to divert (faulty) execution via fault injection. As the table shows, only 16 out of 101 library functions have operational semantics *unsuitable* for crash recovery. Note that, in our analysis, we use the most conservative definition for the *irrecoverable* class (no external side effects allowed), but more less-conservative approximations (e.g., allowing a socket `write()` to produce network-visible side effects that can be masked by injecting a network error) may enable a larger recovery surface.

For all the other (85 out of 101) suitable library functions, we created wrappers to perform *compensation actions* which revert the effects of the functions and allow for fault injection afterwards. This took 503 source lines of C code. Note that this is a one-time effort to cover widely used standard library functions and need not be repeated to extend the protection to additional programs. Statically linked with the application, our instrumentation facilitates their invocation when necessary.

Using static analysis, we then trace the use of the return value to determine which library call sites are followed by error handling code and are thus suitable for fault injection-based crash recovery. Table II shows that 61 out of 101 calls are checked for errors, allowing them to be used at the start of a transaction to automatically divert execution upon restart.

We cannot statically determine where a transaction ends because it depends on run-time control flow, but we do use the results of the static analysis to decide on the transaction size at runtime. Library calls typically indicate failure by their return values. A few library calls, such as `strlen()`, cannot report errors; in other cases, the application may ignore return values of library calls that do report errors. In both these cases, there would not be an error handler. Therefore, such call sites are unsuitable for fault injection because we cannot change the execution path. Nonetheless, if such calls can be reverted (or are simply idempotent), the Adaptive Transaction Shaper extends the transaction at runtime to include them and reverts the results if the transaction is rolled back. If they cannot be reverted, the Shaper ends the transaction and the application cannot recover until the next library call amenable to fault injection.

B. Fault Injector

The Fault Injector adds instrumentation to inject library faults to divert execution upon restart. We instrument each transaction to perform a *compensation action*, reverting the library call in which a fault is to be injected, and then modify the runtime return value of the library call to report an error to the application. The altered return value causes the program to handle the injected error, repurposing the error handler

for recoverability. The transaction entry gate facilitates this operation, as shown in figure 2.

Reverting the effects of library calls poses two challenges: (1) effects of library calls are not necessarily limited to the same process, and (2) the library call is outside our transactions and hence we must explicitly undo any of its memory modifications within the process. We use the operational semantics of the various standard library calls, obtained by the Library Interface Analyzer, to preserve any application memory that the library calls may modify. After rolling back memory operations that occurred after the library call and running its compensation action, we also restore the library call-affected memory areas (and state in general).

Errors occurring in critical paths typically render an application non-functional. We define *critical paths* of an application as those code paths without which the application would not be able to perform any useful service. In terms of testing efforts, these are code paths that the high-priority test cases exercise. Typically, error handlers in critical paths either retry the failed operation indefinitely or ignore the failure and continue in the same code path anyway. They do not divert execution paths since they are critically required to be executed. For example, the Nginx web server calls `epoll_wait()` to wait for incoming HTTP requests. If this operation fails, there is no way around it but to retry. In contrast, error handlers in non-critical paths typically take an abortive approach since the operation does not fundamentally hinder the application's services. They take a distinct failure path to gracefully propagate the error to the user or divert execution towards other paths. For example, the Nginx web server calls `malloc()` to allocate memory when handling a client request. If this operation fails, Nginx detects an out-of-memory condition, gracefully aborts request handling, and returns an internal server error to the client before moving to the next request. Fortunately, fatal faults rarely occur in critical (and thus heavily tested) code paths of production software. We specifically aim to recover from residual faults that remain after extensive testing. Since these rarely occur in critical paths, for most cases, error handlers provide an effective way to bypass faulty code.

C. Running example

Figure 4 shows the code of our running Nginx example (part a) as well as the result of the LLVM [38] transformations that FIRestarter applies (parts b, c, and d) to enable crash recovery between two library calls, `setsockopt()` and `bind()` in this example. To protect the region between the library calls in Figure 4(a), we first determine the operation and failure semantics of the first library call, `setsockopt()`. The call belongs to the idempotent recovery class and returns -1 to indicate an error condition. In Figure 4(b), we add a conditional branch after the first library call. The global variable `tx_gate[]` for this site controls the branch. If the condition is true, it injects an error. Figure 4(c) shows the instrumentation to enable lightweight checkpointing. We clone the snippet and merge the local variables between the two copies. Afterwards, we update the labels and transaction entry gate to point to each other, depending on the runtime value of the chosen checkpointing mechanism in the global gate `tx_gate[]` for this site. For HTM, we add the `XBEGIN` and `XEND` instructions. The abort handler consults our dynamic transaction policies to potentially switch to STM. For STM, we instrument all the store instructions to update the undo log in memory, in addition to calling a hook that preserves the register state. Finally, as shown in Figure 4(d), the Fault Injector consults the Library Interface Analysis for error semantics and sets the value of `ret_s` to -1 in both the copies of the fault injection code block.

During the application's execution, if a crash occurs anywhere between the library calls, the following events occur. Inside the HTM code region, Intel TSX aborts the transaction, restoring the memory and the registers to the state just after the `setsockopt()` call (i.e., right at the `XBEGIN` instruction within `xbegin()`) and calls the abort handler specified by `xbegin()`. Our abort handler consults the dynamic transaction policy, which sets the gate to `DIVERT` and applies the compensation action for the library call. `setsockopt()` belongs to the recoverable recoverability class and the compensation action simply closes the socket. Based on the failure semantics for the call, the Fault Injector sets the return value of the library call to -1. When the execution resumes after the abort handler finishes, the execution path is diverted towards the error handling code, gracefully preserving the rest of the runtime state. Thus, FIRestarter converts a crash into an error condition that the application can gracefully handle.

	Nginx	*Apache*	*Lighttpd*
# unique transactions	78	75	136
# libcalls embedded within	102	468	17
# unique irrecoverable transactions	12	17	30
Unique recoverable transactions	**84.6%**	**77.3%**	**77.9%**

TABLE III: Runtime recoverable surface w.r.t standard test suite workloads of Nginx, Apache, and Lighttpd web servers.

VI. Evaluation

We evaluate FIRestarter along the following dimensions: (1) recoverable surface, (2) effectiveness in surviving faults, (3) crash recovery latency, (3) run-time performance overhead, and (4) memory overhead. We run all evaluations on an Intel i7-6700K with 16 cores at 4.00 GHz and 16 GB of DDR4 memory, running 64-bit Ubuntu 16.04 LTS Linux distribution.

A. Recoverable surface

We define the recoverable surface as the fraction of transactions where, upon a crash, FIRestarter can (1) restore to a sane state, and (2) can divert execution away from the persistent fault. The former depends on the amenability of the library call to recovery. In general, we consider code/calls with externally visible effects as irrecoverable since we cannot undo their effects by process-local operations (e.g, `send()`). The latter depends on the availability of suitable error handling code at the start of a transaction. To measure the runtime recoverable surface, we instrument and profile the Nginx, Apache and

	Fail-stop		Fail-silent		
Server	*Injected*	***Crashes recovered***	*Injected*	*Crashes Triggered*	***Crashes recovered***
Nginx	10	10	13	2	2
Apache	4	4	12	0	-
Lighttpd	41	29	11	0	-
Redis	10	9	10	0	-
Postgre	27	22	33	0	-

TABLE IV: FIRestarter's crash recovery effectiveness against injected persistent faults.

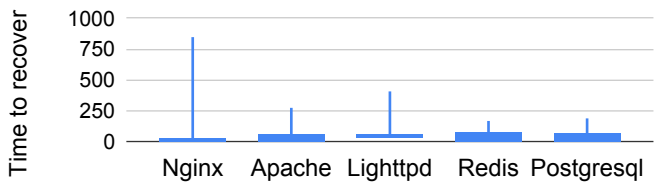

Fig. 5: Crash recovery latency in milliseconds.

Lighttpd web servers running their standard test suite workloads and measure the fraction of unique transactions that are recoverable.

Table III depicts the runtime recoverable surface for the three web servers. Certain library calls may be unsuitable for execution diversion based on library "faults" (e.g., the return value of `printf()` is typically ignored), but have no adverse effect on the application state. FIRestarter embeds such code regions in their last transaction. The table also shows the number of such calls. We observe that with FIRestarter, 84.6%, 77.3% and 77.9% of the runtime transactions of Nginx, Apache and Lighttpd servers respectively, are recoverable. Improved error handling in applications can further increase the recovery surface, except in case of *irrecoverable* transactions.

B. Survivability

To evaluate how effective FIRestarter is at crash recoverability, we perform fault injection experiments on the web servers Nginx, Apache, and Lighttpd. We use the HSFI [39] framework to inject persistent and fatal faults in every crash transaction after the server starts up, activating one fault per experiment. We use each server's standard test suite as the workload to drive our fault injection experiments.

We first inject the faults and then apply FIRestarter's compiler instrumentations to emulate a real world scenario where bugs remain in the source code while FIRestarter facilitates crash recovery for the compiled source code. Since we can only exercise those injected faults which are actually executed by the workload, we first perform program execution profiling to determine which basic blocks execute during the workload's run and instruct HSFI to inject faults in only those basic blocks—activating one fatal fault per experiment. By repeating the experiment, we trigger each injected fault.

In reality, bugs are not uniformly distributed over the code in a production-quality application. Faults in *critical* code paths that are necessary for a program's core behavior are unlikely to escape testing. So, faults representative of deployed software are typically in code triggered by rarely executed features or corner cases, the *non-critical* code paths. FIRestarter's goal is to protect exactly those least tested code regions likely to contain residual faults. Therefore, we measure the effectiveness of FIRestarter on non-critical code paths. We define critical paths using the programmer's intentions as observable from the error handling code. Specifically, we find that error handlers in critical code paths either exit the program or retry the same code path, implicitly assuming temporary failure. In contrast, in non-critical parts, the code branches off to handling code (e.g., to report an error), steering away from its original execution code path to continue the execution. Coverage of the remaining paths highly depends on the test suites. We found 10, 4, 41 and 27 transactions within non-critical paths of Nginx, Apache, Lighttpd web servers, and PostgreSQL database server respectively, while running their standard tests. The Redis key-value server (which, unlike memcached, does not require multithreading for parallelism), has 10 such transactions while running SET/GET workloads. We inject one fault in each unique transaction, in a randomly selected basic block. Table IV presents our results. As shown in the table, for Apache and Nginx, FIRestarter successfully performs crash recovery in 100% of the cases, bypassing the region with the fault and retaining both the runtime state and availability. The other tests reach success rates greater than 70%: Lighttpd successfully recovered from 29 of the 41 (70%) injected faults, Redis from 9 of 10 (90%), and PostgreSQL from 22 of 27 (81%). Since the remaining cases fall in the irrecoverable class (Table II), FIRestarter aborts the execution as intended.

Going beyond our fault model, we next instruct HSFI to inject non-fatal latent faults to observe FIRestarter's response to a variety of other software faults (including buffer overflows, or corrupted pointers, array indices, integers, and operators). Injecting 13 such faults in Nginx, 12 in Apache, 11 in Lighttpd, 10 in Redis, and 33 in PostgreSQL (one fault per run and in one transaction per run), only two of them eventually triggered crashes, both in Nginx. In the remainder of the cases, the corruptions sometimes caused deviations in test results. This is expected as FIRestarter handles crashes only. It successfully recovered from the crashes and successfully diverted execution using error handlers, demonstrating its effectiveness even for faults *beyond* our target fault model.

C. Recovery Latency

Figure 5 shows how the recovery latency varies across several fault-triggered executions of Nginx, Apache and Lighttpd servers. We injected fatal faults and measured the time between the invocation of FIRestarter's signal handler and handing the execution back to the application after the recovery. The typical latency is in the order of several tens of milliseconds. Depending on the execution context just before the crash, undolog-based recovery may take longer for restoring memory operations (including restoring the stack to its previous state), but even in the outliers, the latency remains under a second.

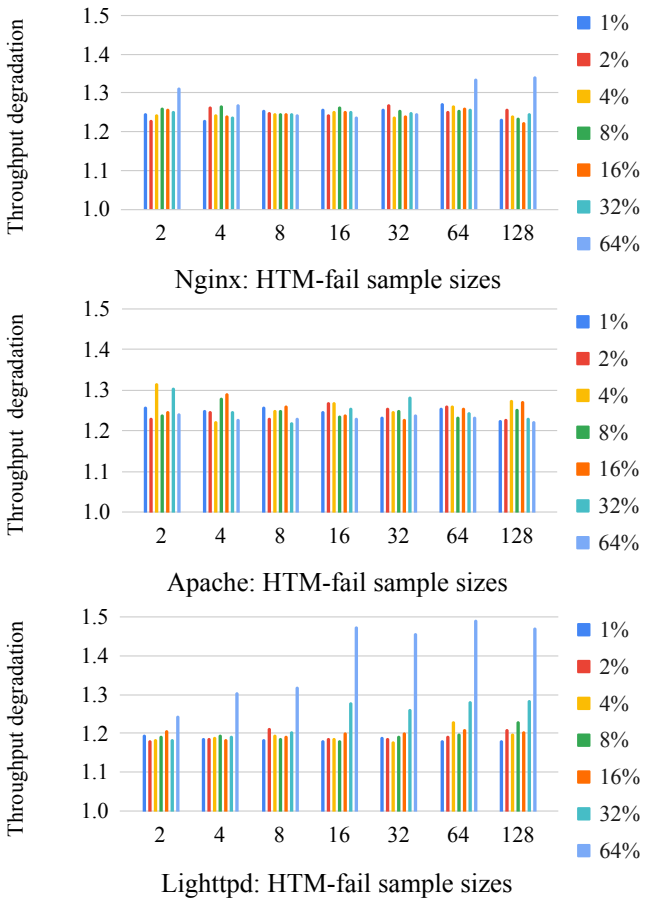

Fig. 6: Dynamic Tx Adaptation on web servers for different HTM failure thresholds of 1%-64%.

D. Performance overhead

FIRestarter dynamically adapts to its workload to maximize performance while retaining recoverability. Two factors primarily affect FIRestarter's performance overhead: (1) the amount of execution time spent in STM mode (which is slower than HTM), and (2) the number of HTM failures (which cost time due to execution rewinding) tolerated before switching to STM. Intuitively, the less time spent in STM, and the sooner the switch to STM where necessary, the lower the performance overhead. In other words, switching too soon and switching too late both harm performance. In order to understand these dynamics, we set up the following experiment. We subject the instrumented web servers to an onslaught of client requests from the standard ApacheBench benchmark suite and let FIRestarter track the number of executions and corresponding HTM failures for every protected region. We define two parameters, the *HTM abort threshold* and the *accounting sample size*. The threshold defines the maximum ratio of HTM failures to the number of executions of a protected region before we switch to STM. The sample size determines how often FIRestarter performs the threshold check. For example, with a sample size of 4, we perform the threshold check every 4 executions of a particular protected

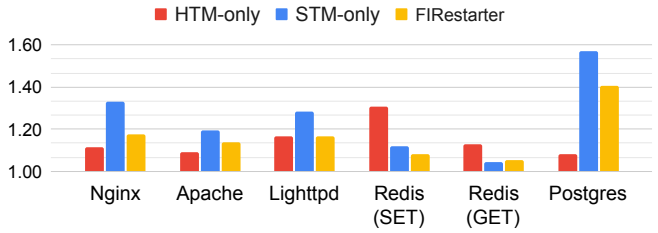

Fig. 7: Normalized runtime performance overhead of HTM, STM, and FIRestarter. FIRestarter significantly reduces the performance impact in comparison to STM-only.

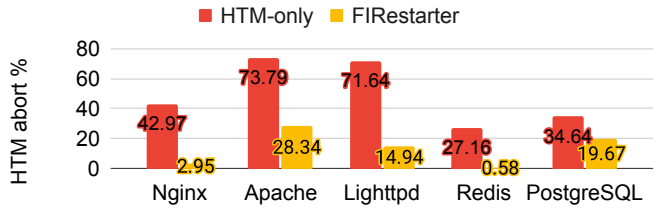

Fig. 8: HTM failures observed for HTM-only and FIRestarter. FIRestarter significantly reduces HTM aborts.

region. We run the ApacheBench workload as described above on each of the three web servers, with varying thresholds between 1% and 64% and varying sample sizes between 2 and 128. Figure 6 shows the recorded results for the three web servers during normal (crash-free) execution. The results show that the performance is sensitive neither to the sample size, nor to the threshold, although lower thresholds do perform slightly better. Overall, a threshold of 1% with a sampling size of 4 achieves the best performance.

Figure 7 shows a comparison of FIRestarter's performance overhead with HTM-only and STM-only approaches of recoverability. The HTM-only variant has all the library call intervals instrumented with HTM transactional boundaries and we set the HTM transaction abort handler to automatically fall back to executing the vanilla unprotected code. HTM aborts occur due to exceeding HTM resource limits, page faults and interrupts. So, this gives no recovery guarantees at all (even for transient faults) and is only included for performance comparison. The STM-only variant has all the library call intervals instrumented with undolog-based memory checkpointing [27], extended with support for register restoration. It provides full protection. We then run the `wrk` [40] benchmark for the web servers and the standard benchmarks for Redis and PostgreSQL to measure the runtime overhead under normal execution. We saturate the servers while running the client workloads on a separate machine connected to the server by a dedicated 100 Gbit/s network link. Figure 7 shows that compared to STM-only, FIRestarter brings down the performance overhead significantly. The runtime overhead of 17% on Nginx and Lighttpd, 14% on Apache and under 12% on Redis show that much better reliability is within the reach of reasonable performance overheads even for off-the-shelf software. Figure 8 shows that the percentage of HTM

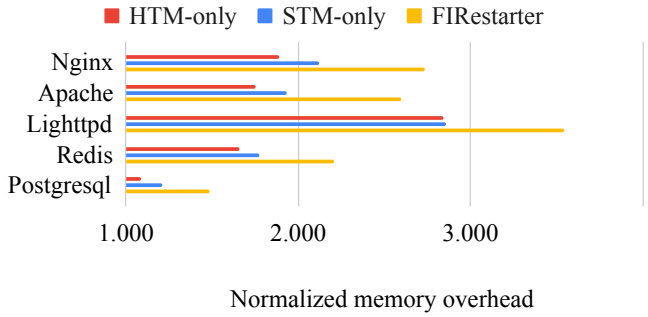

Fig. 9: Normalized mean memory overhead (RSS) of HTM, STM, and FIRestarter.

aborts drops drastically for all the listed applications. The smaller reduction in HTM failures for the FIRestarter variant of PostgreSQL implies that FIRestarter switches to STM more often, matching the limited performance gain in Figure 7.

E. Memory overhead

Figure 9 shows the mean memory overhead (RSS) of FIRestarter for each of the servers. We observed similar increase in binary sizes indicating that the overhead is mainly due to our instrumentation which includes code duplication and facilitating the adaptive switch. The undo log instrumentation for STM-only adds additional overhead [27] compared to an HTM-only solution.

F. Effectiveness against real-world bugs

Finally, we evaluate the real-world effectiveness of FIRestarter by reproducing several fatal bugs previously found to have affected servers in production environments.

Nginx (version 1.11.0) had a NULL pointer dereferencing bug in its `ngx_http_ssi_get_variable()` function [41]. Nginx worker process would crash when an incoming request contained a subrequest that requires Server Side Includes (SSI) substitution. We instrument the same version of Nginx with FIRestarter and subject it to the same conditions to trigger the crash. The instrumented server runs the crash-prone code within a transaction. This transaction begins after the last executed library call, `pread()`. Upon hitting the crash, our `SIGSEGV` handler initiates crash recovery. The library call, `pread()`, needs no compensation action, and the fault injector makes the library call return -1 and sets `errno` to `EINVAL`. This error percolates and the Nginx server eventually returns an empty response, thus effectively surviving the crash by turning it into an error.

On Lighttpd version 1.4.44, a WebDAV [42] request mixed with other incoming requests on the same keep-alive connection triggers a crash [43]. Due to a missing cleanup operation in the `mod_webdav_connection_reset()` function, the server can access an already freed memory pointer, leading to an immediate crash. We subject a FIRestarter-instrumented Lighttpd server to the same conditions to trigger the crash. However, our `SIGSEGV` handler initiates crash recovery in the last transaction, which in this case, begins after an `open64()` library call. After checkpoint rollback, its compensation action closes the opened file descriptor and sets the library call's return value to -1. The error value percolates and instead of crashing, the server returns an HTTP error response of "403 - Forbidden". Thus, it survives the crash to continue responding to other requests, showing the effectiveness of FIRestarter against fatal bugs encountered in production.

VII. Limitations and Future Work

FIRestarter repurposes existing error handling code to recover from crashes. We describe its limitations below.

Faults in error handlers and critical paths: Faulty error handling code hinders successful crash recovery. If a crash occurs while the application is executing error handling code, there will typically not be an error handler for the error handler. We do not expect FIRestarter to provide crash recoverability in such scenarios, and accept that crash recovery can only be as good as the application's error handling capabilities. Similarly, faults in critical paths of an application obstruct meaningful execution progress (e.g., in the event processing loop of an event driven server). Even if FIRestarter successfully disables the crash-affected region, availability of the application suffers since it would repeatedly try to execute the disabled region. We expect pre-deployment testing efforts to eliminate the bugs in critical paths, as these paths should be covered by all test cases.

Bugs in shared libraries: FIRestarter does not protect from bugs in shared libraries, although we can overcome this limitation by statically linking the libraries. It should be noted that doing so would change the locations where faults can be injected from the original library calls to the library calls issues by the now-embedded shared library. In theory, this may be either beneficial or detrimental. In practice, we expect this will be beneficial for the common case of custom shared libraries that are part of the application. Such libraries would typically forward environmental errors to the caller, leaving the recoverable surface intact while allowing for smaller transactions.

Shared memory interactions: Applications may use shared memory for inter-process communication. Such operations have externally visible consequences. Although we did not run into this in the evaluated set of applications (except PostgreSQL, where such interactions are irrecoverable), we must exclude regions within shared memory areas from transactions by instrumenting library calls that setup and manage shared memory.

Multithreading: Our use of HTM and STM limits concurrent executions. Since we repurposed Intel TSX for HTM, allowing concurrent executions within the transactions can result in additional aborts due to conflicting concurrent writes. Even if we extend STM to support concurrency, the restored values in memory may not remain consistent for re-execution because of non-determinism in thread scheduling. We leave this for future work, to explore supporting deterministic thread scheduling and restricting concurrency at library calls by some form of batching. However, many servers also provide

multi-process configurations (for example, Apache) where this limitation would not apply.

Despite these limitations in the general case, our experiments show that for practical server software (our main target), our approach is effective and efficient.

VIII. Related work

Restart-based recovery mechanisms provide protection against crashes caused by transient faults. As an example, in microreboots [16], Candea et al. propose to build applications out of small, loosely coupled components [44]—each of which can be recovered, by separating application logic from data, by means of a restart. In contrast, FIRestarter recovers from crashes due to both transient and persistent faults. Moreover, it recovers at a finer granularity (environment interaction intervals rather than full-fledged components), resulting in less disruption to availability.

Since operating system drivers and extensions are major sources of bugs with a large impact [45], [46], solutions such as Nooks [46] and shadow drivers [17] provide execution monitoring capabilities to replace faulty drivers and allow the system to remain available. Similarly, multiserver operating systems offer resilience against transient faults in (mostly stateless) drivers [47], [48] and core operating systems functions [49], often through component restart. More recently, researchers have extended (limited) recovery procedures to stateful interaction across multiple components and (some) non-transient faults through checkpointing [23]. However, these techniques are specific to particular types of software components, and do not generalize to event-driven server software.

A related reliability technique, software rejuvenation [50] also uses restarts, but this time to proactively restart target applications to avoid failures due to resource leaks and deadlocks. In contrast, FIRestarter *recovers* from crashes.

HAFT [22] partitions applications using instrumentation to detect transient hardware faults (e.g., due to cosmic rays). HAFT also uses hardware transactions for fault tolerance, but falls back to unprotected execution whenever the hardware runs out of resources to record execution state changes. In contrast, FIRestarter switches to software checkpointing, increasing the recoverable surface even for transient faults. Indeed, we show that hybrid transactions that have been used for concurrency in the past [51], are also a powerful technique for crash recovery.

To address faults from external sources, other researchers also developed compiler-assisted checkpoint-restart techniques, for instance, to allow continued program execution in low-powered energy-harvesting devices [52]–[54]. Specifically, they handle the transient faults resulting from frequent power disruptions by restoring the state to the last checkpoint when power gets restored, allowing continued application execution in the face of hardware (power) faults. FIRestarter, on the other hand, minimizes downtime due to fatal persistent *software* faults in the application code.

Fault injection [39], [55]–[59] has been a well-studied dependability technique to improve software testing, by automatically injecting various kinds of faults to ascertain robustness of software amidst potential fault conditions. FIRestarter repurposes fault injection to instead facilitate crash recovery from previously undiscovered faults in deployed software.

For crash recovery of persistent faults [19]–[21], restarting applications or components does not suffice, since re-execution may simply hit the same fault again, leading to an infinite repetition of crash recovery. Rx [19] therefore applies checkpoint-restart methods to recover an application from a crash and then modifies the application environment parameters such as memory availability and allocation patterns, in an attempt to divert re-execution away from fault-inducing code paths. It may take several rollbacks (potentially to progressively older checkpoints), while trying out several environment changes, for recovery to be successful. Instead, we take a direct approach by injecting environment interaction failures to force the execution towards an error handling path. ASSURE [20] and REASSURE [21] introduce binary instrumentation to perform error virtualization to divert application execution from crash-prone code towards locations in the application's code marked as *rescue points*. Unlike FIRestarter, they require offline analysis or manual annotations to map failure regions to suitable rescue points. Moreover, since they use a record-replay infrastructure based on execution logs to support offline analysis, the runtime overheads are significantly higher (up to 115%) [20]. We take a different approach toward error virtualization, where we instead utilize known environment interaction semantics for execution path diversion, eliminating the need for runtime profiling or manual analysis of application behavior. Moreover, runtime state checkpointing in FIRestarter is lightweight and limited to active crash transactions only.

IX. Conclusion

We presented FIRestarter, which demonstrated novel environment-oriented crash recovery by utilizing the application's own error handling code for crash recovery. We showed that the choice of library call intervals to define the boundaries of our crash transactions opens up opportunities for performance optimizations by leveraging hardware support to speed up runtime tracking of memory operations.

Acknowledgements

We would like to thank our shepherd, Domenico Cotroneo, and the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE), by the Netherlands Organisation for Scientific Research through grants NWO 639.021.753 VENI "PantaRhei" and NWO 628.001.030 "TROPICS", and by the Office of Naval Research (ONR) under awards N00014-16-1-2261 and N00014-17-1-2788. This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

REFERENCES

[1] "Github is just like all of us: The week has just started but it needed 4 whole hours of downtime," https://www.theregister.com/2020/07/13/github_takes_some_downtime_availability/, 2020.

[2] "Wobbly wednesday whacks ibm as 30 cloud services take unplanned naps," https://www.theregister.com/2020/06/25/ibm_multiple_cloud_outages/, 2020.

[3] "Aws's s3 outage was so bad amazon couldn't get into its own dashboard to warn the world," https://www.theregister.com/2017/03/01/aws_s3_outage, 2017.

[4] B. Littlewood and L. Strigini, "The risks of software," *Scientific American*, vol. 267, no. 5, pp. 62–75, 1992.

[5] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 291–301.

[6] 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 *USENIX Security Symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[7] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[8] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[9] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[10] "National vulnerability database," https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&search_type=last3years.

[11] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, "Why does the cloud stop computing? lessons from hundreds of service outages," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 1–16.

[12] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You, "Crashtuner: detecting crash-recovery bugs in cloud systems via meta-info analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 114–130.

[13] İ. E. Akkuş and A. Goel, "Data recovery for web applications," in *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2010, pp. 81–90.

[14] A. Verma, K. Voruganti, R. Routray, and R. Jain, "Sweeper: an efficient disaster recovery point identification mechanism," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008, pp. 1–16.

[15] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks: An architecture for reliable device drivers," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 2002, pp. 102–107.

[16] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot: A technique for cheap recovery. in 2004 usenix," in *ACM Symposium on Operating Systems Design & Implementation (OSDI 2004), San Francisco, CA, USA*, 2004.

[17] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 4, pp. 333–360, 2006.

[18] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: An organizing principle for recoverable operating systems," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, p. 49–60. [Online]. Available: https://doi.org/10.1145/1508244.1508251

[19] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," in *Acm sigops operating systems review*, vol. 39, no. 5. ACM, 2005, pp. 235–248.

[20] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "Assure: Automatic software self-healing using rescue points," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, p. 37–48. [Online]. Available: https://doi.org/10.1145/1508244.1508250

[21] G. Portokalidis and A. D. Keromytis, "Reassure: A self-contained mechanism for healing software using rescue points," in *International Workshop on Security*. Springer, 2011, pp. 16–32.

[22] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "Haft: hardware-assisted fault tolerance," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 25.

[23] K. Bhat, D. Vogt, E. van der Kouwe, B. Gras, L. Sambuc, A. S. Tanenbaum, H. Bos, and C. Giuffrida, "OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems," in *DSN*, Jun. 2016.

[24] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 87–102.

[25] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 539–554.

[26] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[27] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Lightweight memory checkpointing," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 474–484.

[28] D. Vogt, A. Miraglia, G. Portokalidis, A. S. Tanenbaum, H. Bos, and C. Giuffrida, "Speculative Memory Checkpointing," in *Middleware*, 2015. [Online]. Available: http://www.cs.vu.nl/~giuffrida/papers/middleware-2015.pdf

[29] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum, "Techniques for Efficient In-Memory Checkpointing," in *HotDep*, 2013. [Online]. Available: http://www.cs.vu.nl/~giuffrida/papers/hotdep-2013.pdf

[30] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 401–410.

[31] W. Zhang, M. De Kruijf, A. Li, S. Lu, and K. Sankaralingam, "Conair: featherweight concurrency bug recovery via single-threaded idempotent execution," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 113–126.

[32] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 2006, pp. 83–92.

[33] P. D. Marinescu and G. Candea, "Lfi: A practical and general library-level fault injector," in *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009, pp. 379–388.

[34] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–14.

[35] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 815–832.

[36] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 405–419.

[37] M. Bozyigit and M. Wasiq, "User-level process checkpoint and restore for migration," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 2, pp. 86–96, 2001.

[38] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004.

[39] E. Van Der Kouwe and A. S. Tanenbaum, "Hsfi: Accurate fault injection scalable to large code bases," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 144–155.

[40] W. Glozer, https://github.com/wg/wrk.

[41] Nginx.org, https://trac.nginx.org/nginx/ticket/1263.

[42] Lighttpd.net, https://redmine.lighttpd.net/projects/lighttpd/wiki/Docs_ModWebDAV/27.

[43] ——, https://redmine.lighttpd.net/issues/2780?tab=notes.

[44] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda, "Reducing recovery time in a small recursively restartable system," 02 2002, pp. 605– 614.

[45] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 73–88.

[46] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 207–222.

[47] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Failure Resilience for Device Drivers," in *DSN*, Oct. 2007. [Online]. Available: http://cs.vu.nl/~ast/Publications/Papers/dsn-2007.pdf

[48] ——, "Fault Isolation for Device Drivers," in *DSN*, Oct. 2009.

[49] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum, "Keep Net Working - On a Dependable and Fast Networking Stack," in *DSN*, Oct. 2012. [Online]. Available: http://www.few.vu.nl/~herbertb/papers/dsn2012.pdf

[50] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*. IEEE, 1995, pp. 381–390.

[51] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer, "Optimizing hybrid transactional memory: The importance of nonspeculative operations," in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 5364. [Online]. Available: https://doi.org/10.1145/1989493.1989501

[52] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 17–32.

[53] K. Maeng, A. Colin, and B. Lucia, "Alpaca: intermittent execution without checkpoints," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 96, 2017.

[54] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 129–144.

[55] E. van der Kouwe, "Improving software fault injection," Ph.D. dissertation, Ph. D. Thesis, Vrije Universiteit Amsterdam, 2016. https://www. cs. vu. nl . . . , 2016.

[56] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Edfi: A dependable fault injection tool for dependability benchmarking experiments," in *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*. IEEE, 2013, pp. 31–40.

[57] P. Marinescu, R. Banabic, and G. Candea, "An extensible technique for high-precision testing of recovery code," in *Proceedings of the USENIX annual technical conference*, no. CONF, 2010.

[58] P. D. Marinescu and G. Candea, "Lfi: A practical and general library-level fault injector," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 379–388.

[59] A. Johansson, N. Suri, and B. Murphy, "On the impact of injection triggers for os robustness evaluation," in *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)*. IEEE, 2007, pp. 127–126.