Parikshan: Live Debugging of Production Systems in Isolation

Nipun Arora
NEC Labs
nipun@nec-labs.com

Franjo Ivančić
Google
ivancic@google.com

Gail Kaiser
Columbia University
kaiser@cs.columbia.edu

ABSTRACT

Modern 24x7 SOA applications rely on short deployment cycles, and fast bug resolution to maintain their services. Hence, time-to-bug localization is extremely important for any SOA application. We present *live debugging*, a mechanism which allows debugging of production systems (run test-cases, debug, or profile, etc.) on-the-fly. We leverage user-space virtualization technology (OpenVZ/LXC), to launch containers cloned and migrated from running instances of an application, thereby having two containers: *production* (which provides the real output), and *debug* (for debugging). The *debug container* provides a sandbox environment for debugging without any perturbation to the production environment. Customized network-proxy agents replicate or replay network inputs from clients to both the production and debug-container, as well as safely discard all network output from the debug-container. We used our system, called `Parikshan`, to do *live debugging* on several real-world bugs, and effectively reduced debugging complexity and time.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: [distributed debugging, tracing, monitoring]

Keywords

software debugging, live cloning, network duplication

1. INTRODUCTION

Rapid resolution of incidence(error/alert) management [31] in online service oriented systems is extremely important. However, the complexities of virtualized environments coupled with large distributed systems have made bug localization harder. The large size of distributed systems means that any downtime has significant financial penalties for all parties involved. This problem is further compounded by the recent trend in software engineering industry towards DevOps [9]. DevOps stresses on close coupling between software developers and operators, in order to have shorter release and debug cycles (Facebook mobile has 2 releases a day, and Flickr has 10 deployment cycles per day [10]). This re-emphasizes the need to have a very short time to diagnose and fix a bug. Hence, debugging is hard not only because of difficulty to capture the root-cause, it is also increasingly important to localize and fix bugs in a very short period of time.

Existing state-of-art techniques for monitoring production systems [33, 13, 39] rely on light-weight dynamic instrumentation to capture execution traces. Operators then feed these traces to analytic tools [15, 42] or do offline debugging, to find the root-cause of the error. However, dynamic instrumentation has a trade-off between granularity of tracing and the performance overhead. Operators keep instrumentation granularity low, to avoid higher overheads in the production environment. This often leads to multiple iterations between the debugger and the operator, to increase instrumentation in specific modules, in order to diagnose the root-cause of the bug.

Another body of work has looked into record-and-replay [11, 21, 29, 24] systems which capture execution traces, in order to faithfully replay them in an offline environment. These systems try and capture system level information, user-input, as well as all possible sources of non-determinism, to allow for in-depth *post-facto* analysis of the error. However, owing to the amount of instrumentation required, record-and-replay tools deal with an even heavier overhead, making them impractical for real-world production systems.

Our system, called `Parikshan`[1], allows **real-time debugging and inspection** without any performance impact on the production service. We provide a facility to sandbox the production and debug environments such that any modification in the debug environment does not impact user-facing operations. `Parikshan` avoids the need for large test-clusters, and can target specific sections of a large scale distributed application. In particular, `Parikshan` allows system administrators to apply debugging techniques with deeper granularity instrumentation, without impacting performance.

`Parikshan` is composed of three modules: (a) a *clone manager*, which manages cloning of production containers to debug-containers. (b) a *network duplicator* module, which duplicates all incoming traffic from downstream servers to both production and debug containers. (c) a *network aggregator* module, which manages all communication from upstream servers to the debug-container. The cloning operation is "live", hence there is no suspension of the services

[1] `Parikshan` is the Sanskrit word for testing

of the production server. The debug container acts like a sandbox which restricts it from causing any perturbation to the state of the parent process, or impacting the sanity of the responses to the production client. This allows debuggers to use debugging tools without any fear of crashing or modifying the production application.

Studies have shown [43, 30, 28] that several bugs lead to either an inconsistent output, or impact the performance of the application, without crashing the system. `Parikshan` primarily focuses on such *non-crashing bugs*, where the application continues to get input from the user. Examples of these bugs are slow memory leaks, configuration, or performance bugs, which do not crash the application, but need to be fixed quickly to avoid degradation in service quality. Although many interesting methods have been developed to trace crashing bugs (memory violation, core dumps etc.), it is still difficult to analyze non-crashing bugs, as they often happen in scaled out systems, or because of difficult to reproduce edge-case scenarios, that are hard to replicate in unit tests or integration tests.

We have deployed `Parikshan` in the context of cloud IAAS platform using user-space container virtualization technologies (OpenVZ/LXC [4, 2]). We assume that the target system utilizes micro-service architectures (`Docker` [34]), where each service (application, DNS, indexing, storage) is sandboxed in separate containers. This allows us to launch debug-containers, which can target one application at a time.
The key contributions of our system are:

- **Secure & sandboxed debugging:** `Parikshan` provides a cloned sandbox environment to debug the production application. This allows a safe and secure mechanism to diagnose the application, without impacting the functionality of the production container.
- **No performance impact:** A network duplication and aggregation mechanism, which ensures non-blocking request duplication to the debug-container, thereby ensuring no performance impact on the application.
- **Capture large-scale context:** Allows to capture the context of large scale production systems, with long running applications. Under normal circumstances capturing such states is extremely difficult as they need a long running test input, and large test-clusters.
- **Short time-to-debug:** These techniques contribute to a shortened debug time, by allowing system debuggers to directly gather trace data rather than wait for operators.

For the remainder of this paper, we define **downstream servers** as servers from which requests are being sent to the production container, and **upstream servers** as servers to which the production container sends request. In figure 2, the webserver is downstream, and the backend is upstream of our target container.

The rest of the paper is organized as follows. In section 2, we describe a motivating scenario. Section 3, describes the design of our system, and section 4 describes the implementation. Next we discuss several real-world bugs, which we analyzed using `Parikshan`. Finally, we evaluate our system, and conclude.

2. MOTIVATING SCENARIO

As a motivating scenario, let us take a complex multi-tier service-oriented system (Figure 1), with several interacting services (web-servers, application servers, search and indexing, database etc.). The system is maintained by an operator, who can observe the health of the system using light-weight monitoring, which is part of the deployed system. In the interest of application performance, production system monitoring is usually limited to system resource usage, application usage statistics, transaction logs, and error logs.

At a certain time in the execution of the system, the operator observes unusual memory usage in the glassfish application server, and some error logs being generated in the nginx web-server. He surmises, that there is a potential memory leak/allocation problem in the app-server, or a problem in the web-server. However, with a limited amount of monitoring information, he can only go so far.

Typically, trouble tickets are generated for such problems, and they are debugged offline. However using `Parikshan`, the operator can fork off clones of the nginx and glassfish containers as ***nginx-debug*** and ***glassfish-debug***. Our network duplication mechanisms ensure that the debug containers receive the same inputs as the production containers, and that the production containers continue to provide service without interruption. This separation of the production and debug environment, allows the operator to use dynamic instrumentation tools to perform deeper diagnosis without fear of additional disruptions due to debugging. Since the system has been cloned from the original potentially "buggy" production container, it will also exhibit the same memory leaks/or logical errors. Additionally, `Parikshan` can focus on the "buggy" parts of the system, without needing to replicate the entire system in a test-cluster. This process will greatly reduce the time to bug resolution, and allow real-time bug diagnosis capability.

3. DESIGN

In Figure 2, we show the architecture of `Parikshan` when applied to a single mid-tier application server. `Parikshan` consists of 3 modules: **Clone Manager**: manages "live cloning" between the production and the debug containers, **Network Duplicator**: manages network traffic duplication from downstream servers to both the production and debug containers, and **Network Aggregator**: manages network communication from the production and debug containers to upstream servers. 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. Furthermore, the aggregator module is not required if the debug-container has no upstream services. At the end of this section, we also discuss the **debug window** during which we believe that the debug-container faithfully represents the execution of the production container. Finally, we discuss **divergence checking** which allows us to observe if the production and debug containers are still in sync.

3.1 Clone Manager

Before describing cloning in our context, let us first review some details about live migration [35, 20, 23]. Live migration refers to the process of moving a running virtual machine, or container from one server to another, without disconnecting any client or process running within the machine. In general, this involves the following steps: (1) A pre-copy `rsync`, whereby the underlying hypervisor copies all the memory pages from the source node to the destination node. (2) Once the pre-copy phase is finished, the VM is temporarily suspended, and all pages which were modified are transferred to the target node. The second step can

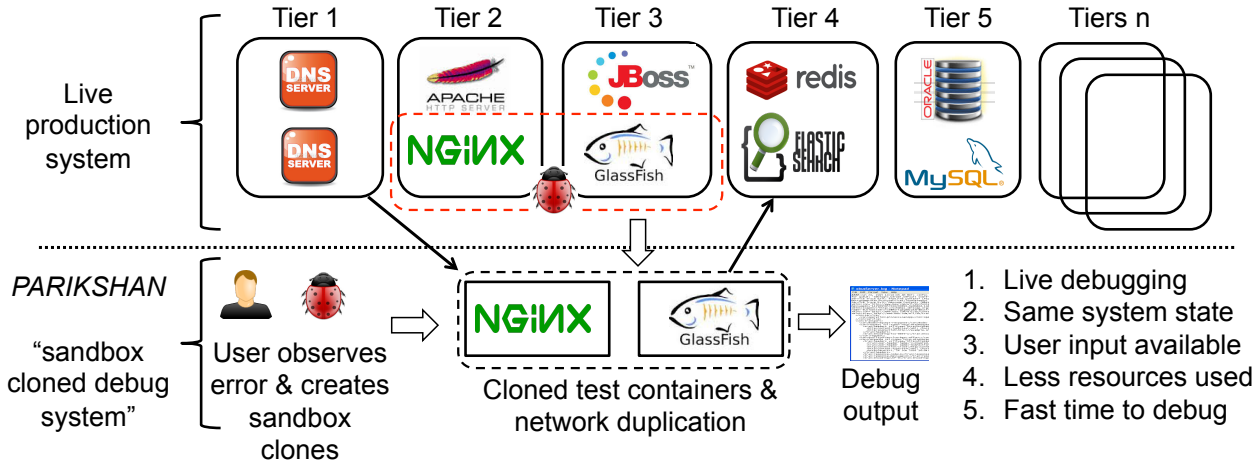

Figure 1: Workflow of `Parikshan` in a live multi-tier production system with several interacting services. When the administrator of the system observes errors in two of it's tiers, he can create a sandboxed clone of these tiers and observe/debug them in a sandbox environment without impacting the actual production system.

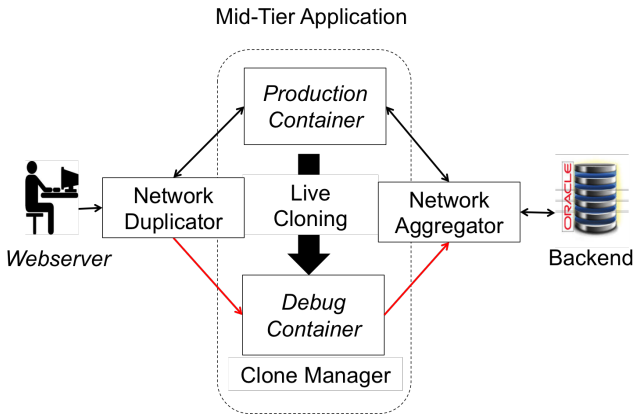

Figure 2: `Parikshan` applied to a mid-tier service: It is comprised of: (1) Clone Manager for Live Cloning, (2) Proxy Duplicator to duplicate network traffic, and (3) Proxy Aggregator to replay network traffic to the cloned debug container.

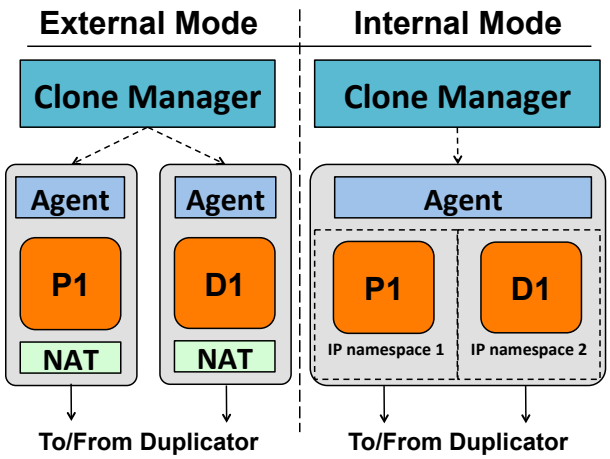

Figure 3: External and Internal Mode for live cloning: P1 is the production container, and D1 is the debug container, the clone manager interacts with an agent which has drivers to implement live cloning.

be done in multiple iterations to avoid a long suspend time, in-case the memory pages have significantly changed after the first rsync.

In contrast, live migration where the original container is destroyed, the "Live Cloning" process used in `Parikshan` requires both containers to be actively running, and be still attached to the original network. 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 the IP address, which cannot be changed on the fly. Hence, the same network identifier should map to two separate addresses, and have communication with no problems or slowdowns.

We now describe two modes (see Figure 3) in which cloning has been applied, followed by the algorithm for live cloning:

Internal Mode: In this mode we allocate the production and debug containers to the same host node. This would mean less suspend time, as the production container can be locally cloned (instead of streaming over the network). Additionally, it is more cost-effective since the number of servers remain the same. On the other hand, co-hosting the debug and production containers could potentially have an adverse effect on the performance of the production container because of resource contention.

To manage network identities, we encapsulate each container in separate network namespaces. This allows both the containers, to have the same IP address, although in separate interfaces. The duplicator is then able to communicate to both these containers with no networking conflict.

External Mode: In this mode we provision an extra 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 (dependent on workload), and requires provisioning an extra host-node, 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. We believe that external mode will be more practical in comparison to internal mode, as cloning is likely to be transient, and high network bandwidth between physical hosts can offset the slowdown in cloning performance.

Network identities in external mode are managed using NAT [3] (network address translator) in both host machines. Hence both containers can have the same address without any conflict.[2]

Algorithm: In our current implementation, we are using OpenVZ [4] as our container engine, and leverage OpenVZ tools [8] [35] to support cloning. We tested this out on multiple VM's acting as host nodes for OpenVZ containers. To make the cloning easier and faster, we used OpenVZ's *ploop* devices [6] as the container disk layout. *Ploop* devices are a variant of disk loopback devices where the entire file system of the container is stored as a single file. This allows for easier snapshots/checkpoints, as *ploop* ensures separation of inodes of each container file system.

Let's imagine we are cloning production container P1 on Host H1 as debug-container D1 on Host H2. The initial setup requires certain safety checks and pre-processing to ensure easy cloning. These include: ssh-copy-id operation for password-less rsync, checking pre-existing container ID's, version of OpenVZ etc. This ensures that H1 and H2 are compatible, and ready for live-cloning. Next, we run an initial rsync of container P1, from Host H1 to Host H2. This step does not require any suspension of P1, and copies the bulk of the container file system to the destination server (H2). The next step involves checkpointing, and dumping the process in memory state to a file. This is what allows the container to be restarted from the same checkpointed state. Since the first rsync and the memory dump are non-atomic operations, some of the files in the file system may be outdated. We freeze the processes in the production-container, and take a second rsync of the ploop device of P1, to ensure both containers have the same file-system. Simultaneously, we setup the networking proxy and aggregation infrastructure to allow for communication from the downstream and upstream servers. Next we copy the dump file from step 3, from H1 to H2, and resume both containers.

The suspend time of cloning depends on the operations happening within the container between step 2 and step 4 (the first and the second rsync), as this will increase the number of dirty pages in the memory, which in turn will impact the amount of memory that needs to be copied during the suspend phase. In section 6.1, we evaluate the performance of live cloning on some real-world examples and a micro-benchmark of I/O operations.

3.2 Proxy Network Duplicator

[2]Another additional mode can be *Scaled Mode*: This can be viewed as a variant of the external mode, where we can execute debug analysis in parallel on more than one debug-containers each having it's own cloned connection. This will distribute the instrumentation load and allow us to do more analysis concurrently, without overflowing the buffer. We aim to explore this in the future.

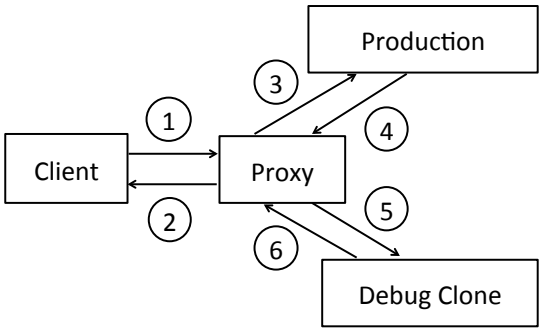

Figure 4: Description of the Network Duplicator. In *synchronized* mode: Thread 1 executes steps [1,3,5], and Thread 2 executes [2,4,6] sequentially. In *asynchronous* mode: Thread 1 executes steps [1,3], Thread 2 executes [2,4], Thread 3 executes [5], and Thread 4 executes [6]

In order for *live debugging* to work, both production and debug containers must receive the same input. A major challenge in this process is that the production and debug container may execute at different speeds (debug will be slower than production), which will result in them being out of sync. Additionally, we need to accept responses from both servers, and drop all the traffic coming from the debug-container, while still maintaining an active connection with the client. Hence simple port-mirroring and proxy mechanisms will not work for us.

Our solution is a customized TCP level proxy, which also 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, the applications are completely oblivious of it. In this section, we discuss two strategies (we call *duplication modes*) to duplicate and forward network traffic to the debug and production containers.

3.2.1 Duplication Modes

Synchronized Packet Forwarding Mode: A naive strategy is to have a single worker thread to send and receive TCP stream to the production container as well as the debug container. To understand this better let us look at figure 4: Here each incoming connection would be handled by 2 parallel threads T1, and T2 in the proxy. Where T1 sends data from the client to the proxy (link 1), then sends data from proxy to production (link 3), and finally from proxy to debug container (link 5). Whereas, T2 sends replies from production to proxy(link 4), then receives replies from debug container to proxy (link 6), which are then dropped. Thread T2 then forwards packets received on link 4 to the client on link 2.

By design TCP is a connection oriented protocol and is designed for stateful delivery and acknowledgment that each packet has been delivered. Packet sending and receiving are blocking operations, and if either the sender or the receiver is faster than the other the send/receive operations are automatically blocked or throttled.

This can be viewed as follows: Let us assume that the client was sending packets at $XMbps$ (link 1), and the production container was receiving/processing packets at $YMbps$ (link 3), where $Y < X$. Then automatically, the speed of

link 1 and link 2 would be throttled to $YMbps$ per second, i.e the packet sending at the client would be throttled to accommodate the production server. Network throttling is a default TCP behavior to keep the sender and receiver synchronized. However, we also send packets to the debug-container (link 5) in the same sequential thread T1. Hence, if the speed of link 5 is $Z\ Mbps$, where $Z < Y$, and $Z < X$, then the speed of link 1, and link 3 would also be throttled to $Z\ Mbps$.

The speed of the debug container is likely to be less than the production container, hence this would definitely impact the performance of the production container. Clearly, this is a non-optimal solution. Next we discuss an asynchronous packet forwarding scheme, which avoids any slowdown to the production container.

Asynchronous Packet Forwarding Model: In the asynchronous packet forwarding mode (see Figure 4): we use 4 threads T1, T2, T3, T4 to manage each incoming connection to the proxy. Thread T1 forwards packets from client to proxy (link 1), and from proxy to production container (link 2). It then uses a non-blocking send to forward packets to an internal pipe buffer shared between thread T1, and thread T3. Thread T3, then reads from this piped buffer and sends traffic forward to the debug-container. Similarly Thread T2, receives packets from production container, and forwards them to the client, while Thread T4, receives packets from the debug-container and drops them.

The advantage of this strategy is that any slowdown in the debug-container will not impact the production container's communication as a non-blocking send is used to forward traffic to the production container. A side-effect of this strategy is that if the speed of the debug-container is too slow compared to the production container, it may eventually lead to a buffer overflow. We call the time taken by a connection before which the buffer overflows it's *debug-window*. We discuss the implications of the *debug window* in section 3.4.

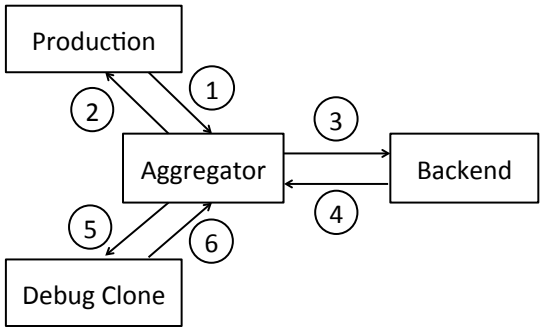

Figure 5: Description of the Network Aggregator. Thread 1 executes step [1,3], Thread 2 executes [2,4], and Thread 3 executes [5], and Thread 4 executes [6]

3.3 Proxy Network Aggregator & Replay

The proxy described in section 3.2 is used to forward requests from downstream tiers to production and debug containers. This manages incoming "requests" to the target container, however the same mechanism cannot be directly applied for isolating responses from upstream servers. Imagine if you are trying to debug a mid-tier application container, the proxy network duplicator will replicate all incoming traffic from the client to both debug and the production container. Both the debug container and the production, will then try to communicate further to the back-end containers. This could mean duplicate queries to the backend servers (for e.g. duplicate deletes to MySQL), thereby leading to an inconsistent state. However, to have forward progress the debug-container must be able to communicate and get responses from upstream servers. 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. As well as dropping all packets sent from it to upstream servers.

As shown in Fig 5, when an incoming request comes to the aggregator, it first checks if the connection is from the production container or debug container. In case of the production container (link 1), the aggregator forwards the connection to the backend (link 3), responses from the backend are sent to the aggregator (link 4), and then forwarded to the production container (link 2) and simultaneously saved in an internal queue. The aggregator creates an in-memory persistent inter-process FIFO queue for each connection where the responses for each of these connections are stored. When the corresponding connection from the duplicate debug container connects to the proxy (link 5); all packets being sent are quietly dropped. The aggregator then uses the queue to send replies to the debug-container (link 6). In a way this is a streaming online record-and-replay, where we are recording the data in our buffer. 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 stack 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.

In this design we assume that the order of incoming connections remains largely the same. We use a fuzzy checking mechanism using the hash value of the data being sent to correlate the connections. Each queue has a short wait time to check against incoming connections, this allows us to match slightly out of order connections. In case a connection cannot be correlated, we send a TCP_FIN, to close the connection, and inform the debugger.

3.4 Debug Window

In the asynchronous forwarding mode, an unblocking send forwards traffic to a separate thread which manages communication to the debug-container. This thread has an internal buffer, where all incoming requests are queued, and subsequently forwarded to the debug-container. 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). Whereas the outgoing rate from the buffer is dependent on how fast the debug-container processes the requests. Instrumentation overhead in the debug-container is likely to cause an increase in the debug-container, The time period till buffer overflow happens is called the *debug-window*. This depends on the size of the buffer, the incoming request rate, and the overhead induced in the debug-container. For the duration of the debugging-window, we assume that the debug-container faithfully represents the production container. Once the buffer has overflown, the

production container may need to be cloned again to ensure it has the same state.

The debug window size also depends on the application behavior, in particular how it launches TCP connections. `Parikshan` generates a pipe 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 happens 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", hence the proxy is able to tolerate significant slowdowns. Database servers, 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, the amount of calls in a single session can eventually have a cumulative effect to cause overflows. Photosharing, file sharing and other datasharing services transfer a lot of data in a single burst over each TCP connection. These protocols will have an overflow relatively sooner, and will be able to tolerate only small amounts of slowdown.

To further increase the *debug window*, we propose load balancing debugging across multiple debug-containers, which can each get a duplicate copy of the incoming data. This would mean that there are multiple threads handling the incoming connection, one for the production container, and one for each of the debug containers. We believe that such a strategy would significantly reduce the chance of a buffer overflow in cases where the debug-container is significantly slower. We evaluate debug-window overflows in section 6.2.

3.5 Divergence Checking

Record-and-replay systems capture all possible sources of non-determinism, which ensures that they replay the exact same trace. However, in `Parikshan` it is possible that non-deterministic behavior in the two containers, or modifications because of user instrumentation, causes the production and debug container to diverge with time. To understand and capture this divergence, we compare the corresponding network output received in the proxy. This is an optional component, which gives us a black-box mechanism to check the fidelity of the debug-container based on it's communication with external components. Comparisons use hash of data packets, which are collected and stored in memory for the duration that the connections are active. The degree of acceptable divergence (the point till which the production and debug containers can be assumed to be in sync) is dependent on the application behavior, and the operator. For example, an application which is sending timestamps in each of it's send messages can be expected to have a much higher degree of divergence, in comparison to an application which is querying a database.

4. IMPLEMENTATION

`Parikshan` is built on top of user-space container virtualization software, OpenVZ [4], with Centos 6.5 with Linux Kernel 2.6.32. We implemented the system in 3 different configurations: (1). We applied `Parikshan`'s internal mode configuration by installing it in a single host OS VM with Intel i7 CPU, with 4 Cores, and 16GB RAM. Containers were cloned within the machine, with a separate VM acting as the client. We used NAT, and IP namespaces for network access to the VM's. (2). We implemented our system's external mode on the base kernel in identical host nodes. Each of these host nodes have an Intel Core 2 Duo Processor, 8GB of RAM, and ran Centos 6.5 with Linux Kernel 2.6.32. While the current virtualization is based on containers, we believe that `Parikshan` can easily be applied to traditional virtualization softwares where live migration has been further optimized. The reason for choosing a container based implementation was that containers take much less resources in comparison to traditional VM's.

The network proxy was implemented in C/C++. The forwarding in the proxy is done by forking off multiple processes each handling one send/or receive connection in a loop. Data from processes handling communication with the production container, is transferred to those handling communication with the debug containers using pipes. Pipe buffer size is a configurable input based on user-specifications.

5. CASE STUDIES

Production applications are usually deployed with transaction logs, and error logs that alert the operator of any anomalous behavior. Debuggers use these logs, to run offline test-cases in order to find the root-cause of any bugs. This process may need multiple iterations between the operator and developer, as more information may be required from the production system. This eventually increases the time to debug the error.

The idea behind `Parikshan` is to allow operators to use debugging tools and mechanisms in the production environment, that one would normally use in the development environment. These tools allow developers to better understand program flow and data flow, which is key to localizing the root cause of any bug. Dynamic instrumentation tools like DTrace [33], and iProbe [13] can generate *execution traces*, which give insight into the program flow (functions, and conditional statements executed). iProbe has statically compiled "hooks" in the beginning and end of functions in the target applications, which can be patched with dynamic instrumentation on-the-fly. On the other hand, DTrace uses interrupt mechanisms to insert a probe at any point in the application using symbolic information. Similarly, tools like dyninst [18] allow users to track data-flow of variables in the code. *Memory profiling* tools like Valgrind [38], PIN [32], and VisualVM [7] give insight into memory allocation and deallocation, thus helping in catching memory leaks. Next, we discuss how these techniques can be applied to different categories of real-life bugs with the help of `Parikshan`:

Performance Bugs: One of the most subtle problems that manifest in production systems is performance bugs. These bugs do not usually lead to crashes, but cause significant impact to user satisfaction. A casestudy [26] showed that a large percentage of real-world performance bugs can be attributed to uncoordinated functions, executing functions that can be skipped, and inefficient synchronization among threads (for example locks held for too long etc.). Typically, such bugs can be caught by function level execution tracing and tracking the time taken in each execution function. Another key insight provided in [26] was that two-thirds of the bugs manifested themselves when special input conditions were met, or execution was done at scale. Hence, it is difficult to capture these bugs with traditional offline white-box testing mechanisms.

In an empirical evaluation of three such real-world bugs, we used `Parikshan` and iProbe [13] (a dynamic instrumen-

Bug Type	Bug Desc	Application	Tool Used	Debug Mechanism	Slow-down	Nodes Cloned	Comments/ Bug Caught
Performance Bug	#15811	MySQL	iProbe	Exec. Trace	1.5x	1	Yes
	#45464	Apache	iProbe	Exec. Trace	1.4x	1	Yes
	#49491	MySQL	iProbe	Exec. Trace	1.8x	1	Yes
Memory Leak	Injected	Glassfish	VisualVM/ mTrace	Memory Profiling	3x	1	Yes
	Injected	MySQL	Valgrind	memcheck	N.A	1	Yes, with overflows
Config. Errors	DNS	JBOSS	DTrace	Tracing	1.3-4x	2	Yes
	Max no. of threads	MySQL	DTrace	Tracing	1.5x	2	Yes

Table 1: Case-Study of production system bugs, with approximate slowdowns, debug mechanisms, and tools used.

tation tool) to capture a representative function trace. The tracing caused a 1.5x - 2x slowdown in the targeted session of the application, with no impact on the production system. For one of the bugs in MySQL (Bug 15811), it was 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. We generated a random workload of user-requests, including queries that triggered the bug. Now the slow queries may be reported by either a user, or a transaction latency monitoring available in several production systems. Based on this trigger, we created a live clone of the MySQL container. We then inserted function level instrumentation in MySQL for higher level functions, object constructors, and destructors belonging to each module in the code. Finally, we used the query handler to trigger instrumentation for queries containing complex scripts(this ensures no slow-down for other queries). We then compared the profile (time taken in each function) of the queries for complex scripts, with the queries for Latin scripts. Using trial-and-error, and iteratively digging into deeper functions, we were able to localize the bug to string compare functions in MySQL.

Memory Leaks: Memory leaks are a common error in service oriented systems, especially in C/C++ based applications which allows low-level memory management by users. These leaks build up over time, and can cause slowdowns because of resource shortage, or crash the system. Debugging leaks can be done either using systematic debugging tools like Valgrind, which use shadow memory to track all objects, or memory profiling tools like VisualVM, mTrace, or PIN, which track allocations, de-allocations, and heap size. Although valgrind is more complete, it has a very high overhead, and needs to capture the execution from the beginning to the end (i.e., needs application restart). On the other hand, profiling tools are much lighter, and can be dynamically patched to a running process.

For our empirical evaluation, we injected repetitive memory allocations to an object in a glassfish app-server, and sent requests to app-server to trigger these allocations. We then, used `Parikshan` to clone & create a debug-container, where we used VisualVM to track the increase of heap-size, and object allocations. Tracking memory allocation allowed us to localize the class file causing the memory leak, which greatly simplifies finding the root-cause. We experienced a slowdown of 3x, but did not observe any overflow in the buffer.

We also used memory leak detection using Valgrind in MySQL for an injected memory leak in the request handler for MySQL. We wish to stress here, that while tools like valgrind, need application restart, and will most likely cause extremely high slow-downs in the debug-container, `Parikshan` can still be helpful, as it can debug a small part of the target system in isolation, thereby quickly resolving errors. Application service restart may not lose context in all applications, especially a database application has a persistent database, and may exhibit errors even after the restart. With Valgrind, we saw requests being dropped in the proxy because of the high overhead, but we were able to capture and trace memory leaks for several requests.

Configuration Errors: Configuration errors are usually caused by wrongly configured configuration parameters, i.e., they are not a bugs in the application, but bugs in the input (configuration). Configuration bugs usually get triggered at scale, or for certain edge cases, making them extremely difficult to catch.

For our empirical evaluation, we created a 3 tier (Apache, JBoss, MySQL) service using Java PetStore [5] a J2EE example application. We inserted a configuration error by setting an arbitrarily low number for the max. number of requests MySQL can handle. We then triggered requests to PetStore, and found some of them timing out. Using `Parikshan`, we cloned both JBoss and MySQL servers, to captured the flow of queries across distributed machines. We traced higher object constructors, and library calls to track the execution. This allowed us to see that the MySQL server was not creating worker threads for the incoming requests(no calls to libc fork events). Debug logs had been turned off for optimization, hence the error could not be easily reported. `Parikshan` was able to successfully localize the error to misconfiguration in the "maximum no. of worker thread" allowed parameter. While the error may seem simple, without `Parikshan` an offline testbed would be required with a large enough test workload.

6. EVALUATION

To evaluate the performance of `Parikshan`, we pose and answer the following research questions:

RQ1: How long does it take to create a live clone of a production container and what is it's impact on the performance of the production container?

RQ2: What is the size of the debugging window, and how does it depend on various resource constraints?

6.1 Live Cloning Performance

As explained in section 3, live cloning requires a small suspend time, which has minimal impact on users. Suspending is necessary to ensure that both containers are in the exact same system state. Broadly, suspend time during live cloning can be divided in 4 parts: (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. (4) Undump & Resume: time taken to resume the containers. To evaluate "live cloning", we ran a micro-benchmark of I/O operations, and evaluated live-cloning on some real-world applications running real-workloads:

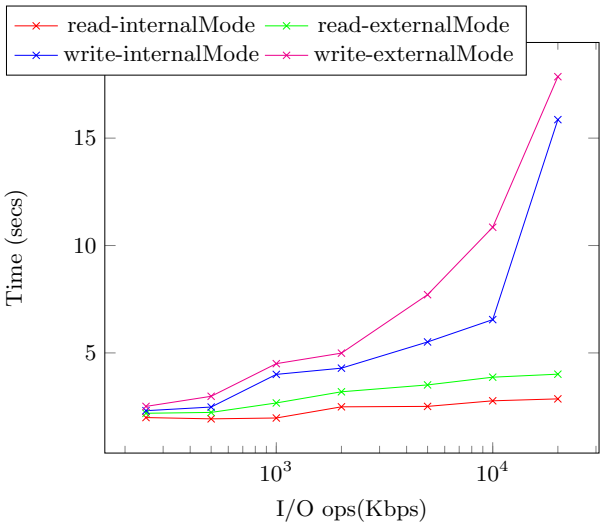

Figure 6: Live Cloning suspend time with increasing amounts of I/O operations

Micro Benchmark using I/O operations: The main factor that impacts suspend time is the number of "dirty pages" in the suspend phase, which have not been copied over in the pre-copy rsync operation (see section 3.1). To understand this better, we use fio (flexible I/O tool for Linux) [14], to gradually increase the number of I/O operations while doing live cloning. Fio reads or writes random values in a file with a rate controlled I/O bandwidth as specified by the user. The suspend time is observed by instrumentation within the cloning script, which reports time taken by each of the suspend processes. Additionally, we ensure that the I/O job being processed by fio is long enough to last through the entire cloning process.

We found that in comparison to write, read operations have a much smaller impact on suspend time of live cloning. This can be attributed to the increase of "dirty pages" in write operations, whereas for read, the disk image remains largely the same. We also found, that internal mode is much faster than external mode, as both the production and debug-container are hosted in the same physical device. We believe, that for higher I/O operations, with a large amount of "dirty-pages", network bandwidth becomes a bottleneck: leading to longer suspend times. Overall, 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 upto 5-6 mbps, for a 5 sec suspend time.

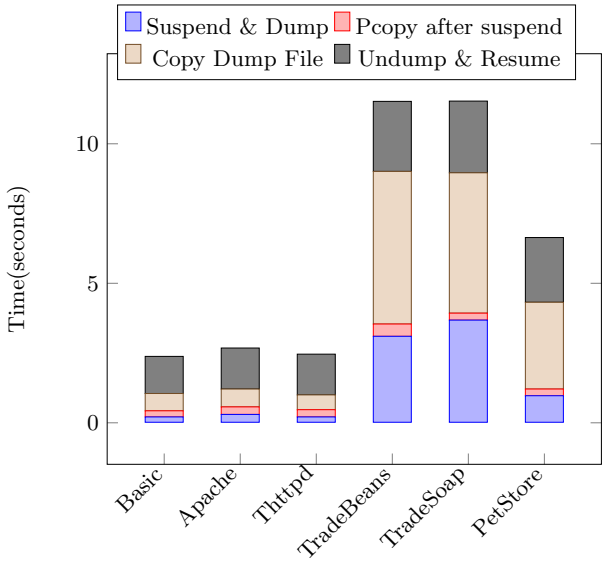

Figure 7: Suspend time for live cloning, when running a representative benchmark

Real-World application, and workloads: In figure 7, we show the suspend times for five well known applications - Apache, Thttpd, TradeBeans, TradeSoap, PetStore. For Apache, and thttpd which are web-servers, we ran the httperf [36] hog benchmark. The hog benchmark tries to compute max throughput of the web-server, by sending a large number of concurrent requests. Tradebeans and Tradesoap are both part of the dacapo [17] benchmark "DayTrader" application. These are realistic workloads, which run on a multi-tier trading application provided by IBM. PetStore [5] is a well known J2EE reference application. We deployed PetStore in a 3-tier system with JBoss, MySQL and Apache servers. Here we cloned the app-server, and the input workload was a random set of transactions which were repeated for the duration of the cloning process. We found that for hog benchmarks, the container suspend time was 2-3 seconds. However, for heavy workloads in more memory intensive application servers such as PetStore, DayTrader, the total suspend time was higher (6-12 seconds). We found that we did not experience any timeouts or errors for the requests in the workload[3]. However, this did slowdown requests in the workload. This confirmed our hypothesis that short suspend times are largely not visible/have minimal performance impact to the user, as they are within the time out range of most applications. Further a clean network migration process ensures that connections are not dropped, and the workload is executed successfully.

[3]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

To answer **RQ1**, live cloning introduces a short suspend time in the production container, the duration of the suspend time depends on the workload. Typically write intensive workloads will lead to longer suspend times, while read intensive workloads will take very less. Additionally, we ran live cloning on some real world applications and found suspend time to vary from 2-3 seconds for webserver workloads to 10-11 seconds for application/database server workloads. Live cloning can be viewed as an amortized cost paid upfront, instead of recording overhead throughout the execution. For our future work, we aim to reduce suspend time, rate-limiting of incoming requests in the proxy, or copy-on-write mechanisms between the production container and the clone.

6.2 Debug Window Size

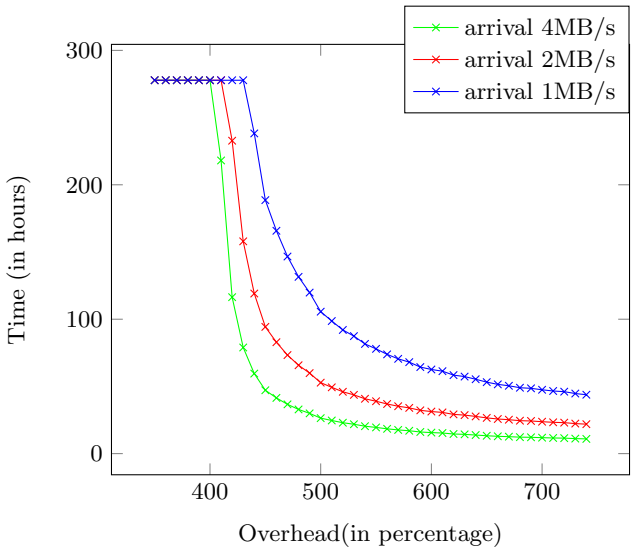

Figure 8: Simulation results for debug-window sizes with gradually increasing the overhead in service processing times. The buffer is kept at a constant 64GB.

Debug Window Size: We call the time taken to reach a buffer overflow the "debug-window" for the debug-container. As explained earlier (see section 6.2), the size of this debug-window, depends both on the overhead of the "instrumentation", the incoming workload distribution, and the size of the buffer. To evaluate the approximate size of the debug-window, we sent requests to 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[4] in a single user connect session. For each of our experiments we vary the buffer sizes to get an idea of debug-window. Additionally, we generated a slowdown by first modeling the time taken by MySQL to process requests (27 req/s or 17req/s), and then putting an approximate sleep in the request handler.

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 had a linear increase primarily based 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 let's the cloned debug-container catch up with the production container during idle time-periods in between request bursts. We observed, that with the default buffer size (linux default pipe size is 65536), the system was able to tolerate a much higher overhead (3.2x) with no buffer overflows.

In our next set of experiments, we used a small simulator, which mimics a queuing SOA system [**?**, **?**], with packet inter arrival rates and processing times distributed based on a poisson distribution. As discussed earlier, there are three parameters which can impact the time period of the debug-window: (1) arrival rate (λ), (2) service processing time (μ), and (3) Buffer Size. In our simulations , 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 experiment), 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. Each simulation instance was run for 1000000 seconds or 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 in Figure 8, there is no buffer overflow in any of the experiments till the overhead reaches around 420-470%, beyond this the debug-window decreases exponentially

To answer **RQ2**, we found that the debug window size depends on the incoming workload, the slowdown caused due to instrumentation overhead, and the size of the buffer. We conclude that with a large enough buffer size, most applications will have a sufficiently long debug-window (several minutes), to allow them to capture bug traces, and find the error. Buffer overflow will typically happen only for very large spikes in a connected session from the user.

Input Rate	Debug Window	Pipe Size	Slow-down
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
Poisson, $\lambda = 17$ rq/s	16 sec	4096	8x
Poisson, $\lambda = 17$ rq/s	18 sec	4096	5x
Poisson, $\lambda = 17$ rq/s	∞	65536	3.2x
Poisson, $\lambda = 17$ rq/s	376 sec	16384	3.2x

Table 2: Approximate debug window sizes for a MySQL request workload

7. APPLICATION

Statistical Testing: One well known technique for debugging production application is statistical debugging. This is achieved by having predicate profiles from both successful and failing runs of a program and applying statistical

[4]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

techniques to pinpoint the cause of the failure. The core advantage of statistical debugging is that the sampling frequency of the instrumentation can be decreased to reduce the instrumentation overhead. Despite it's advantages the instrumentation frequency for such debugging to be successful needs to be statistically significant. Furthermore, unlike Parikshan, statistical debugging will impose an overhead on the actual application.

We believe, that statistical debugging can be successfully applied in the debug-container when an error has been observed. The entire scope of such a live debugging mechanism is out of scope of this paper, however for completeness sakes we have briefly described it's application here. Parikshan can complement statistical debugging in several ways to make it more effective, while at the same time isolating the instrumentation impact on the production container. Firstly, using dynamic instrumentation tools Parikshan can instrument as well as modify the predicates that are instrumented. Further the instrumentation can be increased or decreased dynamically, by taking the amount of buffer currently utilized. The buffer utilization is an indication of how much the debug-container lags behind the production container, and how long it would take for the buffer to overflow.

Record and Replay:

8. RELATED WORK

Record and Replay: Record and Replay systems [11, 21, 25, 24] have been an active area of research in the academic community for several years. These systems offer highly faithful re-execution in lieu of performance overhead. For instance ODR [11] reports 1.6x, and [19] reports 1.35x overhead, with much higher worst-case overheads. Parikshan avoids run-time overhead, but its cloning suspend time be viewed as an amortized cost in comparison to the overhead in record-replay systems.

Among record and replay systems, the work most closely related to ours is aftersight [19]. Aftersight records a production system, and replays it concurrently in a parallel VM. While both Aftersight and Parikshan allow debuggers an almost real-time diagnosis facility, aftersight suffers from recording overhead in the production VM . Additionally, it needs diagnosis VM to catch up with the production VM, which further slows down the application. On the other hand, in Parikshan we do not impact the production container, as we only duplicate the incoming traffic instead of recording. Furthermore, unlike replay based systems, Parikshan debug-containers can tolerate some amount of divergence from the original application: i.e., the debug container may continue to run even if the analysis slightly modifies it.

Large Scale System Debugging Another approach for production level bug diagnosis [15, 42, 41] is to use light weight instrumentation to capture traces like system calls, or windows event traces. These techniques use end-to-end trace event stitching to capture flows across multiple tiers to infer performance bugs. While the instrumentation for these tools have a low overhead, the corresponding granularity of the logs is also less. This limits the diagnosis capability of these tools, and they are only able to give a hint towards the bug root-cause rather than debug it completely.

Real-Time techniques In recent years, there have been approaches which are similar to Parikshan in applying real-time diagnosis on production systems. One of these approaches, Chaos Monkey [16] from Netflix which uses fault injection in real production systems to do fault tolerance testing. ChaosMonkey induces time-outs, resource hogs etc. to inject faults in a running system. This allows Netflix to test the robustness of their system at scale, and avoid large-scale system crashes. Another approach called AB Testing [22], probabilistically tests updates or beta releases on some percentage of users, while letting the majority of the application users work on the original system. AB Testing allows the developer to understand user-response to any new additions to the software, while most users get the same software. Unlike Parikshan, both these approaches are restricted to software testing, and directly impact the user.

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 *pre-copy* approach, which copies the memory state over several iterations, and then copies the process state. This includes hypervisors such as VMWare [37], Xen [20], and KVM [27]. VM Cloning, on the other hand, is usually done offline by taking a snapshot of a suspended/ shutdown VM, and restarting it in another machine. Cloning is helpful for scaling out applications, which use multiple instances of the same server. There has also been limited work towards live cloning. For example Sun et al. [40] use copy-on-write mechanism, to create a duplicate of the target VM without shutting it down. Similarly, another approach [23] uses live-cloning to do cluster-expansion of their system. However, unlike Parikshan, both these approaches, start a VM with a new network identity, and may require re-configuration of the duplicate node.

9. LIMITATIONS AND THREATS TO VALIDITY

The first key potential threat is **non-determinism**. Non-determinism can be caused due to three main reasons (1) system configuration, (2) application input, and (3) ordering in concurrent threads. Live cloning of the application state ensures that both applications are in the same "system-state" and have the same configuration parameters for itself and all dependencies in it's stack. Furthermore, in offline debugging it is often difficult to capture all possible inputs, and hence deal with input non-determinism. Parikshan's network proxy ensures that all inputs received in the production container are also forwarded to the debug-container. However, concurrency based non-determinism can still lead to different execution paths in the production and debug containers. While the current prototype version of Parikshan is not currently handling concurrency non-determinism within the system, we believe there are several existing techniques that can be applied to tackle this problem in the context of live debugging. Firstly, as long as the bug is statistically significant, it can still be caught by execution tracing mechanisms as shown in several previous approaches [**?**]. Furthermore, techniques like deterministic scheduling [**?**], can also be used to counter concurrency based non-deterministic bugs. Other techniques such as constraint solvers [**?**], can trace synchronization events in a program trace, and search through all possible execution orderings to trigger the concurrency error.

Another threat to Parikshan is that the error once detected, can be

10. DISCUSSION

Distributed Services: Large scale distributed systems are often comprised of several interacting services such as storage, NTP, backup service, controllers and resource managers. `Parikshan` can be used on one or more containers, and can be used to clone more than one communicating service. Based on the nature of the service, it may be (a). Cloned, (b). Turned off or (c). allowed without any modification. For example, storage services supporting a "debug-container" need to be cloned or turned off(depending on debugging environment) as they would propagate changes from the cloned container to the production containers. Similarly, services such as NTP service can be allowed, as they are publishsubscribe systems and the debug container cannot impact it in anyway. Furthermore, instrumentation inserted in the debug-container, will not necessarily slowdown all services. For instance, instrumentation in a MySQL query handler, will not slowdown file-sharing or NTP services running in the same container.

Consistency Requirements: Not all debugging may require very high consistency requirements, and not all overflows would mean that the clone is suddenly out of sync. In such cases the time-window can be arbitrarily long. For example in MySQL read transactions, even if there is a transaction failure or an overflow, it will not effect MySQL itself. Even with an overflow in some write operations, we could get a weakly consistent representation of the production system's execution.

11. CONCLUSION & FUTURE WORK

We presented `Parikshan` a framework to do live-debugging and analysis of large scale systems. `Parikshan` presents a novel technique to debug systems in real-time allowing for fast bug resolution. We show several case-studies to see how our system can be applied on real problems.

In the future, we wish to explore `Parikshan` further in 3 key areas: (1). Application: we aim to apply our system to real-time intrusion detection, and fault tolerance testing. (2). Analysis: we wish to define "real-time" data analysis techniques for traces and instrumentation done in `Parikshan`. We believe with streaming data analytic techniques, we can do much better than execution tracing/profiling. (3). Live Cloning: we plan to reduce the suspend time of "live cloning", by applying optimizations suggested in several recent works in live migration. To verify our implementation in a enterprise level public cloud provider, we also implemented a prototype of `Parikshan` on Google's Cloud Infrastructure (Google Compute [1]), we plan to use this prototype in future publications for scaled out testing.

An extended version of the paper is present in CUCS Tech Reports [12]. Kaiser is funded in part by NSF CCF-1302269 and CCF-1161079

12. REFERENCES

[1] Google Compute. https://cloud.google.com/compute/.
[2] Linux containers, userspace tools for the linux kernel containers. https://linuxcontainers.org.
[3] NAT: Network address translation. http://en.wikipedia.org/wiki/Network_address_translation.
[4] OpenVZ Linux Containers. http://www.openvz.org.
[5] Petstore a sample java platform, enterprise edition reference application. http://www.oracle.com/technetwork/java/petstore1-1-2-136742.html.
[6] Ploop: Containers in a file. http://openvz.org/Ploop.
[7] VisualVM. http://visualvm.java.net.
[8] vzctl: Toolset to perform various operations in OpenVZ. http://www.openvz.org/Man/vzctl.8.
[9] What is devops? http://www.radar.oreilly.com.
[10] Allspaw J., H. P. 10+ deploys per day: Dev and ops cooperation at flickr. OâĂŹReilly Velocity Web Performance and Operations Conference, 2009.
[11] Altekar, G., and Stoica, I. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 193–206.
[12] Arora, N., Ivancic, F., and Kaiser, G. Parikshan: Live debugging of production systems in isolation. Tech. rep., Aug 2015.
[13] Arora, N., Zhang, H., Rhee, J., Yoshihira, K., and Jiang, G. iProbe: A lightweight user-level dynamic instrumentation tool. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on* (2013), IEEE, pp. 742–745.
[14] Axboe, J. Fio-flexible io tester. *uRL: http://freecode.com/projects/fio* (2008).
[15] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for request extraction and workload modelling. In *OSDI* (2004), vol. 4, pp. 18–18.
[16] Bennett, C., and Tseitlin, A. Netflix: Chaos Monkey released into the wild. netflix tech blog, 2012.
[17] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices* (2006), vol. 41, ACM, pp. 169–190.
[18] Buck, B., and Hollingsworth, J. K. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*.
[19] Chow, J., Garfinkel, T., and Chen, P. M. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference* (2008), pp. 1–14.
[20] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), USENIX Association, pp. 273–286.
[21] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 211–224.
[22] Eisenberg, B., and Quarto-vonTivadar, J. *Always be testing: The complete guide to Google website optimizer*. John Wiley & Sons, 2009.
[23] Gebhart, A., and Bozak, E. Dynamic cluster expansion through virtualization-based live cloning, Sept. 10 2009. US Patent App. 12/044,888.
[24] Geels, D., Altekar, G., Maniatis, P., Roscoe, T., and Stoica, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007), vol. 7, pp. 285–298.
[25] Guo, Z., Wang, X., Tang, J., Liu, X., Xu, Z., Wu, M., Kaashoek, M. F., and Zhang, Z. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), USENIX Association, pp. 193–208.
[26] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu,

S. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 77–88.

[27] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[28] Kremenek, T., Ng, A. Y., and Engler, D. R. A factor graph model for software bug finding. In *IJCAI* (2007), pp. 2510–2516.

[29] Laadan, O., Viennot, N., and Nieh, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *ACM SIGMETRICS Performance Evaluation Review* (2010), vol. 38, ACM, pp. 155–166.

[30] Liu, C., Yan, X., Yu, H., Han, J., and Philip, S. Y. Mining behavior graphs for Backtrace of noncrashing bugs. In *SDM* (2005), SIAM, pp. 286–297.

[31] Lou, J.-G., Lin, Q., Ding, R., Fu, Q., Zhang, D., and Xie, T. Software analytics for incident management of online services: An experience report. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on* (2013), IEEE, pp. 475–485.

[32] Luk, C.-K. e. a. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05* (2005).

[33] McDougall, R., Mauro, J., and Gregg, B. *Solaris (TM) Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris (Solaris Series)*. Prentice Hall PTR, 2006.

[34] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (2014), 2.

[35] Mirkin, A., Kuznetsov, A., and Kolyshkin, K. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium* (2008), pp. 85–92.

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

[37] Nelson, M., Lim, B.-H., Hutchins, G., et al. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference, General Track* (2005), pp. 391–394.

[38] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07* (2007).

[39] Park, I., and Buch, R. Event tracing for windows: Best practices. In *Int. CMG Conference* (2004), pp. 565–574.

[40] Sun, Y., Luo, Y., Wang, X., Wang, Z., Zhang, B., Chen, H., and Li, X. Fast live cloning of virtual machine based on xen. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications* (Washington, DC, USA, 2009), HPCC '09, IEEE Computer Society, pp. 392–399.

[41] Tak, B.-C., Tang, C., Zhang, C., Govindan, S., Urgaonkar, B., and Chang, R. N. vPath: Precise discovery of request processing paths from black-box observations of thread and network activities. In *USENIX Annual technical conference* (2009).

[42] Zhang, H., Rhee, J., Arora, N., Gamage, S., Jiang, G., Yoshihira, K., and Xu, D. CLUE: System trace analytics for cloud service performance diagnosis. In *Network Operations and Management Symposium (NOMS), 2014 IEEE* (2014), IEEE, pp. 1–9.

[43] Zhang, S., and Ernst, M. D. Automated diagnosis of software configuration errors. In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 312–321.