Replay without Recording of Production Bugs for Service Oriented Applications

Nipun Arora
Dropbox
New York, NY, USA
nipun@dropbox.com

Jonathan Bell
George Mason University
Fairfax, VA, USA
bellj@gmu.edu

Franjo Ivančić
Google
New York, NY, USA
ivancic@google.com

Gail Kaiser
Columbia University
New York, NY, USA
kaiser@cs.columbia.edu

Baishakhi Ray
Columbia University
New York, NY, USA
rayb@cs.columbia.edu

ABSTRACT

Short time-to-localize and time-to-fix for production bugs is extremely important for any 24x7 service-oriented application (SOA). Debugging buggy behavior in deployed applications is hard, as it requires careful reproduction of a similar environment and workload. Prior approaches for automatically reproducing production failures do not scale to large SOA systems. Our key insight is that for many failures in SOA systems (e.g., many semantic and performance bugs), a failure can automatically be reproduced solely by relaying network packets to replicas of suspect services, an insight that we validated through a manual study of 16 real bugs across five different systems. This paper presents *Parikshan*, an application monitoring framework that leverages user-space virtualization and network proxy technologies to provide a sandbox "debug" environment. In this "debug" environment, developers are free to attach debuggers and analysis tools without impacting performance or correctness of the production environment. In comparison to existing monitoring solutions that can slow down production applications, *Parikshan* allows application monitoring at significantly lower overhead.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Fault reproduction, live debugging

ACM Reference Format:
Nipun Arora, Jonathan Bell, Franjo Ivančić, Gail Kaiser and Baishakhi Ray. 2018. Replay without Recording of Production Bugs for Service Oriented Applications. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238186

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238186

1 INTRODUCTION

Modern-day devices rely on interactive and responsive applications that provide a rich interface to end-users. Behind the scenes of these applications are often several service-oriented applications working in concert to provide the final service. Such services include storage, compute, queuing, synchronization, and application-layer functionality. Applications following such service-oriented architectures (SOA) require the orchestration of a variety of components. Rapid resolution of incident (error/alert) management [58] in SOA [17, 20, 55, 69] is extremely important, as failure of one service can lead to cascading failure of the whole system. The large scale of such systems means that any downtime has significant impact on the "user experience, a product's image, and a company's brand and, potentially, revenue" [22].

Debugging production bugs in SOA is notoriously challenging because (1) it requires careful reproduction of a similarly orchestrated environment and workload so that developers can identify the root cause, and (2) bugs need to be resolved ASAP to ensure minimum downtime. Worse still, a single observed failure in one component might in fact be due to several latent bugs in other components. Thus, localizing a single bug might require understanding complex interactions across multiple components running on different hosts. Debugging becomes even more frustrating for *non-crashing* bugs, such as performance bugs, semantic bugs, and resource leaks, which tend to arise due to accumulated state, making them particularly complicated to reproduce in testing environments.

Most debugging techniques are not suitable for on-the-fly debugging of SOA production bugs. For instance, approaches like record-and-replay (R&R) typically collect execution trace information from the production environment and use that trace to reproduce the bug in a testing environment where developers can use traditional debugging tools. While R&R can be very effective in reproducing a bug, if sufficient execution traces can be captured to allow the entire application execution to be faithfully reproduced in a debugging environment [5, 27, 53], this can cause significant performance overhead. Despite much work towards optimizing the trace data captured, overheads imposed by such tracing can still be unacceptable for SOA production use: the overhead can balloon up to 2-10x overhead [71, 86].

In contrast, some monitoring systems capture only very minimal, high level information, for instance, collecting existing log

information and from it building a model of the system and its irregularities [10, 29, 32, 48]. While these systems impose almost no overhead on the production system being debugged (since they simply collect log information already being collected, or have light-weight monitoring), they can not automatically reproduce all bugs, and hence may be limited in their utility.

Thus, for on-the-fly debugging of SOA production bugs, we require a solution which allows developers to observe, instrument, and debug the system components in parallel with the production. In this paper, we propose a live debugging environment for SOA, which allows debuggers a free reign to debug, without impacting the user-facing application. By manually studying 220 real world SOA production bugs, we observe that it is possible to successfully replay a SOA bug *solely by capturing network transmissions*. For these bugs, conventional R&R capture of very low-level sources of non-determinism (e.g., thread scheduling, general system calls) is unnecessary to automatically reproduce the buggy execution.

Guided by this insight, we have developed *Parikshan*[1], a "live debugging" architecture that supports online debugging of production SOA applications without degrading access to the app during debugging. Our approach leverages technologies commonly used in SOA systems, such as lightweight containers, to automatically create sandboxed debugging environments that mirror their production environments. Each replica is kept isolated so developers can modify it without fear of impacting the production system. *Parikshan* replicates all network inputs flowing to the corresponding production container, buffering and feeding them (without blocking) to the debugging container. Within the debug environment, developers are free to use heavyweight instrumentation, that would not be suitable in a production environment, to diagnose the fault. This approach could be used offline, recording rather than replicating traffic and storing the cloned replicas for later, but *Parikshan* focuses on helping developers debug faults *online* — as they occur in production systems. The key benefits of *Parikshan* are:

Very low overhead in production: *Parikshan does* impose a short pause when debug environments are launched, but then developers are free to use very high overhead debugging tools (e.g. gdb) in the debug environment, yet the production environment continues to service requests at near-native speed.

Captures large-scale context: *Parikshan* captures the context of large scale, long running production systems by cloning services *in situ*, creating sandbox environments. Capturing such state is extremely difficult in conventional testing environments as they would need long-running test input sequences and large test-clusters.

We evaluate *Parikshan* by successfully replaying 16 real-world bugs, finding that *Parikshan* imposed very low overhead. Manually reproducing real-world bugs is a very time-intensive process, and hence, to lend additional validity to this evaluation of 16 bugs, we categorized 220 additional bugs from three applications, finding that most were similar in nature to the 16 that we reproduced.

2 MOTIVATION

To better understand what kinds of bugs occur in production SOA systems and how they can best be debugged, we studied 220 real-world production bugs from three SOA applications: Apache, MySQL and HDFS. We searched issue trackers for each project, ignoring bugs in non-production components. We also filtered bug reports that were feature requests or did not include a triggering test—our goal was to focus only on bugs that arose during production scenarios. To understand the nature of these bugs, we classified them (e.g. based on description and fix) into the following categories: Performance, Semantic, Concurrency, and Resource Leak, as shown in Table 1. Complete details on how we selected and categorized these bugs (along with a listing of the bugs themselves) are available in the full version of this work [7].

Table 1: Survey and classification of bugs

Category	Apache	MySQL	HDFS	Total
Performance	3	10	6	19
Semantic	37	73	63	173
Concurrency	3	7	6	16
Resource Leak	5	6	1	12
Total	48	96	76	220

One of the key insights from this study is that most of the bugs we examined (93%) are deterministic in nature (everything but concurrency bugs), and in fact, most are semantic bugs (80%). For many of them, the application behavior is incorrect (e.g. it provides the wrong output to the user), but there is no error or warning generated in the system log(s). To trigger these bugs, we only need to capture the state of the system and the input that results in the bug, and *not* all non-deterministic events (e.g. thread scheduling). We capture the state of the system through live cloning, replicating the entire state of each production container that is relevant to the bug. To capture the inputs to the system that result in the bug, we replicate all network inputs that enter the production containers.

2.1 Sample Scenario

Consider the complex multi-tier service-oriented system shown in Figure 1, which contains several interacting services each running in its own container (segregating components in separate containers is generally considered a best practice [4]). Operators might observe unusual memory usage in the Glassfish application server, causing error logs to be generated in the Nginx web server. Operators surmise there is a potential memory leak/allocation problem. However, with monitoring restricted to avoid performance

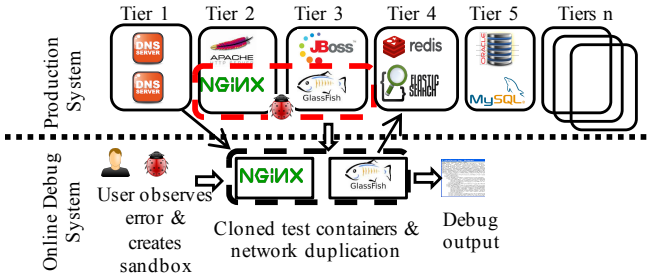

Figure 1: Workflow of *Parikshan* in a multi-tier system with interacting services. When the administrator observes errors in two of the tiers, she can create a sandboxed debug environment.

[1] *Parikshan* is the *Sanskrit* word for testing.

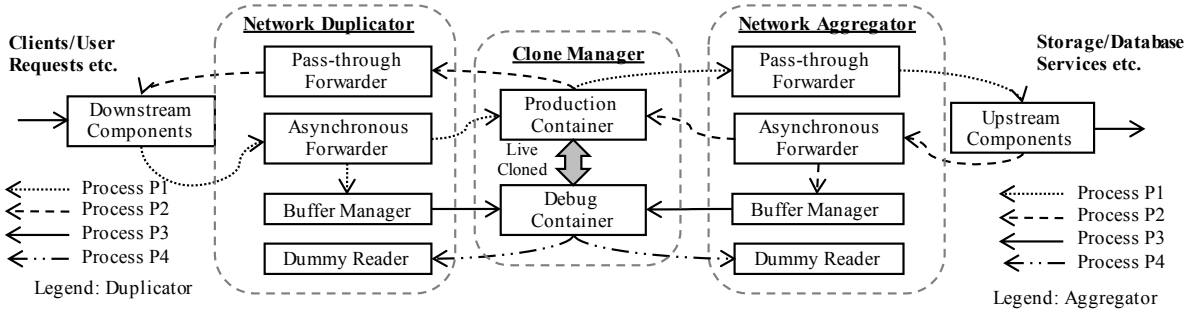

Figure 2: High level architecture of *Parikshan*, showing the main components: Network Duplicator, Network Aggregator, and Cloning Manager. The replica (debug container) is kept in sync with the master (production container) through network-level record and replay. In our evaluation, we found that this light-weight procedure was sufficient to reproduce many real bugs.

penalties on production, they can only go so far. Extensive trace collection in the production environment for reliable R&R debugging is also not feasible as that will hurt the system's performance. Thus, trouble tickets are typically generated for such problems, to be debugged offline. However, reproducing similar scenario offline involving so many SOA applications is challenging.

We observe that it is possible to replay the bug faithfully (without hurting the performance of the system), by simply cloning the potentially buggy containers and then sending the same network inputs as the production containers to these replicas. We design a fault reproduction framework, *Parikshan*, based on this observation.

Based on the erroneous behavior, the administrators can choose the Nginx and Glassfish containers for cloning and debugging, asking *Parikshan* to create the new *Nginx-debug* and *Glassfish-debug* containers. *Parikshan*'s network duplication mechanism ensures that the debug replicas receive the same inputs as the production containers and that the production containers continue service without further interruption. Once the debug environment is created, *Parikshan* can be used with any existing automated or manual debugging tools that developers may wish to use. This separation of production and debugging environment allows the developers to use heavier dynamic instrumentation for deeper diagnosis in the debug containers without fear of disrupting production. Since each replica is cloned from its original "buggy" *production container*, it exhibits the same persistent memory leaks and/or logical errors. Note that we primarily envision *Parikshan* being applied to reproduce application bugs, and not to reproduce security attacks.

Debug containers can be created and recreated at any time: either at the start of execution or at any point during execution, allowing post-facto analysis of the bugs. Within debug replicas, analysis tools that slow down the buggy execution may be used without impacting production performance. Hence, developers can use any of their preferred debugging approaches in these replicas in order to determine the cause of the failure.

3 DESIGN

Since *Parikshan* is built for on-the-fly debugging of SOA production bugs, its design is guided by the following principles.

(1) *Real-Time Insights:* Observing application behavior as a bug presents itself will allow for quick insights and shorter time to debug. Developers should be able to monitor system status as they debug.
(2) *Sanity and Correctness*: If debugging is to be done in a running application with real users, it should be done without impacting the outcome of the program. The framework must ensure that any changes to the application's state or to the environment does not impact the user-facing production application.
(3) *Language/Application Agnostic*: The mechanisms presented should be applicable to any language, and any service oriented application (our scope is limited to SOA architectures).
(4) *Performance Impact*: The end user of a system that is being debugged should not observe any noticeable performance degradation. Debugging must be unobtrusive to the end user, both in terms of functionality and any configuration or setup, in addition to performance.
(5) *Service Interruption*: Since we are focusing our efforts on service oriented systems, any solution should ensure that there is no impact on the service, and the user facing service should not be interrupted.

Figure 2 shows the architecture of *Parikshan* when applied to a single mid-tier application server. *Parikshan* consists of 3 modules: (1) **Clone Manager**: manages *live cloning* between the production containers and the debug replicas. Live cloning allows developers to decide to create new debug environments at anytime while an application is running. (2) **Network Duplicator**: manages network traffic duplication from downstream servers to both the production and debug containers. (3) **Network Aggregator**: manages network communication from the production and debug containers to upstream servers. The network duplicator also performs the important task of ensuring that the production and debug container executions do not diverge. The duplicator and aggregator can be used to target multiple connected tiers of a system by duplicating traffic at the beginning and end of a workflow. *Parikshan* can dynamically detect which ports an application uses and prompt the developer to choose if traffic on each port should be aggregated or duplicated.

3.1 Clone Manager

Parikshan uses live cloning (a variant of live migration [23, 37, 62]) to spawn debug containers that exactly mirror the corresponding production services without disconnecting any clients or stopping

any processes, incurring a negligible suspend time. The challenge here is to manage two containers with the same identities in the network and application domain. This is important as the operating system and the application processes running in it may be configured with IP addresses that cannot be changed on the fly. Hence, the same network identifier should map to two separate addresses, and enable communication with no problems or slowdowns.

Parikshan supports two high-level modes of live cloning:

Internal Cloning: In this mode, we allocate the production and debug containers to the same physical host. This mode takes less time to perform the initial clone (since the container does not need to be transferred over the network), and may be more cost-effective since it does not require additional machines. However, co-hosting the debug and production containers could potentially decrease performance of the production container due to resource contention. Network identities in this mode are managed by encapsulating each container in separate network namespaces [2]. This allows both containers to have the same IP address with different interfaces. The duplicator is then able to communicate to both these containers with no networking conflict.

External Cloning: In this mode, we provision an external server as the host of our debug container (this server can host more than one debug container). While this mechanism can have a higher overhead in terms of suspend time and requires provisioning an additional host, the advantage of this mechanism is that once cloned, the debug container is totally separate and will not impact the performance of the production container. Network identities in external mode are managed using NAT (network address translation) in both host machines. Hence both containers can have the same address without any conflict. Currently, we assume that each production container has at most a single debug replica, in the future we will consider supporting multiple replicas for each production container, which might make it easier for developers to apply multiple debugging techniques simultaneously.

The suspend time of cloning depends on the operations happening between step 2 and step 4 (the first and the second rsync): more operations will result in more modified pages of memory, impacting the amount of memory that needs to be later copied. This suspend time can be viewed as an amortized cost in lieu of instrumentation overhead. We evaluate the performance of live cloning in §4.1.

3.2 Network Duplicator and Aggregator

Once the debug container is provisioned, *Parikshan* keeps the debug container in sync with the production container by duplicating network traffic into the container. The network proxy duplicator and aggregator are composed of the following internal components:

- **Synchronous Passthrough**: The synchronous passthrough takes input from a source port and forwards it to a destination port. The passthrough is used for communication from the production container out to other components (which are not duplicated).
- **Asynchronous Forwarder**: The asynchronous forwarder takes input from a source port and forwards it to both a destination port and to an internal buffer. Forwarding to the buffer is done in a non-blocking manner, so as to not delay network forwarding.
- **Buffer Manager**: Manages a FIFO queue for data kept internally in the proxy for the debug container. It records the incoming data, and forwards it to a destination port.
- **Dummy Reader**: This is a standalone daemon that reads and drops packets from a source port.

Proxy Network Duplicator: All requests inbound to the production container are duplicated and forwarded to the debug container. A simple network proxy or port mirror would duplicate all traffic from the production container to the debug container but would not be able to cope with the different execution speeds of the two containers, and would not be able to correctly filter *responses* from the debug container back to the client (which should only receive responses from the production container).

Our solution is a customized TCP level proxy. This proxy duplicates network traffic to the debug container while maintaining the TCP session and state with the production container. Since it works at the TCP/IP layer, applications are completely oblivious to it. Figure 2 shows how our proxy works: each incoming connection is forwarded to both the production container and the debug container. This is a multi-process job involving 4 parallel processes (P1-P4): In P1, the asynchronous forwarder sends data from client to the production service, while simultaneously sending it to the buffer manager in a non-blocking send. This ensures that there is no delay in the flow to the production container because of slow-down in the debug container. In P2, the pass-through forwarder reads data from the production and sends it to the client (downstream component). Process P3 then sends data from Buffer Manager to the debug container, and Process P4 uses a dummy reader to read from the production container and drops all the packets.

The above strategy allows for non-blocking packet forwarding and enables a key feature of *Parikshan*, whereby it avoids slowdowns in the debug container to impact the production container using an in-memory buffer (discussed further in §3.3).

Proxy Network Aggregator: While the network duplicator duplicates incoming requests, the network aggregator manages incoming "responses" for requests sent from the debug container. In addition to dropping duplicate responses to clients, the network aggregator must also drop duplicate requests to backend servers. For instance, processing a request in a mid-tier server might require inserting or deleting data from a backend database: since both the production and debug containers will process this request, there will be duplicate requests sent to these backend services, leading to an inconsistent state. The "proxy aggregator" module stubs the requests from a duplicate debug container by replaying the responses sent to the production container to the debug container and dropping all packets sent from it to upstream clients.

As shown in Figure 2, when an incoming request comes to the aggregator, it first checks if the connection is from the production container or debug container. In process P1, the aggregator forwards the packets to the upstream component using the pass-through forwarder. In P2, the asynchronous forwarder sends the responses from the upstream component to the production container, and sends the response in a non-blocking manner to the internal queue in the buffer manager. Once again this ensures no slow-down in the responses sent to the production container. The buffer manager then forwards the responses to the debug container (Process P3). Finally, in process P4 a dummy reader reads all the responses from the debug container and discards them.

We assume that the production and the debug container are in the same state, and are sending the same requests. Hence, sending the corresponding responses from the FIFO queue instead of the backend ensures: (a) all communications to and from the debug container are isolated from the rest of the network, (b) the debug container gets a logical response for all it's outgoing requests, making forward progress possible, and (c). similar to the proxy duplicator, the communications from the proxy to internal buffer is non-blocking to ensure no overhead on the production container.

3.3 Debug Window

Parikshan's asynchronous forwarder uses an internal buffer to ensure that incoming requests proceed directly to the production container without any latency, regardless of the speed at which the debug replica processes requests. The incoming request rate to the buffer is dependent on the user, and is limited by how fast the production container manages the requests (i.e. the production container is the rate-limiter). The outgoing rate from the buffer is dependent on how fast the debug container processes the requests. As instrumentation overhead increases, the incoming rate of requests may eventually exceed the transaction processing rate in the debug container, leading to a buffer overflow. We call the time period until buffer overflow happens the *debug window*. Once the buffer has overflowed, the debug container may be out of sync with production, and a fresh debug container would need to be launched.

The debug window size depends on the size of the buffer, the incoming request rate, the overhead of any debugging activities and the application behavior, in particular how it launches TCP connections. *Parikshan* generates a pipe buffer for each TCP connect call, and the number of pipes are limited to the maximum number of connections allowed in the application. Hence, buffer overflows happen only if the requests being sent in the same connection overflow the queue. For webservers and application servers, the debugging window size is generally not a problem, as each request is a new "connection." This enables *Parikshan* to tolerate significant instrumentation overhead without a buffer overflow. On the other hand, database and other session based services usually have small request sizes, but multiple requests can be sent in one session which is initiated by a user. In such cases, for a server receiving a heavy workload, the number of calls in a single session may eventually have a cumulative effect and cause overflows.

To further increase the debug window, *Parikshan* could load balance debugging instrumentation overhead across multiple debug containers, each of which can get a duplicate copy of the incoming data. For instance, debug container 1 could have 50% of the instrumentation, and the rest could occur in debug container 2. Such a strategy would significantly reduce the chance of a buffer overflow in cases where heavy instrumentation is needed.

3.4 Divergence Checking

It is possible that non-deterministic behavior (discussed in §6) in the containers or instrumentation could cause the production and debug container to diverge over time. To understand and capture this divergence, we compare the corresponding network outputs received by the proxy, providing a black-box mechanism to check the fidelity of the replica based on its communication with external components. We use a hash of each data packet, which is collected and stored in memory for the duration that each packet's connection is active. The degree of acceptable divergence is dependent on the application behavior, and the operator's wishes. For example, an application that includes timestamps in each of its messages (i.e. is expected to have some non-determinism) could perhaps be expected to have a much higher degree of acceptable divergence than an application that should normally be returning deterministic results.

3.5 Implementation

Parikshan is publicly available under the MIT open source license on GitHub [6]. The clone manager and the live cloning utility are built on top of the user-space container virtualization software OpenVZ [51]. *Parikshan* extends *VZCTL* 4.8 [35] live migration facility [62], to provide support for online cloning. The network isolation for the production container was done using Linux network namespaces [2] and NAT. While *Parikshan* is based on lightweight containers, we believe it can also be applied to traditional virtualization software where live migration has been further optimized [26, 81].

The network proxy duplicator and the network aggregator are implemented in C/C++. The forwarding in the proxy is done by forking off multiple processes each handling one send/or receive a connection in a loop from a source port to a destination port. Data from processes handling communication with the production container, is transferred to those handling communication with the debug containers using *Linux Pipes* [1]. Pipe buffer size is a configurable input based on user specifications.

4 EVALUATION

To evaluated *Parikshan* through the following research questions:
RQ1: How long does it take to create a live clone of a production container and what is its impact on the performance of the production container?
RQ2: What is the impact of *Parikshan* on the throughput and latency of the production application?
RQ3: What is the size of the debugging window, and how does it depend on resource constraints?
RQ4: Can *Parikshan* successfully reproduce real bugs?

We compared *Parikshan*'s two cloning modes (internal and external). Our internal cloning mode was evaluated using two identical VM's with an Intel i7 CPU, with 4 Cores, and 16GB RAM each in the same physical host (one each for production and debug containers). We evaluated the external cloning mode on two identical host nodes with Intel Core 2 Duo Processor, 8GB of RAM. All evaluations were performed on CentOS 6.5. Apart from cloning performance evaluation in RQ1, other evaluations use external mode (i.e. different identical machines for debug and production containers).

4.1 RQ1: Live Cloning Performance

As explained in §3, a short suspend time during live cloning is necessary to ensure that both containers are in the exact same system state. We measure this overhead on both real and synthetic workloads, and separate the suspend time into its four components: (1) Suspend & Dump: time taken to pause and dump the container, (2) Pcopy after suspend: time required to complete rsync operation, (3) Copy Dump File: time taken to copy an initial dump file, and (4)

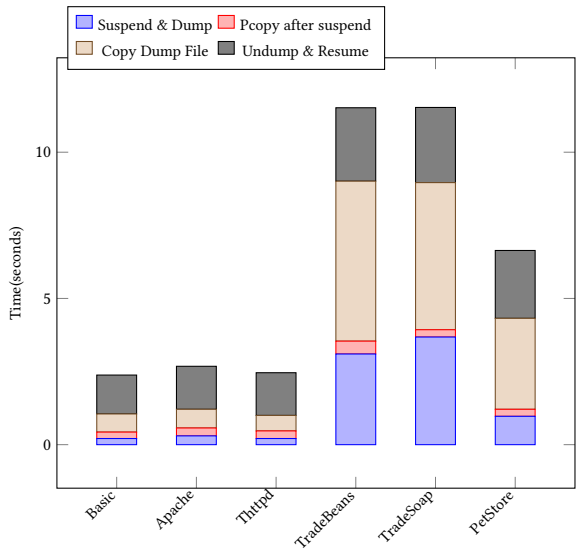

Figure 3: Suspend time for live cloning, when running a representative benchmark. Here 'basic' indicates baseline without any services running in the container.

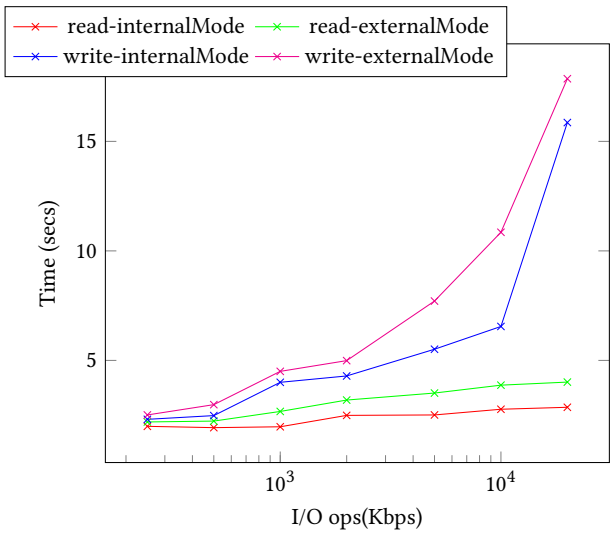

Figure 4: Live Cloning suspend time with increasing amounts of I/O operations

Undump & Resume: time taken to resume the containers. We used both micro and macro benchmarks to evaluate live cloning.

Real-world applications and workloads: First, we evaluated *Parikshan*'s suspend time using five well-known applications workloads. We ran the httperf [63] benchmark on Apache and *thttpd* to compute max throughput of the web-servers, by sending a large number of concurrent requests. Tradebeans and Tradesoap are realistic workloads running a multi-tier stock trading application and are part of the DaCapo [16] benchmark "DayTrader" application. PetStore [3] is also a well known JEE reference application. We deployed PetStore in a 3-tier system with JBoss, MySQL and Apache servers, and cloned the app-server. The input workload was a random set of transactions which were repeated for the duration of the cloning process.

As shown in Figure 3, for Apache and Thttpd the container suspend time ranged between 2-3 seconds. However, in more memory intensive application servers such as PetStore and DayTrader, the total suspend time was higher (6-12 seconds). Nevertheless, we did not experience any timeouts or errors for the requests in the workload[2]. We felt that these relatively fast temporary app suspensions were a reasonable price to pay to launch an otherwise overhead-free debug replica.

Microbenchmark: The main factor that impacts suspend time is the number of "dirty pages" (recently modified) in the suspend phase that have not been copied over in the pre-copy rsync operation (see § 3.1). Hence, to further characterize the suspend time imposed by the cloning phase of *Parikshan*, we created a microbenchmark that controls this variable. We used the *fio* utility [9] to gradually increase the number of I/O operations while doing live cloning. We ran *fio* to read and writes of random values with a controlled I/O bandwidth. We ensured that the I/O workload being processed by *fio* was long enough to last through the cloning process.

As shown in Figure 4, read operations have a much smaller impact on suspend time of live cloning compared to write operations. This can be attributed to the increase of dirty pages in write operations, whereas for read, the disk image remains largely the same. The internal mode is much faster than the external mode, as both the production and debug container are hosted in the same physical device. For higher I/O operations, with a large amount of dirty pages, network bandwidth becomes a bottleneck: leading to longer suspend times. Overall in our experiments, the internal mode is able to manage write operation up to 10 Mbps, with a total suspend-time of approx 5 seconds, whereas, the external mode is only able to manage up to 5-6 Mbps, for a 5 sec suspend time.

> To answer **RQ1**, live cloning introduces a short suspend time in the production container dependent on the workload. Write intensive workloads will lead to longer suspend times, while read intensive workloads will take much less. Suspend times in real workload on real-world systems vary from 2-3 seconds for webserver workloads to 10-11 seconds for application/database server workloads. Compared to external mode, internal mode had a shorter suspend time. A production-quality implementation could reduce suspend time further by rate-limiting incoming requests in the proxy, or using copy-on-write mechanisms and faster shared file system/storage devices already available in several existing live migration solutions.

4.2 RQ2: Impact on Production Performance

We measured the impact of *Parikshan* on a running production application (after the debug environment was created) in terms of

[2]In case of packet drops, requests are resent both at the TCP layer, and the application layer. This slows down the requests for the user, but does not drop them

Table 2: First four latencies of GET/POST requests from wikipedia traces. The third and fourth column show overhead of proxy compared to native, and overhead of duplication compared to proxy mode

Native	Proxy	Duplication	Proxy Overhead	Duplication Overhead
0.29702	0.30696	0.30623	3.347	-0.2405
0.06117	0.06154	0.06250	6.08	1.55
0.05342	0.05676	0.05564	6.25	-1.97825
0.05424	0.05438	0.05437	0.261	-0.0168

throughput and latency. To understand the impact of duplication on network throughput, we ran a microbenchmark using *iperf* [82] in three different modes — native (just the client and server), proxy (client communicates to server through a proxy) and duplication (client communicates to original server, and a clone via *Parikshan*'s duplicator). We observed that while the native and proxy communication had no discernible difference with both transferring at 941Mb/s, the duplication mode was on an average 0.5% slower than the other two. We believe this difference is negligible for most practical applications and will not impact application end-to-end performance.

To measure the impact on latency, we created a scaled down version of Wikipedia (MediaWiki) [12] where we populated data from data dumps available through wikibench [84]. We used a sample workload of requests from 2008, and compared the latencies of about 500 HTTP requests in the same three deployments (native, proxy and duplicate). Table 2 shows a snapshot of 4 such requests and their latencies and overheads in different modes. We found that the proxy was generally slower than the native connection, with the slowdown ranging from 1-8%. More importantly we found that when comparing the latencies in the duplication mode to our proxy mode, the overhead was negligible (±2 % due to caching). These experiments are described in greater detail in the full version of this work [7].

> To answer **RQ2**, we found that duplication of traffic has minimal impact on throughput (0.5%), and no discernible impact on network latency.

4.3 RQ3: Debug Window Size

The network-level proxies are responsible for buffering communication to/from the debug container(s), allowing the production application to operate without slowing down to account for any overheads in the debug application. Hence, the size of this buffer directly impacts how far the debug environment is able to fall behind production. We refer to this time window (where debugging can call behind production) as the *debug window*, and evaluated how different size buffers impact the ability of developers to debug in both real-world experiments, and also in controlled simulations.

Experimental Results: To evaluate the approximate size of the debug window, we sent requests to both a production and debug MySQL container via our network duplicator. Each workload ran for about 7 minutes (10,000 "select * from table" queries), with varying request workloads. We also profiled the server, and found that is able to process a max of 27 req/s[3] in a single user connect session. For each of our experiments, we vary the buffer sizes to get an idea of debug window. We generated a slowdown by modeling the time taken by MySQL to process requests (27 req/s or 17req/s), and putting an approximate sleep in the request handler.

Table 3: Approximate debug window sizes for a MySQL request workload

Input Rate	Debug Window	Pipe Size	Slowdown
530 bps, 27 rq/s	∞	4096	1.8x
530 bps, 27 rq/s	8 sec	4096	3x
530 bps, 27 rq/s	72 sec	16384	3x
Pois., λ = 17 rq/s	16 sec	4096	8x
Pois., λ = 17 rq/s	18 sec	4096	5x
Pois.,λ = 17 rq/s	∞	65536	3.2x
Pois.,λ = 17 rq/s	376 sec	16384	3.2x

Initially, we created a connection and sent requests at the maximum request rate the server was able to handle (27 req/s). We found that for overheads up-to 1.8x (approx) we experienced no buffer overflows. For higher overheads the debug window decreased, primarily dependent on buffer size, request size, and slowdown.

Next, we mimic user behavior, to generate a realistic workload. We send packets using a Poisson process with an average request rate of 17 requests per second to our proxy. This varies the inter-request arrival time, and lets the debug container catch up with the production container during idle periods between request bursts. We observed that compared to earlier experiments there was more slack in the system, allowing it to tolerate a much higher overhead (3.2x) with no buffer overflows.

Simulation Results: In our next set of experiments, we simulate packet arrival and service processing for a buffered queue in SOA applications. We use a discrete event simulation based on an MM1 queue, which is a classic queuing model based on Kendall's notation [49], and is often used to model SOA applications with a single buffer based queue. Essentially, we are sending and processing requests based on a Poisson distribution with a finite buffer capacity. In our simulations (see Figure 5), we kept a constant buffer size of 64GB, and iteratively increased the overhead of instrumentation, thereby decreasing the service processing time. Each series (set of experiments), starts with an arrival rate approximately 5 times less than the service processing time. This means that at 400% overhead, the system would be running at full capacity (for stable systems SOA applications generally operate at much less than system capacity). Each simulation instance was run for 1,000,000 seconds (277.7 hours). We gradually increased the instrumentation by 10% each time, and observed the *hitting time* of the buffer (time it takes for the buffer to overflow for the first time). As shown, there is no buffer overflow in any of the simulations until the overhead reaches around 420-470%, beyond this the debug window decreases exponentially. Since beyond 400% overhead, the system is over-capacity, the queue will start filling up fairly quickly. This clarifies the behavior we observed in our experiments, where for lower overheads

[3]Not the same as bandwidth, 27 req/s is the maximum rate of sequential requests MySQL server is able to handle for a user session

(1.8-3.2x) we did not observe any overflow, but beyond a certain point, we observed that the buffer would overflow fairly quickly. Also as shown in the system, since the buffer size is significantly larger than the packet arrival rate, it takes some time for the buffer to overflow (several hours). We believe that while most systems will run significantly under capacity, large buffer sizes can ensure that our debug container may be able to handle short bursts in the workload. However, a system running continuously at capacity is unlikely to tolerate significant instrumentation overhead.

> To answer **RQ3**, we found that the debug container can stay in a stable state without any buffer overflows as long as the instrumentation does not cause the service times to become more than the request arrival rate. Furthermore, a large buffer will allow handling of short bursts in the workload until the system returns back to a stable state. The debug window can allow for a significant slowdown, which means that many existing dynamic analysis techniques [33, 68], as well as most fine-grained tracing [32, 48] can be applied on the debug container without leading to an incorrect state.

4.4 RQ4: Reproducing Real Bugs

One of our core insights is that for most SOA systems, production bugs can hence be triggered by network replay alone. To validate this insight, we selected sixteen real-world bugs, applied *Parikshan*, reproduced them in a production container, and observed whether they were also simultaneously reproduced in the replica. For each of the sixteen bugs that we triggered in the production environments, *Parikshan* faithfully reproduced them in the replica.

We selected our bugs from those examined in previous studies [59, 89], focusing on bugs that involved performance, resource-leaks, semantics, concurrency, and configuration. We have further categorized these bugs whether they lead to a crash or not, and if they can be deterministically reproduced. Table 4 presents an overview of the bugs that we studied. A thorough description of each bug, the steps that we took to reproduce it with *Parikshan*, and description of the debugging experience is available in the full version of this work [7] in section 4.3.

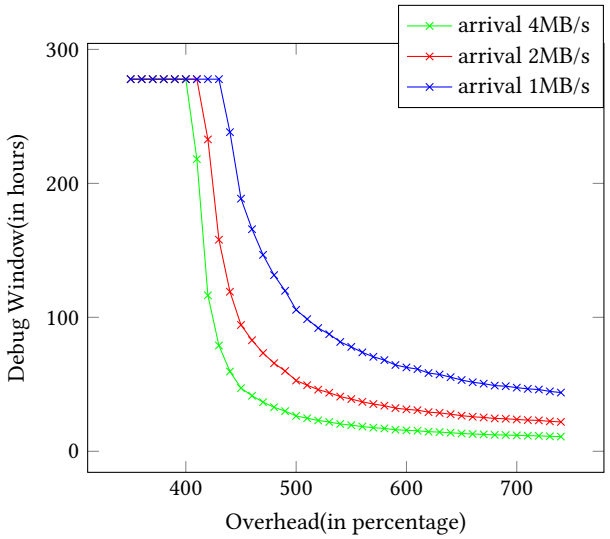

Figure 5: Simulation results for debug window size (buffer=64GB). Each series has a constant arrival rate.

Semantic Bugs: We recreated 4 semantic bugs from Redis [20] queuing system, and Cassandra [55] (a NoSQL database). For instance, Redis#761 is an integer overflow error. This error is triggered, when the client tries to insert and store a very large number. This leads to an unmanaged exception, which crashes the production system. Others such as Redis#487 resulted in expired keys still being retained in Redis, because of an unchecked edge condition. While this error does not lead to any exception or any error report in application logs, it gives the user a wrong output. In the case of such logical errors, the application keeps processing, but the internal state can stay incorrect. In our experiments, we were able to clone the input of the production in the debug containers and easily replayed both these errors.

Performance Bugs: We replayed 3 MySQL production bugs. For example, iMySQL#15811 reported that some of the user requests which were dealing with complex scripts (Chinese, Japanese), were running significantly slower than others. To evaluate *Parikshan*, we re-created a two-tier client-server setup with the server (container) running a buggy MySQL server and sent queries to the production container with complex scripts (Chinese). These queries were asynchronously replicated, in the debug container. To further investigate the bug-diagnosis process, we also turned on execution tracing in the debug container using SystemTap [30]. This gives us the added advantage, of being able to profile and identify the functions responsible for the slow-down, without the tracing having any impact on production.

Resource Leaks: *Parikshan* successfully reproduced 2 resource leak bugs in Redis. Let us take Redis#417 for instance, here we had a redis master and slave set up for both production and debug container. We then triggered the bug by running concurrent requests through the client which can trigger the memory leak. The memory leak was easily replayed in the debug container by turning on debug tracing, which showed a growing memory usage.

Concurrency Bugs: One of the most subtle bugs in production systems is caused due to concurrency errors. These bugs are hard to reproduce, as they are non-deterministic, and may or may not happen in a given execution. Although *Parikshan* cannot guarantee the replay of concurrency bugs, in our experiment we could successfully reproduce all the concurrency bugs showing in Table 4. Given that the debug container is a live-clone of the production container, and that it replicates the state of the production container entirely, we believe that the chances of replaying the non-deterministic concurrency bug in the debug container are quite high, as evident by our experiments. Additionally, the debug container is a useful tracing utility to track thread lock and unlock sequences, to get an idea of the concurrency bug.

Configuration Bugs: Configuration errors are usually caused by wrongly configured parameters, i.e., they are not bugs in the application, but bugs in the input (configuration). These bugs usually get triggered at scale or for certain edge cases, making them extremely difficult to catch. A simple example of such a bug is Redis#957, here the slave is unable to sync with the master. The connection with the slave times out and it's unable to sync because of the large data.

Table 4: List of real-world production bugs studied with *Parikshan*

Bug Type	Bug ID	Application	Symptom/Cause	Determ-inistic	Crash	Steps to Reproduce
Semantic	Redis #487	redis-2.6.14	Keys* command duplicate or omits keys	Yes	No	Set keys to expire, execute specific reqs
	Cassandra #5225	cassandra-1.5.2	Missing columns from wide row	Yes	No	Fetch columns from cassandra
	Cassandra #1837	cassandra-0.7.0	Deleted columns become available after flush	Yes	No	Insert, delete, and flush columns
	Redis #761	redis-2.6.0	Crash with large integer input	Yes	Yes	Query for input of large integer
Performance	MySQL #15811	mysql-5.0.15	Bug caused due to multiple calls in a loop	Yes	No	Repeated insert into table
	MySQL #26527	mysql-5.1.14	Load data is slow in a partitioned table	Yes	No	Create table with partition and load data
	MySQL #49491	mysql-5.1.38	calculation of hash values inefficient	Yes	No	MySql client select requests
Concurrency	Apache #25520	httpd-2.0.4	Per-child buffer management not thread safe	No	No	Continuous concurrent requests
	Apache #21287	httpd-2.0.48, php-4.4.1	Dangling pointer due to atomicity violation	No	Yes	Continuous concurrent request
	MySQL #644	mysql-4.1	data-race leading to crash	No	Yes	Concurrent select queries
	MySQL #169	mysql-3.23	Race condition leading to out-of-order logging	No	No	Delete and insert requests
	MySQL #791	mysql-4.0	Race - visible in logging	No	No	Concurrent flush log and insert requests
Resource Leak	Redis #614	redis-2.6.0	Master + slave, not replicated correctly	Yes	No	Setup replication, push and pop some elements
	Redis #417	redis-2.4.9	Memory leak in master	Yes	No	Concurrent key set requests
Configuration	Redis #957	redis-2.6.11	Slave cannot sync with master	Yes	No	Load a very large DB
	HDFS #1904	hdfs-0.23.0	Create a directory in wrong location	Yes	No	Create new directory

While the bug is partially a semantic bug, as it could potentially have checks and balances in the code. The root cause itself is a lower output buffer limit. Once again, it was easily replayed in our debug containers that the slave was not synced, and investigated further by the debugger.

To answer **RQ4**, we found that *Parikshan*'s approach of capturing the network traffic and replaying it in an offline environment is efficient to reproduce real production bugs.

5 APPLICATIONS OF LIVE DEBUGGING

Statistical Testing: One well-known technique for debugging production applications is statistical testing. This is achieved by having predicate profiles from both successful and failing runs of a program and applying statistical techniques to pinpoint the cause of the failure. The core advantage of statistical testing is that the sampling frequency of the instrumentation can be decreased to reduce the instrumentation overhead. However, the instrumentation frequency for such testing to be successful needs to be statistically significant. Unfortunately, overhead concerns in the production environment limit the frequency of instrumentation. In *Parikshan*, the buffer utilization can be used to control the frequency of such statistical instrumentation in the debug container. This would allow the user to utilize the slack available in the debug container for instrumentation to it's maximum, without leading to an overflow. Thereby improving the efficiency of statistical testing.

Record and Replay: Record and Replay techniques have been proposed to replay production site bugs. However, they are not yet used in practice as they can impose unacceptable overheads in the service processing time. *Parikshan* replicas can be used to do recording at a much finer granularity (higher overhead), allowing for easy and fast replays offline. Similar to existing mechanisms, the system can be replayed can then be used for offline debugging, without imposing any recording overhead to the production container.

Patch Testing: Bug fixes and patches to resolve errors, often need to undergo testing in the offline environment and are not guaranteed to perform correctly. Patches can be made to the replica instead. The fix can be traced and observed if it is correctly working, before moving it to the production container. This is similar in nature to AB-Tesing, which is applied to find if a new fix is useful or works [31].

6 LIMITATIONS AND THREATS TO VALIDITY

There may be several threats to the validity of our findings. For instance, the bugs that we selected to study may not be truly representative of a broad range of different faults. Perhaps *Parikshan*'s low-overhead network replay approach is less suitable to some classes of bugs. To alleviate this concern, we selected from several established bug categories, and further, evaluated *Parikshan* with bugs that had already been studied in other literature, to alleviate a risk of selection bias. We further strengthened this by categorizing 220 bug reports from three real-world applications, finding that most were semantic in nature, and very few were non-deterministic, with similar characteristics to the 16 that we demonstrated *Parikshan* can reproduce.

There are also several underlying limitations and assumptions regarding *Parikshan*'s applicability:

Non-determinism: Non-determinism can be attributed to three main sources (1) system configuration, (2) application input, and (3)

ordering in concurrent threads. Live cloning of the application state of a service container ensures that both the production and debug services are in the same "system-state" and have the same configuration parameters for itself and all dependencies. *Parikshan*'s network proxy ensures that all inputs received in the production container are also forwarded to the debug container. However, any non-determinism from other sources (e.g., thread interleaving, random numbers, reliance on timing) may limit *Parikshan*'s ability to faithfully reproduce an execution. While our current prototype does not handle these, we believe there are several existing techniques that can be applied to tackle this problem in the context of live debugging, such as deterministic scheduling [88]. *Parikshan* allows significant tracing of synchronization points, often required for constraint solvers [33, 36] to go explore all synchronization orderings to find concurrency errors. We have also tried to alleviate this problem using our divergence checker (Section 3.4). And, as was seen in our case-studies, even in the face of limited non-determinism bugs will often still be triggered in the replica.

Distributed Services: Large-scale distributed systems are often comprised of several interacting services such as storage, NTP, backup services, controllers and resource managers. *Parikshan* can be used for multiple communicating services, where any given service may be cloned, turned off or continue as is, depending on its nature. For example, storage services supporting a replica should be cloned or turned off (depending on debugging environment) as they could propagate changes from the debug container to the production containers. Services like NTP can be allowed to continue without cloning as their publishsubscribe broadcast cannot be impacted by cloning of other services anyway. Furthermore, instrumentation inserted in a replica will not necessarily slowdown all its services, e.g., adding instrumentation to a MySQL query handler will not slow down file-sharing or NTP services running in the same container.

Data Privacy: This is a limitation that *Parikshan* shares with most existing record-replay systems [19, 25]. *Parikshan* clones incoming traffic thereby any debugger having access to the debug machine can potentially look at the user data (depending on the kind of instrumentation they use). Currently we do not propose any way to address this issue in our system, and leave it to the debugger (and the data access control policies of their production deployments) as to how this can be addressed.

7 RELATED WORK

Record and Replay Systems: Record and Replay [8, 11, 13, 15, 18, 24, 28, 34, 38–41, 43–47, 52, 54, 56, 57, 60, 61, 65, 66, 70, 71, 73–79, 83, 85–87, 90] has been an active area of research in the academic community for many years. These systems offer highly faithful re-execution but incur performance overhead on the production application — for instance, ODR [5] reports 1.6x slowdown and rr [64] 1.2x; Scribe [53] reduces to 2.5% for server applications and 15% for desktop applications. *Parikshan* avoids recording overhead entirely, but its cloning suspend time may be viewed as an amortized cost in comparison to the overhead in record-replay systems.

Among record and replay systems, the work we know of most closely related to ours is Aftersight [21]. Aftersight records a production system and replays it concurrently alongside in another VM. While Aftersight intends, like *Parikshan*, to enable nearly real-time diagnosis facility, Aftersight suffers from recording overhead in the production VM. The average slow-down in Aftersight is 5% and can balloon up to 2.6x in worst-case scenarios. VARAN [42] is an N-version execution monitor that manages simultaneous executions of a production application, checking among them for divergence. VARAN effectively replicates applications at the system call level, but *Parikshan*'s lower overhead mechanism does not impact the performance of the master (production) application. *Parikshan* also tolerates greater divergence from the production execution, i.e., a debug replica continues to run even if its execution path is modified by the analysis instrumentation. VARAN has recently been applied to run multiple incompatible dynamic analyses in parallel [72] — it would be interesting to use *Parikshan* for this application as well.

Real-Time Diagnosis Techniques: Chaos Monkey [14] injects faults into production systems to conduct fault-tolerance testing, randomly introducing time-outs, resource hogs, etc. This allows Netflix to test the robustness of their system at scale, and avoid large-scale system crashes. AB Testing [31] probabilistically tests updates or beta releases on some percentage of users, while letting the majority of the users continue working with the original application. AB Testing allows the developer to understand user-response to any new additions to the software, which could be used to detect bugs as well as feature problems. Unlike *Parikshan*, this kind of approach directly impacts (some) users.

Live Migration & Cloning Live migration of virtual machines facilitates fault management, load balancing, and low-level system maintenance for the administrator. Most existing approaches use a *pre-copy* approach that copies the memory state over several iterations, and then copies the process state. This includes hypervisors such as VMWare [67], Xen [23], and KVM [50]. VM Cloning, on the other hand, is usually done offline by taking a snapshot of a suspended/shutdown VM and restarting it on another machine. Cloning is helpful for scaling out applications using multiple instances of the same server. Live cloning such as Sun et al. [80] uses copy-on-write mechanisms, to create a duplicate of the target VM without shutting down the original. Other work [37] uses live-cloning to do cluster-expansion.

8 CONCLUSION & FUTURE WORK

Parikshan is a novel framework for live debugging of production SOA applications. We show that in combination with existing bug diagnosis techniques, *Parikshan* successfully localizes several real-world production bugs that would be hard to find otherwise. Compared to existing monitoring solutions that focus on reducing instrumentation overhead, our approach enables minimal performance slowdown while at the same time allowing heavyweight debugging instrumentation. The *Parikshan* prototype is publicly available under the MIT open source license on GitHub [6].

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF CNS-1563555, CCF-1619123, and CNS-1618771.

REFERENCES

[1] [n. d.]. Linux IPC pipes. http://man7.org/linux/man-pages/man7/pipe.7.html.
[2] [n. d.]. Network Namespaces. https://lwn.net/Articles/580893/.
[3] [n. d.]. PetStore a sample Java Platform, Enterprise Edition reference application. http://www.oracle.com/technetwork/java/petstore1-1-2-136742.html.
[4] 2017. Microservices Architecture. https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.
[5] Gautam Altekar and Ion Stoica. 2009. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 193–206.
[6] Nipun Arora. [n. d.]. https://github.com/Programming-Systems-Lab/Parikshan
[7] Nipun Arora. 2018. *Sandboxed, Online Debugging of Production Bugs for SOA Systems*. Ph.D. Dissertation. Columbia University, Columbia University Academic Commons. http://www.nipunarora.net/pdf/sandbox_thesis.pdf.
[8] Shay Artzi, Sunghun Kim, and Michael D. Ernst. 2008. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *ECOOP*.
[9] Jens Axboe. 2008. Fio-flexible io tester. http://freecode.com/projects/fio.
[10] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for Request Extraction and Workload Modelling.. In *OSDI*.
[11] Earl T. Barr and Mark Marron. 2014. Tardis: Affordable Time-travel Debugging in Managed Runtimes. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*. ACM, New York, NY, USA, 67–82.
[12] Daniel J Barrett. 2008. *MediaWiki*. " O'Reilly Media, Inc.".
[13] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. 2013. Chronicler: Lightweight Recording to Reproduce Field Failures. In *International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 362–371. http://dl.acm.org/citation.cfm?id=2486788.2486836
[14] C Bennett and A Tseitlin. 2012. Netflix: Chaos Monkey released into the wild. Netflix Tech Blog.
[15] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*. 154–163.
[16] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, Vol. 41. ACM, 169–190.
[17] Dhruba Borthakur. 2008. HDFS architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf* (2008), 39.
[18] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *26th ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 473–484. https://doi.org/10.1145/2501988.2502050
[19] Yu Cao, Hongyu Zhang, and Sun Ding. 2014. SymCrash: Selective Recording for Reproducing Crashes. In *29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 791–802. https://doi.org/10.1145/2642937.2642993
[20] Josiah L Carlson. 2013. *Redis in Action*. Manning Publications Co.
[21] Jim Chow, Tal Garfinkel, and Peter M Chen. 2008. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. 1–14.
[22] Ben Christensen. 2013. Application Resilience in a Service-oriented Architecture. http://radar.oreilly.com/2013/06/application-resilience-in-a-service-oriented-architecture.html
[23] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 273–286.
[24] James Clause and Alessandro Orso. 2007. A Technique for Enabling and Supporting Debugging of Field Failures. In *29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 261–270. https://doi.org/10.1109/ICSE.2007.10
[25] James Clause and Alessandro Orso. 2011. Camouflage: Automated Anonymization of Field Data. In *33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 21–30. https://doi.org/10.1145/1985793.1985797
[26] Umesh Deshpande and Kate Keahey. 2016. Traffic-sensitive live migration of virtual machines. *Future Generation Computer Systems* (2016).
[27] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 211–224.
[28] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. ACM, 211–224. https://doi.org/10.1145/1060289.1060309
[29] Frank Ch Eigler and Red Hat. 2006. Problem solving with systemtap. In *Proc. of the Ottawa Linux Symposium*. Citeseer, 261–268.
[30] Frank C Eigler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. 2005. Architecture of systemtap: a Linux trace/probe tool. (2005).
[31] Bryan Eisenberg and John Quarto-vonTivadar. 2009. *Always be testing: The complete guide to Google website optimizer*. John Wiley & Sons.
[32] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. 2012. Fay: Extensible Distributed Tracing from Kernels to Clusters. *ACM Trans. Comput. Syst.* 30, 4, Article 13 (Nov. 2012), 35 pages.
[33] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-order Reduction for Model Checking Software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315
[34] Free Software Foundation. [n. d.]. GDB and Reverse Debugging. http://www.gnu.org/software/gdb/news/reversible.html.
[35] Mark Furman. 2014. *OpenVZ Essentials*. Packt Publishing Ltd.
[36] Malay K. Ganai, Nipun Arora, Chao Wang, Aarti Gupta, and Gogul Balakrishnan. 2011. BEST: A Symbolic Testing Tool for Predicting Multi-threaded Program Failures. In *ASE*.
[37] A. Gebhart and E. Bozak. 2009. Dynamic cluster expansion through virtualization-based live cloning. https://www.google.com/patents/US20090228883 US Patent App. 12/044,888.
[38] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2007. Friday: Global Comprehension for Distributed Replay.. In *NSDI*, Vol. 7. 285–298.
[39] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay Debugging for Distributed Applications. In *2006 USENIX Annual Technical Conference*. 289–300.
[40] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: an application-level kernel for record and replay. In *OSDI*. Berkeley, CA, USA.
[41] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. 2008. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 193–208.
[42] Petr Hosek and Cristian Cadar. 2015. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 339–353. https://doi.org/10.1145/2694344.2694390
[43] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs. In *FSE*.
[44] Jeff Huang and Charles Zhang. 2012. LEAN: Simplifying Concurrency Bug Reproduction via Replay-supported Execution Reduction. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 451–466. https://doi.org/10.1145/2384616.2384649
[45] Yanyan Jiang, Tianxiao Gu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2014. CARE: Cache Guided Deterministic Replay for Concurrent Java Programs. In *36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 457–467. https://doi.org/10.1145/2568225.2568236
[46] Wei Jin and Alessandro Orso. 2012. BugRedux: reproducing field failures for in-house debugging. In *2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 474–484. http://dl.acm.org/citation.cfm?id=2337223.2337279
[47] Shrinivas Joshi and Alessandro Orso. 2007. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *ICSM*. https://doi.org/10.1109/ICSM.2007.4362636
[48] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures *(SOSP '15)*. ACM, New York, NY, USA, 344–360. https://doi.org/10.1145/2815400.2815412
[49] David G. Kendall. 1953. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *Ann. Math. Statist.* 24, 3 (09 1953), 338–354. https://doi.org/10.1214/aoms/1177728975
[50] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, Vol. 1. 225–230.
[51] Kirill Kolyshkin. 2006. Virtualization in linux. *White paper, OpenVZ* 3 (2006), 39.
[52] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. 2000. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*. 219–228.
[53] Oren Laadan, Nicolas Viennot, and Jason Nieh. 2010. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 38. ACM, 155–166.
[54] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. 2011. Pervasive Detection of Process Races in Deployed Systems. In *23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 353–367. https://doi.org/10.1145/2043556.2043589

[55] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[56] T. J. LeBlanc and J. M. Mellor-Crummey. 1987. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.* 36, 4 (1987), 471–482.

[57] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. iReplayer: In-situ and Identical Record-and-Replay for Multithreaded Applications. *arXiv preprint arXiv:1804.01226* (2018).

[58] Jian-Guang Lou, Qingwei Lin, Rui Ding, Qiang Fu, Dongmei Zhang, and Tao Xie. 2013. Software analytics for incident management of online services: An experience report. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 475–485.

[59] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5.

[60] Toshinori Matsumura, Takashi Ishio, Yu Kashima, and Katsuro Inoue. 2014. Repeatedly-executed-method Viewer for Efficient Visualization of Execution Paths and States in Java *(ICPC)*.

[61] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *NSDI*.

[62] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. 2008. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*.

[63] David Mosberger and Tai Jin. 1998. httperf—a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review* 26, 3 (1998), 31–37.

[64] mozilla. [n. d.]. what rr does. http://rr-project.org/.

[65] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *32Nd Annual International Symposium on Computer Architecture (ISCA '05)*. IEEE Computer Society, Washington, DC, USA, 284–295. https://doi.org/10.1109/ISCA.2005.16

[66] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races using Replay Analysis. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*.

[67] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. 2005. Fast Transparent Migration for Virtual Machines.. In *USENIX Annual Technical Conference, General Track*. 391–394.

[68] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *PLDI '07*.

[69] Sam Newman. 2015. *Building Microservices.* " O'Reilly Media, Inc.".

[70] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: probabilistic replay with execution sketching on multiprocessors. In *22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. 177–192.

[71] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs *(CGO '10)*. ACM.

[72] Luis Pina, Anastasios Andronidis, and Cristian Cadar. 2018. FreeDA: Deploying Incompatible Stock Dynamic Analyses in Production via Multi-Version Execution. In *ACM International Conference on Computing Frontiers (CF 2018)*.

[73] Tobias Roehm and Bernd Bruegge. 2014. Reproducing Software Failures by Exploiting the Action History of Undo Features. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 496–499. https://doi.org/10.1145/2591062.2591101

[74] Tobias Roehm, Nigar Gurbanova, Bernd Bruegge, Christophe Joubert, and Walid Maalej. 2013. Monitoring user interactions for supporting failure reproduction *(ICPC)*.

[75] Rogue Wave Software. [n. d.]. Reverse debugging with ReplayEngine. http://www.roguewave.com/products-services/totalview/features/reverse-debugging.

[76] Yasushi Saito. 2005. Jockey: A User-space Library for Record-replay Debugging. In *Sixth International Symposium on Automated Analysis-driven Debugging (AADEBUG'05)*. ACM, New York, NY, USA, 69–76. https://doi.org/10.1145/1085130.1085139

[77] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. 2004. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*.

[78] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. 2000. jRapture: A Capture/Replay Tool for Observation-based Testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*. ACM, New York, NY, USA, 158–167. https://doi.org/10.1145/347324.348993

[79] Dinesh Subhraveti and Jason Nieh. 2011. Record and Transplay: Partial Checkpointing for Replay Debugging Across Heterogeneous Systems. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2011)*. San Jose, CA.

[80] Yifeng Sun, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, Binbin Zhang, Haogang Chen, and Xiaoming Li. 2009. Fast live cloning of virtual machine based on xen *(HPCC)*.

[81] Petter Svärd, Benoit Hudzia, Steve Walsh, Johan Tordsson, and Erik Elmroth. 2015. Principles and performance characteristics of algorithms for live VM migration. *ACM SIGOPS Operating Systems Review* 49, 1 (2015), 142–155.

[82] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. 2005. Iperf: The TCP/UDP bandwidth measurement tool. *htt p://dast. nlanr. net/Projects* (2005).

[83] Undo Software. [n. d.]. UndoDB reversible debugging tool for Linux. http://undo-software.com/undodb/.

[84] Erik-Jan van Baaren. 2009. Wikibench: A distributed, wikipedia based web application benchmark. *Master's thesis, VU University Amsterdam* (2009).

[85] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. DoublePlay: Parallelizing Sequential Logging and Replay. *ACM Trans. Comput. Syst.* 30, 1, Article 3 (Feb. 2012), 24 pages. https://doi.org/10.1145/2110356.2110359

[86] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. 2014. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of annual IEEE/ACM international symposium on code generation and optimization*. ACM, 98.

[87] Min Xu, Rastislav Bodik, and Mark D. Hill. 2003. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *30th Annual International Symposium on Computer Architecture (ISCA '03)*. ACM, 122–135. https://doi.org/10.1145/859618.859633

[88] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. 2014. Determinism Is Not Enough: Making Parallel Programs Reliable with Stable Multithreading. *Commun. ACM* (2014).

[89] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 249–265.

[90] Long Zheng, Xiaofei Liao, Bingsheng He, Song Wu, and Hai Jin. 2015. On Performance Debugging of Unnecessary Lock Contentions on Multicore Processors: A Replay-based Approach. In *(CGO '15)*.