DEFINED: Deterministic Execution for Interactive Control-Plane Debugging

Chia-Chi Lin[1] Virajith Jalaparti[1] Matthew Caesar[1] Jacobus Van der Merwe[2]

[1]*University of Illinois at Urbana-Champaign*
[2]*University of Utah*

Abstract

Large-scale networks are among the most complex software infrastructures in existence. Unfortunately, the extreme complexity of their basis, the control-plane software, leads to a rich variety of nondeterministic failure modes and anomalies. Research on debugging modern control-plane software has focused on designing comprehensive record and replay systems, but the large volumes of recordings often hinder the scalability of these designs. Here, we argue for a different approach. Namely, we take the position that *deterministic network execution* would vastly simplify the control-plane debugging process. This paper presents the design and implementation of *DEFINED*, a user-space substrate for *interactive debugging* that provides *deterministic execution* of networks in highly distributed and dynamic environments. We demonstrate our system's advantages by reproducing discovery of known *ordering* and *timing bugs* in popular software routing platforms, XORP and Quagga. Using Rocketfuel topologies and routing data from a Tier-1 backbone, we show *DEFINED* is practical and scalable for interactive fault diagnosis in large networks.

1 Introduction

Large-scale networks such as enterprise and ISP networks consist of a complex intertwining of systems and protocol implementations distributed over wide distances. At the basis of these networks lies the control-plane software that is responsible for controlling and managing data flows. Like other complex software systems, control-plane software is prone to defects or bugs introduced through human error. Indeed, studies have shown that control-plane traffic accounts for 95 – 99% of the observed bugs in networks [1].

One school of research has focused on applying *fully automated* techniques to analyze, test, and debug control-plane software. However, while automated techniques have been developed to localize memory faults [5] and avoid concurrency bugs [21], the larger class of *logical* or *semantic* errors seems to fundamentally require human knowledge to solve. To address this, most research has employed recording mechanisms to assist human troubleshooters in debugging large-scale control-plane software. Packet capture and replay tools such as `tcpdump` [30] record and replay packets at individual nodes. Friday [10] correlates recordings across distributed nodes to provide system-wide reproducibility for control-plane software. OFRewind [32] enables centrally controlled record and replay of control-plane software by leveraging the structure of OpenFlow controller domains. Unfortunately, while these schemes improve troubleshooter's ability to analyze control-plane software, the large volumes of recordings hinder the scalability of these systems. In fact, even the authors of Friday and OFRewind point out that a comprehensive recording of all events in an entire production network is infeasible.

Consequently, troubleshooters often enable only partial recordings in production networks, e.g., logging only packet headers or logging only at specific network locations. However, solutions based on partial recordings can fail to reproduce bugs triggered by nondeterministic behaviors, e.g., message orderings or unsynchronized clocks. Troubleshooting these nondeterministic bugs is challenging. Since a bug may happen only when certain messages arrive at a specific node in a specific ordering, if troubleshooters didn't select the node to record messages beforehand, it is difficult to reproduce the bug. Two types of nondeterministic bugs are particularly notorious in control-plane software: *ordering bugs* that appear only when certain messages occur in specific orderings and *timing bugs* that appear only when certain messages are processed at specific timings.

To address these nondeterministic bugs, in this paper, we present a new system, *DEFINED*, a debugger that allows a troubleshooter to analyze control-plane bugs after detecting erroneous behaviors of a system. *DEFINED* simplifies interactive control-plane software debugging through deterministic network execution. Namely, given the same set of external events (e.g., messages from external routers, failures of links and routers), we make every node in the network always receive messages in a deterministic ordering and timing. Accordingly, with *DEFINED*, troubleshooters can adopt partial recordings and still be able to reproduce nondeterministic bugs.

To enable deterministic network execution, *DEFINED* eliminates all sources of nondeterministic *internal* events in a network, and relies on partial recordings to record and replay nondeterministic *external* events. Specifically, *DEFINED* ensures each node receives messages and fires local timers in a deterministic fashion. To provide more debugging functionality without introducing prohibitive overheads to control-plane software, *DEFINED* consists of two components: *DEFINED-RB* (*RB*: RollBack) that instruments production networks, and *DEFINED-LS* (*LS*: LockStep) that manages debugging networks.

DEFINED-RB introduces minimum overheads to control-plane software with an "optimistic" approach: each node independently decides on a pseudorandom sequence of events, and then lets the network execute in an arbitrary fashion. If the order in which events execute is different from the pseudorandom sequence, the network is "rolled back" to an earlier state, and played forward with the correct ordering. We introduce a novel pseudorandom sequence to reduce to the number of rollbacks, and hence minimize the overheads.

DEFINED-LS provides an interactive stepping functionality to troubleshooters in a debugging network. It allows troubleshooters to investigate and manipulate state, and slowly step through the operation of individual messages. *DEFINED-LS* does so by forcing debugging networks to execute in a *lockstep* fashion. To reproduce nondeterministic bugs, *DEFINED* guarantees that *DEFINED-LS* deterministically reproduces *DEFINED-RB*'s execution.

Deterministic execution [4, 6, 12, 27] and interactive stepping [11, 13] have been widely applied in non-control-plane software to ease interactive debugging. These techniques, however, to our best knowledge, cannot operate efficiently and effectively with control-plane software. Our system, *DEFINED*, is a debugger for control-plane software that address the problem of interactive debugging in modern wide-area networks. To demonstrate the utility of *DEFINED* in assisting troubleshooters analyzing ordering and timing bugs, we use *DEFINED* to reproduce the discovery of known bugs in two popular open-source control-plane implementations, XORP and Quagga. Our evaluation on Emulab [31] with Rocketfuel topologies [28] and Tier-1 ISP traces shows that very little overhead is required to make production networks deterministic and that performance in debugging networks has sufficiently low response time for interactive use.

2 System Design

In this section, we describe the details of the design of *DEFINED*. We first give an overview of the system (Section 2.1). Then, we describe *DEFINED-RB* (Section 2.2), which instruments a production network to make its execution deterministic. We next show *DEFINED-LS* (Section 2.3), which allows a debugging network to be "stepped" through in a manner controlled by a human troubleshooter. We then conclude the section by discussing some properties and limitations of our design (Section 2.4 and Section 2.5). To keep the description concise, in this section, we focus on how our system ensures deterministic message events, and in Section 3, we will describe how *DEFINED* can be extended to provide deterministic timer events.

2.1 Interactive Network Debugging

We first clarify the benefits of a tool for interactive control-plane software debugging. Under our design, control-plane software runs on top of *DEFINED*, a user-space substrate, instead of directly on an operating system. Complementing existing log-based systems [10, 30, 32] that passively record software activities, we instrument control-plane software in a production network and actively manipulate ordering and timing of internal message receptions. The manipulation ensures network-wide execution is deterministic. When human troubleshooters observe any control-plane software bug in a production network, they can reproduce the bug deterministically in a debugging network with only partial recordings, and analyze it through the *debugging coordinator* with the interactive stepping functionality.

Our design consists of two key components, *DEFINED-RB*, which makes control-plane execution deterministic by *masking internal nondeterminism* and *DEFINED-LS*, which introduces distributed lockstep execution for interactive debugging.

***DEFINED-RB*:** Debugging a control-plane system becomes much easier if the operation of that system is deterministic. Unfortunately, existing control-plane software incorporates a high degree of internal randomness in its execution, arising from varying message orderings, delay and jitter, and other variables arising from distributed execution. To address this, our design manipulates the operation of a production network itself, to remove all internal nondeterminism and cause it to run in a deterministic manner.

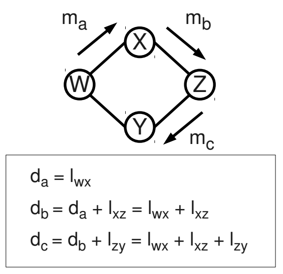
Each node intercepts message and timer events before delivering them to the control-plane software, and then uses a pseudorandom ordering function to determine the exact orderings and timings at which to send the events up to the software. Instead of adopting a stop-and-wait design [12], we employ speculative execution to reduce overheads: upon each event occurrence, a node uses its pseudorandom ordering function to check whether the order in which the events appeared so far follows the computed pseudorandom sequence. If the order is the same as the pseudorandom sequence, the node delivers the event to the control-plane software. On the other hand, if the order is different from the pseudorandom sequence, the network is "rolled back" to an earlier state, and played forward with the correct ordering. To further optimize the performance, we construct the pseudorandom ordering function according to the network topology, so that the computed pseudorandom sequence matches the event sequence that most frequently occurs. Consequently, the number of rollbacks is minimized.

***DEFINED-LS*:** The tremendous load, scale, and rates of change of modern networks make it hard for a human troubleshooter to build an understanding of the entire control-plane system's state. Debugging such a system becomes much easier if the troubleshooter has time to investigate and manipulate state, and slowly step through the operation of individual messages.

To achieve such interactive stepping of software in wide-area networks, a runtime coordinator manages execution across the distributed set of processes making up the control-plane software. This is done by logically dividing the software's execution into a series of *steps*. These steps may be chosen at various levels of granularity (per-event or per-path-change). The coordinator then runs the software in virtual time, executing it in a "lockstep" fashion across nodes (alternating between event-sending and event-processing phases). Distributed user-space substrates replay events to their local software and collect outbound events to be sent in the next cycle. Nodes use a distributed semaphore to coordinate. This approach controls execution, and deterministically reproduces the software's behavior by adopting the exact pseudorandom ordering function used by the production network.

2.2 Interfacing with Production Networks

To remove internal nondeterminism from a production network, *DEFINED-RB* uses speculative execution to ensure determinism while not significantly slowing down network execution. It does this by speculatively letting messages be sent to the software in the order in which they are received. To make sure the ordering can be reproduced, nodes in the network locally compute a pseudorandom ordering over these messages, and if messages do not arrive in the computed pseudorandom order, the node is *rolled back* to the point at which the first message arrived out of sequence, and messages are then played back in the correct order. Rolling back slows down processing, but with appropriate selection of the pseudorandom sequence, we can make rolling back rare. In particular, we design an optimized pseudorandom sequence to match the common-case ordering of events we would expect to see in the production network. Overall, we need to solve two problems: (i) we need to come up with a pseudorandom ordering that matches the common-case ordering of events; (ii) we need to perform the rollback when the predicted ordering is violated.



m_a
m_b
m_c
X
W
Z
Y
$d_a = l_{wx}$
$d_b = d_a + l_{xz} = l_{wx} + l_{xz}$
$d_c = d_b + l_{zy} = l_{wx} + l_{xz} + l_{zy}$

Figure 1: Example: calculating d_i.

Computing a message ordering: There are many ways to compute the pseudorandom ordering, for example using straightforward hashing and permutation. However, to ensure correctness, the pseudorandom ordering needs to maintain causal relationships between messages. In addition, every time the pseudorandom ordering diverges from the production network operation, *DEFINED-RB* requires a rollback. Hence, for efficiency reasons, we would like a pseudorandom ordering function that minimizes the number of rollbacks that are needed.

To do this, we construct an ordering function that reflects the expected ordering of message arrivals. The function takes as input a set of messages $\{m_1, m_2, \ldots, m_k\}$ received at a node n, where each message m_i is annotated with (i) n_i, the identifier of the *originating node* that generated the first message of the causal chain; (ii) s_i, a strictly increasing *sequence number* assigned by the originating node; (iii) d_i, a deterministic estimate of the delay from the originating node n_i to the local node n.

To clarify the meaning of each field, Figure 1 illustrates how *DEFINED-RB* calculates n_i, s_i, and d_i for three causally related messages. For each link (n_i, n_j), *DEFINED-RB* measures the average link delay l_{ij} before launching the control-plane software. When a node generates a message due to external events (e.g., a withdraw message when a link goes down), it is called the *originating node* of the message. The node annotates the message with n_i equal to its id, s_i equal to the current value of a strictly increasing counter, and d_i equal to the average link delay of the outgoing link. On the other hand, assume a node generates a message m_i due to another internal message m_j (e.g., a route update when receiving a message from another node in the system), where m_j is annotated with n_j, s_j, and d_j. The node annotates message m_i with n_i equal to n_j, s_i equal to s_j, and d_i equal to d_j plus the average link delay of the outgoing link. In the figure, we assume m_a is generated due to external events, while m_b is generated due to m_a, and m_c is generated due to m_b. Then, all messages have the same originating node W and sequence number. In addition, d_a equal to l_{wx}, and d_b and d_c are calculated by increasing d_a and d_b by l_{xz} and l_{zy}, respectively.[1]

[1]We use d_i to retain causal relationships by never rolling back mes-

When receiving messages from others, a node uses the ordering function to first sort the messages by d_i values. It then sorts messages with identical d_i values by their n_i values, and messages with identical n_i values with their s_i values. We sort messages by d_i before s_i, since for control-plane software, messages originating from a node can take different paths. The resulting function has three key properties: (i) it is *deterministic*, as it will always compute the same outputs given the same external events; (ii) it is *consistent*, as it retains causal relationships between messages; (iii) it is closely matched to the *common-case ordering* when originating nodes send out messages at roughly the same time, since d_i indicates the average arrival time of a message.

To further avoid long chains of rollbacks, *DEFINED-RB* makes sure the ordering function is applied independently to messages originated at roughly the same time. To do this, we divide time into distinct steps, group external events appearing in a single timestep together, and then independently impose the ordering function mentioned above on the messages corresponding to each timestep. We further bound the length of each causal chain within a timestep: messages over this bound are assigned to the next timestep. All messages in a single timestep correspond to a single group. Each group is associated with a distinct *group number*. One node is selected to periodically broadcast special packets called *beacons* which specify the group numbers to be used by the rest of the nodes in the network. (Leader election algorithms [22] are used to make sure the system can tolerate failures.) Group numbers are strictly increasing. Messages triggered by an external event are tagged with the current group number, while output messages generated due to an internal message are assigned the same group number as the internal message.[2]

In our implementation, we assign fixed values of d_i based on average link delay between nodes rather than dynamically estimating it. However, if desired, the link delay values may be periodically re-estimated, as long as they are applied and recorded at group boundaries.

Detecting if a rollback is necessary: Each node maintains a sliding window history of messages it received since the last group number update, as well as a list of messages it sent since the last group number update. The history is sorted by the ordering function. An entry in the history can be removed after all messages that might be ordered before it have arrived. Depending on the node

sages with lesser d_i values. Therefore, to retain causal relationships for a message with multiple causal parents, we only need to record the largest d_i value among all its parents.

[2]A large variation in distances from nodes to the beacon source may cause unnecessary rollbacks. We can address this by dividing the network into smaller subnetworks, and applying *DEFINED-RB* to each subnetwork independently.

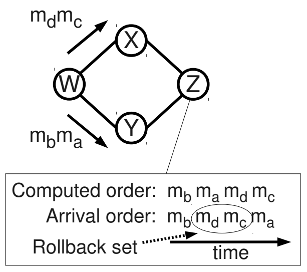

$m_d m_c$
X
W
Z
Y
$m_b m_a$
Computed order: $m_b\ m_a\ m_d\ m_c$
Arrival order: $m_b\ m_d\ m_c\ m_a$
Rollback set
time

Figure 2: Example: detecting and performing rollback. In this example, we assume all messages originate from node W, and all links have the same expected delay. Thus, the order of the messages are determined by the sequence numbers. We assume messages m_b, m_a, m_d, and m_c have sequence numbers 0, 1, 2, and 3, respectively.

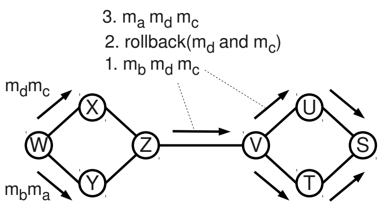


Figure 3: Example: rolling back across nodes.

performance, our experience shows that in a modern production network, we can generally remove an entry after two times the maximum propagation time across the network.[3] Whenever a node receives a new message, it passes the contents of this window to the ordering function to determine if the new message has arrived in the correct pseudorandom sequence.

If the message is received in the correct order, it is sent to the software. Otherwise, the node must roll back the node's state to the first point where the sequences diverge and replay received messages in the correct order. There are two scenarios in which a node needs to roll back its state: (i) when receiving messages that have earlier group numbers; (ii) when receiving messages that have the current group number, but don't arrive in the correct pseudorandom sequence. For example in Figure 2, if node Z receives messages in the order $\{m_b, m_d, m_c, m_a\}$ (message m_b received first), but upon receiving m_a it computes the sequence $\{m_b, m_a, m_d, m_c\}$, it would need to roll back to the point just before it received message m_d (i.e., it would need to roll back messages m_d and m_c). Note the pseudorandom ordering is computed on every message arrival, for example, here, the node would compute $\{m_b\}$ after receiving message m_b, then compute $\{m_b, m_d\}$ after receiving message m_d, then compute $\{m_b, m_d, m_c\}$ after receiving message m_c, but then compute the final sequence $\{m_b, m_a, m_d, m_c\}$ after receiving m_a.

[3]We only need an upper bound of the maximum propagation time. In our implementation, we estimate this bound with the sum of the average propagation time and four times its standard deviation.

Performing the rollback: Finally, performing a rollback at a node may require "unsending" messages previously sent by it to its neighbors. To do this, the node keeps a history of previous messages sent within the last few group intervals. On rolling back, the node informs neighbors of the range of messages that should be rolled back. In the example in Figure 3, node Z had previously sent node V messages $\{m_b, m_d, m_c\}$. On performing the rollback to before receiving m_d, node Z tells node V to roll back the messages m_d and m_c, and then sends messages $\{m_a, m_d, m_c\}$ in the correct order. This process continues downstream: since node V had previously forwarded messages $\{m_d, m_c\}$, it must instruct node U and node T to roll them back as well. Messages at node S are rolled back in a similar manner.

To roll back misordered messages, we restore the state of the control-plane software, and if required, inform neighbors about such a rollback to ensure that all messages that are *causally related* to the rolled back messages are themselves rolled back.

2.3 Stepping through Debugging Networks

In a network instance created to support network debugging, mechanistic delays and overheads are not significant concerns. Therefore, in this environment, we introduce *DEFINED-LS* which allows interactive stepping by forcing the network to execute in a lockstep fashion. This is done by explicitly queuing messages and timer events received by a node, and playing them at coordinated intervals using a predetermined *ordering function* that is exactly the same as that used in the production network (Section 2.2) to ensure determinism. To make the network run in lockstep, our system instructs each node to cycle through two phases: a *transmission* phase and a *processing* phase. The system coordinates all nodes with a mechanism similar to a distributed semaphore to make sure they are in the same phase at the same time. To ensure determinism, *DEFINED-LS* replays partially logged external events according to the group numbers they received in the production network. In a debugging network, one group of events is replayed at a time. When all messages are output, the next group is replayed. The nodes use TCP for communication in order to ensure that messages are not lost, which is necessary for determinism.[4]

Transmission phase: In this phase, each node transmits all messages generated in the previous processing phase. That is, the node sends out messages in a *send buffer* (filled in the processing phase) and stores all messages received in a *receive buffer*. *DEFINED-LS* then uses the same ordering function used in the production network over the received messages to compute the order in which the messages are to be delivered to the application. Hence, the messages in the receive buffer are sorted in the same way as they are received in the production network, thereby ensuring the same ordering of events. This results in the debugging network reproducing the execution of the production network. To indicate readiness to transition to the processing phase, a node sends a marker packet when it has no further messages to send.

Processing phase: In this phase, each node processes all messages received during the previous transmission phase. In particular, the node sends all messages in the receive buffer up to the control-plane software, and enqueues the software's generated messages into the send buffer. The node moves to the transmission phase after the control-plane software processes all messages in the receive buffer.

2.4 System Properties

DEFINED has two provable properties: (i) *DEFINED-LS* exactly reproduces the execution of the production network instrumented by *DEFINED-RB*; (ii) even with the presence of cascading rollbacks (i.e., rollbacks across nodes), *DEFINED-RB* eventually terminates. The first property is the core of our system, as it provides reproducibility of network execution. The second property guarantees there will be no deadlocks when a production network is instrumented by *DEFINED-RB*. We present proofs of these two properties in a technical report [18].

2.5 Limitations

Supporting incremental deployment: *DEFINED* assumes control over all devices that need to be debugged. For example, when using our design to debug an Open Shortest Path First (OSPF) network, all OSPF-speaking routers should be instrumented with *DEFINED*. This may pose a challenge in environments in which the network operator can only instrument subsets the network, or needs to interface with adjacent networks not under the operator's control. Similar issues can also occur between the interactions of control plane and data plane, e.g., external rollbacks might be required when the control-plane software attempts to modify the data-plane forwarding table. To deal with these situations, *DEFINED* records inputs at interfaces with external systems. Our system can then replay these partial recordings at a later point in time to reproduce execution. In addition, we can avoid external rollbacks by employing buffers at border nodes as proposed in earlier work [14].

Inferring causality in closed-source software: Another assumption of our design is that the source code of the software is available, as our design requires the ability to infer causal relationships between incoming mes-

[4]Alternatively, we can also record these message-loss events, and replay them in the debugging network.

sages and outgoing ones. Despite this assumption, *DEFINED* is still highly useful for control-plane software developers as we will demonstrate with case studies in Section 4. In fact, it took a graduate student only one day to instrument the control-plane software in these case studies. In addition, if developers are willing to incorporate *DEFINED* in their software, their customers can still experience the benefits of deterministic network execution even when the shipped software is closed-source. Moreover, work on tracing information flow through application binaries [7] can help enable our design directly on closed-source software.

Imposing determinism on a single node: To provide deterministic network execution, *DEFINED* also needs to eliminate internal nondeterminism triggered by events on a local node (e.g., thread scheduling, memory reordering). In Section 4, we describe a specific implementation that removes internal nondeterminism triggered by local events from XORP and Quagga. Fortunately, existing works [2–4] provide more general solutions to this problem, and *DEFINED* can be combined with these works to ensure determinism of general control-plane software.

3 Implementation

To simplify deployment and operate with existing software bases, we implement *DEFINED* as a user-space "shim layer" in the form of a library consisting of function wrappers to intercept message sending, message receiving, and timer calls.

Our implementation addresses three key challenges:

Providing interfaces to mark causal relationships: As discussed in Section 2.2, *DEFINED-RB* must determine which messages *sent* by the control-plane software are causally related to messages that are *received* by the software. This information is used to determine which messages need to be "unsent" when the pseudorandom sequence is violated. Our implementation overcomes this challenge by providing interfaces for developers to tag a message with a unique identifier when it is originated, and extract the identifier from the message when it is received. With our design in Section 2.2, developers only need to mark "immediate" causal relationships between messages (causal relationships of messages that are triggered by the same external event). Then, *DEFINED* will use these immediate relationships to generate the correct annotated fields and sort messages in the correct order. When instrumenting XORP and Quagga, we track all immediate causal relationships by passing the identifier of an incoming message from message receiving functions, to message processing functions, and finally, to message sending functions. This is done by instrumenting these message related functions in the application software with an extra parameter.

Rolling back: After obtaining a pseudorandom ordering, a node needs to rollback the state of the software when the ordering is violated. To do this, nodes perform three steps: (i) check-point states between message receipts, (ii) restore a particular state, and (iii) play back messages in the given pseudorandom ordering.

To accomplish these steps, *DEFINED* employs the `fork()` system call. When a message is received, the node inserts the message into the history as described in Section 2.2 and, at the same time, checks if the pseudorandom ordering is violated. If the message arrival complies with the ordering, the node invokes the `fork()` system call. Then, a piece of shared memory is established between the parent and child processes for notifications of possible rollbacks. If the received message violates the ordering, the node uses the shared memory to instruct the process ID it wishes to roll back to. As discussed in previous literature [24], a normal `fork()` is not sufficient to ensure determinism. Specifically, *DEFINED* also saves the state of any open files and pending signals, and manipulates process and thread IDs. After restoring its state, the node plays back the received messages according to the pseudorandom ordering.

While using `fork()` may seem somewhat heavyweight, we found its overhead to be low enough in our implementation that pursuing other techniques did not seem necessary for common environments (modern OSs use copy-on-write to reduce overheads). If desired, the overhead of rollback may be reduced further, for example by only calling `fork()` for every several messages, and rolling back to the last `fork()` before the sequences diverge, or by using standard application-specific checkpointing techniques (we investigate some optimizations in Section 5).

Dealing with timers: The mechanisms described above are sufficient to reproduce message events. However, to reproduce timer events in our design, we need to ensure the rate at which the process perceives time as progressing is the same, every time the system is run. To do this we run control-plane software in *virtual time*: instead of triggering timers with the system clock, we make timers expire according to a counter that advances deterministically with respect to the message events. This enables timer events to be reproduced in our system. However, we would like to ensure that we do not substantially change behavior of the protocol when doing this. For example, consider the flap damping algorithm in Border Gateway Protocol (BGP) [23], which "holds down" unstable routes for a certain period of time. When we run flap damping in virtual time, we would like BGP to hold down routes for a similar amount of time, to avoid making the network less or more stable. To achieve this, we use a virtual time that is deterministically reproducible, yet progresses at a rate similar to "real" wallclock time. We do this by using a deterministic counter for virtual time that is advanced on receipt of every beacon message, with the rate of advancement between beacons equal to the configured beacon inter-arrival time. In our implementation, we broadcast one beacon message every 250 ms, corresponding to one unit of virtual time.

4 Case Studies

To demonstrate the practicality of *DEFINED*, we instrument the BGP module in XORP 0.4 and the Routing Information Protocol (RIP) module in Quagga 0.96.5 with our system. We use *DEFINED* to reproduce discovery of two known bugs in these control-plane implementations: an ordering bug in the BGP path selection process and a timing bug in the RIP timer refresh procedure. These case studies demonstrate how an operator might utilize *DEFINED* to troubleshoot control-plane bugs after observing erroneous behaviors. We then conclude this section with a discussion of our experience.

Ordering bug in XORP BGP path selection: A BGP module should select the best path among all paths it receives from its peers. To do so, it checks all valid paths against a list of rules. There are dozens of rules in the BGP path selection process, but to understand the XORP bug, we need to know only three of them. First, the selection process compares the AS path length of each path, and those with shortest AS path length are selected as preferable paths. Then, these preferable paths are grouped by the neighboring AS of each path. Within each group, paths with lowest multi-exit discriminator (MED) are selected. All selected paths are checked against the last rule that compares the interior gateway protocol (IGP) distance of the paths. Finally, the path with the lowest IGP distance is selected as the best path. One peculiar aspect of this process is the MED rule. Because the rule checks the MED attribute only within a group of paths that have the same neighboring AS, it creates a non-transitive ordering among paths. For example, as illustrated in Figure 4, an AS with three routers $R1$, $R2$, and $R3$ peers with another two ASs at external routers $ER1$, $ER2$, and $ER3$. These external routers advertise three paths p_1, p_2, and p_3. Through neighboring routers $R1$ and $R2$, these paths eventually arrive at $R3$. All three paths have the same AS path length, while p_1 and p_2 have the same neighboring AS. In addition, p_1 has a MED attribute of 10, p_2 has 5, and p_3 has 20. Finally, p_1 has an IGP distance of 10, p_2 has 30, and p_3 has 20. Under these settings, when $R3$ considers only a pair of paths each time, p_2 wins over p_1, p_3 wins over p_2, but p_1 wins over p_3. Thus, to avoid choosing a less preferable path, a BGP module on $R3$ should compare all valid paths whenever the process is executed and select p_3 as the best path.

Version 0.4 of the XORP BGP module, however, makes a mistake here. When receiving an incoming path, it only compares the path with the current best path. As a result, the outcome of the selection process implementation can differ across executions: if the ordering of incoming paths at $R3$ is p_1, p_2, and p_3, then p_3 is selected as the best path; unfortunately, if the ordering of the incoming path is p_1, p_3, and p_2, then p_2 is incorrectly selected as the best path.

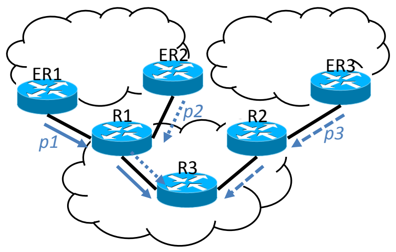


Figure 4: An illustration of a known bug in the BGP module of XORP 0.4.

Using only partial recordings on border routers and *gdb* to troubleshoot this bug, an operator enables logging for both external and internal network nondeterminism at $R1$ and $R2$ in Figure 4. When the bug is triggered, the operator replays log contents to reproduce the bug within a debugging network. However, because internal nondeterminism is recorded only at border routers, the set of paths can still reach router $R3$ in a nondeterministic fashion. The operator faces complications when experimenting with execution in the debugging network due to the inability to mirror behavior of the production network.

To address this, we use *DEFINED* to troubleshoot this XORP bug. We first use six machines to emulate the network depicted in Figure 4 and load them with the version of XORP containing the bug. We intercept nondeterministic system calls from XORP to remove internal nondeterminism triggered by local events. We then run the production network until the bug occurs. During the process, we are only required to enable partial recordings of external events at $R1$ and $R2$ but not recordings of internal events. Upon identifying the bug, we then activate *DEFINED-LS* in the debugging network. Since our system ensures that execution of both these networks match precisely, when we replay the logged external events and run the debugging network, the bug immediately occurs. We then use *DEFINED-LS*'s stepping functionality, to find the exact point at which XORP begins behaving incorrectly. After understanding the bug, we implement a patch for XORP and validate it in the debugging network. Finally, we install the patch in the production network. Deterministic execution again guarantees that all workarounds we create in the debugging network will behave the same way in the production network.

Timing bug in Quagga RIP timer refresh: To handle network dynamics, RIP maintains a timer for each route

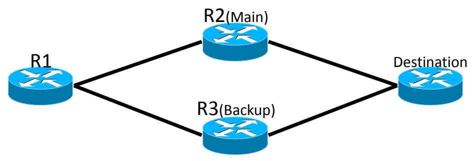


Figure 5: An illustration of the known timing bug in the RIP module of Quagga 0.96.5.

in its routing table. When receiving route announcements, if the route is already in the routing table, RIP updates the timer for the route. When a timer expires, RIP removes the route from the routing table. This mechanism ensures that the routing table contains only valid paths.

One subtle point of this process is that when comparing a route announcement with a route in the routing table, RIP must check both the destination field as well as the next-hop field. The Quagga RIP module, however, makes a mistake by only considering the destination field. As a result, the implementation contains a timing bug that triggers a black hole when certain route announcements are received at particular timings.

As illustrated in Figure 5, a router R_1 connects to two other routers R_2 and R_3. Both R_2 and R_3 provide R_1 routes to the same destination, and R_2 serves as the main router, while R_3 is the backup. All routers are running RIP, so what should happen is that R_1 maintains the route through R_2 in its routing table and refreshes the timer only when receiving announcements from R_2. When R_2 goes down, due to the lack of periodic announcement, the timer for the route will eventually time out. Then, R_1 will remove the route through R_2 from its routing table and pick up the route through R_3.

However, because the RIP implementation in Quagga 0.96.5 checks only the destination field when comparing announcements with routes in the routing table, R_1 will refresh the timer for the route through R_2 when receiving announcements from not only R_2 but also R_3. In this case, when R_2 goes down, two scenarios can happen. If announcements from R_3 reach R_1 after the route through R_2 times out, then R_1 correctly picks up the new route through R_3. Unfortunately, if announcements from R_3 reach R_1 before the route through R_2 times out, then R_1 will incorrectly refreshes the route through R_2 in the routing table. Even worse, the periodic announcements from R_3 will keep the invalid route through R_2 in the routing table and effectively create a black hole.

Using partial recordings of only message events and *gdb* to troubleshoot this bug can take a lot of time and resources, due to nondeterministic timer events embedded in control-plane software. For example, when using *gdb*, a human troubleshooter will experience timers going off unexpectedly while stepping through one instance of the Quagga RIP module on one of the routers. Moreover, to be able to reproduce the bug, it is also challenging for the human troubleshooter to manually coordinate the timing of message receipt and timer expirations.

We use four machines to emulate the network in Figure 5 and load them with the version of Quagga containing the bug. Fortunately, the same approach we used to troubleshoot the BGP path selection bug can address the RIP timer refresh bug, since timing events were also triggered deterministically in networks instrumented by *DEFINED*. As a result, during the debugging process, timers will not go off unexpectedly even when we step through the network execution at different paces.

Discussion: As shown in these case studies, *DEFINED* actively manipulates the ordering and timing of internal network events, and it makes control-plane software easier to test and debug. Another property that comes with the active manipulation, however, is that some network execution paths will never occur, and hence, some bugs will never appear in an instrumented network. For example, as we were debugging the XORP bug, we noticed that if the ordering function in *DEFINED* sorted the paths in the order of p_1, p_2, and p_3, the bug would not happen in the production network nor in the debugging network. This property, though, still protects instrumented networks from the bug, since the deterministic network execution guarantees that the bug will never appear.[5] Nevertheless, a troubleshooter may choose to not instrument the production network with *DEFINED-RB*, but to still leverage the interactive stepping functionality of *DEFINED-LS*. Fortunately, we can apply different ordering functions in *DEFINED-LS*, and then we will be able to examine all possible execution paths in the debugging network.

5 Evaluation

While *DEFINED* simplifies the task of control-plane debugging, it comes with several costs. In order to measure this overhead, we leverage Emulab [31] and take a two-pronged approach. First, to evaluate the performance of *DEFINED* in a practical setting, we perform experiments using topologies from Rocketfuel [28] and traces from a Tier-1 ISP (Section 5.2). Then, to study scalability of our system, we present results under a wide range of topologies and workloads (Section 5.3).

5.1 Methodology

We first give an overview of our experimental approach:

Topologies and traces: To improve the realism of our evaluation, we leverage topologies measured with Rocketfuel and OSPF traces collected at a Tier-1 ISP network.

[5]On the other hand, it is possible that *DEFINED* avoids some network execution paths with particular performance characteristics. In this case, an operator can modify the ordering function to force such paths to occur, potentially trading performance for more rollbacks.

We use PoP-level topologies from Rocketfuel including Sprintlink (43 nodes), Ebone (25 nodes), and Level3 (52 nodes). (Results from these topologies are similar, so we only present Sprintlink results due to space constraints.) Then, the OSPF traces are collected from a Tier-1 ISP area 0 network consisting of 324 nodes during a 2 week period (November 1st to 14th, 2009), resulting in 651 OSPF network events. We post-process these traces to reproduce the network dynamics over time, and then replay this workload in our experiments by randomly mapping events onto Rocketfuel topologies. Finally, to investigate the performance of *DEFINED* at scale, and over a wider range of topologies, we consider synthetic graphs constructed by the BRITE topology generator. Overall, we focus most of our experiments on intra-domain routing as opposed to inter-domain routing, as the lower propagation delays and tighter requirements on fast reaction make our overheads more visible. Unless otherwise specified, we run our implementation with the XORP OSPF router daemon, version 1.6.

Metrics: Since *DEFINED-RB* and *DEFINED-LS* have different purposes, we are concerned with different "success metrics" for each. As *DEFINED-RB* instruments a production network, we measure its *control*, *delay*, and *memory* overheads. On the other hand, as *DEFINED-LS* is designed for use in a debugging network for interactive stepping, overheads are of less concern (though retain some importance). Hence, we measure its response time for user-driven commands (e.g., a step command).

5.2 Performance

To characterize the performance overheads of *DEFINED*, we replay network traces against our implementation deployed on Emulab. We evaluate the design on several scales. First, we collect *network-level* results on our implementation in the Rocketfuel Sprintlink topology. We then gather *node-level* microbenchmarks to uncover the sources of bottlenecks in our implementation.

Network-wide experiments: First, we replay the Tier-1 ISP workload against our XORP-based implementation and measure the control overhead per node, for each event in the trace. Figure 6a shows that a small number of nodes experience more control overhead than others, as the rollback procedure requires additional control packets to be exchanged between nodes. Fortunately, in all cases, the percentage of these nodes is less than 1%.

Then, we measure the time for the network to converge (the time from when a failure is detected, to when all nodes are updated with their correct routing state). To stress our design, we reduce XORP's *hello* and *retransmit* intervals to be as small as possible (1 second). We compare against an unmodified XORP implementation. We found no statistically significant difference between the two. However, to improve stability, XORP's default OSPF configuration introduces a 1-second delay between when routing messages are received and when they are propagated on (due to the retransmit timer). To investigate whether this delay was the reason why our performance overheads could not be seen, we modified the XORP code to eliminate this 1-second delay. After doing this, the delays became more apparent: Figure 6b shows the network-wide convergence time is still close between the two, but our implementation has additional delay in a small number of cases, resulting in a longer tail. In addition, this figure also demonstrates that our technique in imposing local determinism on control-plane software (Section 4) has negligible overhead.

Finally, we measure the response time of *DEFINED-LS*, as it is designed to support *interactive* debugging and should respond quickly to commands from the human troubleshooter. Figure 6c shows the cumulative distribution function of the response time to execute a single *step* command of *DEFINED-LS* (where a single step is measured as the time to complete a transmission phase and a processing phase as described in Section 2.3). In this scenario every step requires less than a second.

Single-node experiments: To investigate the source of the tail, we instrumented a single node of our implementation to collect microbenchmarks. We compare XORP running under *DEFINED-RB* with an unmodified instance of XORP. In particular, we measure the amount of time required to perform a rollback (Figure 7a), as well as the time required to process packets without rollbacks (Figure 7b). We found that rollback code was triggered rarely, but, as expected, when it was triggered, it introduced overhead. To reduce the rollback overhead, instead of using `fork()` calls as described in Section 3 (FK), we manually intercepted memory writes (MI) using */proc/<pid>/mem* to directly access the memory of the process to emulate application-specific memory management, and measured the overhead to only copy changed bytes between the processes.[6] With this optimization, the median overhead for rollback is reduced to around 0.6 ms (Figure 7a), making non-rollback overhead the bottleneck. The variance observed in this figure comes from the variance of `fork()` calls and the number of events to be rolled back. Note that even the unoptimized implementation of rolling back may be tolerable for certain protocols, e.g., BGP uses a timer to intentionally slow convergence for scalability purposes.[7]

To reduce the non-rollback overhead, we investigate two optimizations. First, we try *pre-forking* (PF): instead of performing the fork when the new packet arrives (TF),

[6]We use this optimization to identify the optimal bound of rollbacks. It is not necessary for a system to do so to adopt *DEFINED*.

[7]The MRAI timer determines the minimum time between advertisements of routes to a particular destination from a single BGP device.

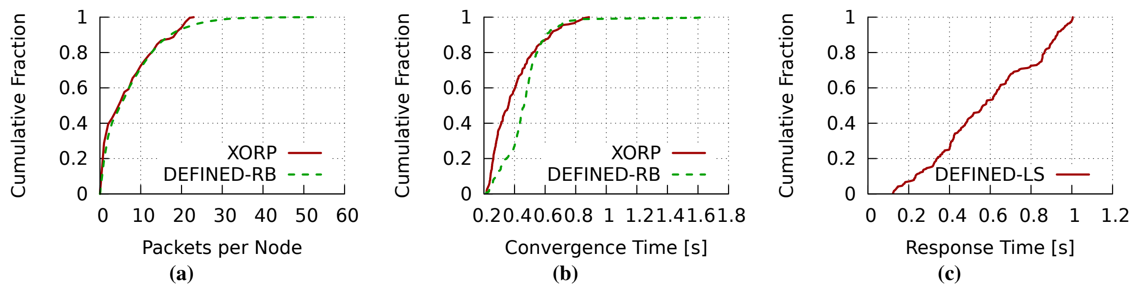


Figure 6: Network-level results of Sprintlink topology with Tier-1 OSPF traces: (a) control message overheads of *DEFINED-RB*; (b) delay of *DEFINED-RB*; (c) response time of *DEFINED-LS*.

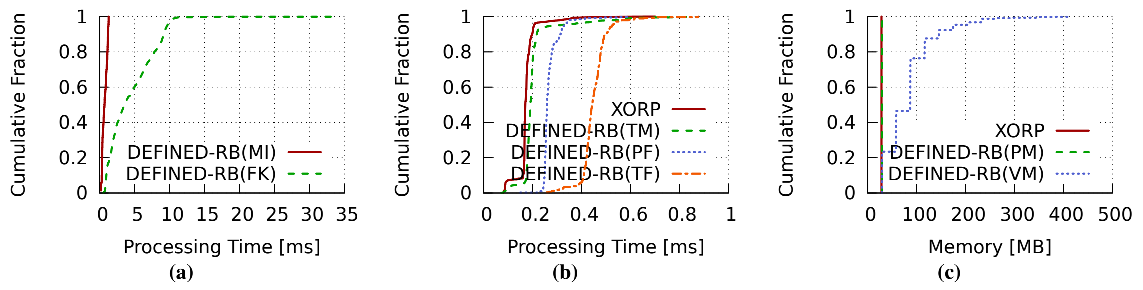


Figure 7: Node-level results of Sprintlink topology with Tier-1 OSPF traces: (a) rollback overhead, (b) non-rollback overhead, and (c) memory overhead of *DEFINED-RB*.

we perform the fork after the packet is processed (to prepare for the next packet). This causes forking to be performed during idle cycles. However, this does not completely remove the forking overhead, as due to copy-on-write, the memory copy associated with the fork is still delayed until the next packet is received. Hence, as a heuristic, we overload `malloc()` to manually touch memory (TM) on the heap when performing the pre-fork. This improves performance further (Figure 7b).

Finally, to achieve its benefits, our approach also incurs some additional memory overhead. Figure 7c shows the amount of virtual memory allocated to each process (VM). We find it increases linearly with the number of forked processes. However, some of this memory is not instantiated in practice due to page sharing. To measure the precise amount of physical memory allocated, we monitor memory writes in */proc/<pid>/mem* in Linux (PM), and plot the memory actually instantiated by the process. Since these processes share the vast majority of memory contents, the amount of memory inflation is small (less than 2% during the entire run).

5.3 Scalability

To investigate the performance of *DEFINED* at scale, and over a wide variety of workloads, we leverage BRITE topologies and synthetic events to investigate the sensitivity of our results to network size and event rate.[8]

Control overhead: We first measure the control overhead with BRITE topologies of varying sizes (Figure 8a). We found that the delay-sensitive pseudorandom ordering optimization described in Section 2.2 (OO) significantly reduces the number of rollbacks (and hence message overhead) of *DEFINED-RB* compared to random orderings (RO). Regardless of the network size, each node only needs to process at most 2 additional packets on average when using the optimized ordering (compared to the unmodified XORP instance).

Delay overhead: Figure 8b shows the network-wide convergence time of *DEFINED-RB* compared to the unmodified XORP instance. Overall, we find that while *DEFINED-RB* has a longer tail in its convergence time distribution (Figure 6b), the average convergence time between the two instances is comparable. In addition, the optimized ordering (OO) again outperforms random orderings (RO).

Response time: To evaluate *DEFINED-LS*, we measure how its response time scales with network size. Figure 8c shows that while the delay of *DEFINED-LS* increases with network size, it increases slowly. In addition, even when the network size grows to 80 nodes, the average delay remains below 0.8 seconds.

Event rates: Finally, to investigate how *DEFINED-RB* scales with event rates, we vary the number of events per second and measure the convergence time. Figure 8d il-

[8]Based on the nature of the bug, debugging can become difficult extremely fast as the network size increases. For nondeterministic bugs, a dozen nodes can already make debugging difficult.

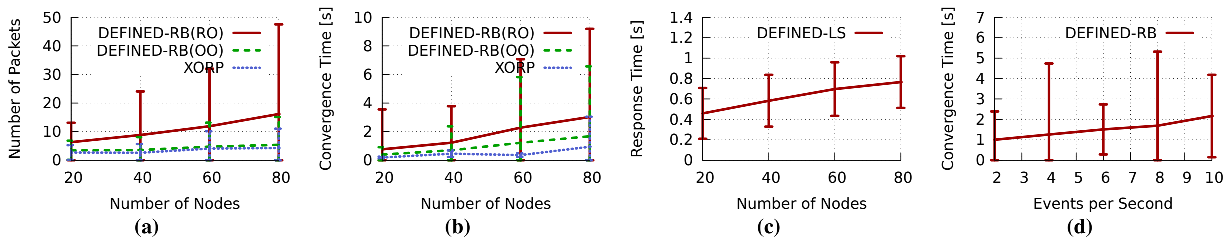


Figure 8: Scalability over network size: (a) control message overheads, and (b) delay of *DEFINED-RB*; (c) response time per step of *DEFINED-LS*. Scalability over event rate: (d) convergence time of *DEFINED-RB*.

lustrates that the convergence time increases slowly as the number of events per second increases, and the average convergence time is only a little bit over 2 seconds when there are 10 events per second. This event rate can easily cover all scenarios we observe in the Tier-1 traces. Nevertheless, when dealing with a higher rate of events, *DEFINED-RB* can decrease its beacon intervals to reduce the number of rollbacks and provide better scalability (as described in Section 2.2).

6 Related Work

DEFINED builds upon existing works and provide new primitives to support debugging of control-plane software in large-scale networks. We leverage works on distributed algorithms [14, 22] to construct the foundations of our design. Our work builds on two key areas:

Deterministic execution: DDOS [12] is the closest work to our design. Similar to *DEFINED*, DDOS introduces deterministic network execution by manipulating message orderings. DDOS runs the distributed software in virtual time, annotates each message with a virtual timestamp, and orders the messages by the source nodes' predefined identification numbers. When the distributed software tries to read a message from the network, DDOS blocks the read request until the correct message arrives. While DDOS provides deterministic network execution to general software, the blocked reads introduced by the algorithm can slow down software that requires constant communications, such as control-plane software. *DEFINED* improves the software's performance in a production network by leveraging speculative execution and introducing an innovative message ordering that minimizes the number of rollbacks.

Jefferson introduced the concept of Virtual Time [14] to provide synchronization for distributed software. Virtual Time is used to determine the ordering of messages, and rollbacks are used to make sure that messages are indeed processed in that order. However, the concept of Virtual Time cannot directly and efficiently be generalized to all software. In *DEFINED*, the message ordering that uses group numbers and estimated delays solidifies and optimizes the Virtual Time idea in the context of control-plane software.

Mechanisms enabling deterministic execution of parallel programs have long been the focus of extensive research. DPJ [6], Dthreads [19], Kendo [25], Tern [8], Determinator [2], and dOS [4] have focused on providing deterministic execution of parallel software with different approaches. DPJ supports determinism at the language level, which ensures more control over the software, but sacrifices generality. On the other hand, Dthreads, Kendo, and Tern offer determinism at the library level, and Determinator and dOS provide determinism at the OS level. These designs allow the system to handle a wider range of software. *DEFINED* takes a step further and guarantees deterministic execution of distributed control-plane software. We leverage a user-space library design, which allows us to support a wide range of control-plane software.

Instead of providing deterministic execution, several works such as Flashback [29], Friday [10], OFRewind [32], Pip [26], ReVirt [9], TTVM [16], and WiDS Checker [20] use comprehensive recordings to ensure reproducibility of execution. However, as the authors of Friday and OFRewind point out, the large storage requirements for logs are one of the limitations of these works. This limitation hinders these works from scaling to large systems, because processing a large amount of logs is prohibitively expensive. Our work targets large-scale networks, where maintaining comprehensive logs may not be tractable.

Finally, *DEFINED* leverages speculative execution, which has been previously used in many systems, for example databases [15] and multi-processor environments [17]. Our work studies the applicability and efficiency of such speculative techniques in large-scale networks. We, further, give several optimizations to reduce the overhead of rollbacks.

Interactive control: *DEFINED* not only ensures that a production network executes in a reproducible fashion, but also enables the network operators to control the execution in a debugging network. Interactive control has been used previously in several works. PDB [13] combines *gdb* with another tool, DISH, to interactively launch, manage, and troubleshoot distributed processes. The effect is similar to using multiple *gdb* instances to troubleshoot multiple processes simultaneously. Similarly, Clairvoyant [33] supports source-level troubleshooting in wireless sensor networks by binding one *gdb* instance to each node. ndb [11] leverages the OpenFlow architecture to provide debugging primitives to software in Software-Defined Networks. Our work complements these techniques by introducing interactive debugging primitives targeting large-scale control-plane software and enabling deterministic network execution.

7 Conclusion

The high complexity of large-scale networks coupled with the rich variety of faults they undergo will require humans to be "in-the-loop" to diagnose complex problems for the foreseeable future. To address this, we proposed techniques for interactive debugging of control-plane software. We specifically addressed two key challenges, namely, deterministic network execution and interactive stepping. Our solution draws from previous work and also proposes new algorithms. We validated our work through a user-space "shim-layer" implementation and extensive evaluation using topologies from Rocketfuel and traces from a Tier-1 ISP. Our results show the practical feasibility and scalability of our approach. Specifically, we leveraged our system to reproduce discovery of known bugs in XORP and Quagga, and showed its benefits over the common debugging method that uses partial recordings and *gdb*.

Acknowledgements

We would like to thank our shepherd Meg Walraed-Sullivan for insightful comments and suggestions. We would also like to thank anonymous reviewers for valuable feedback. This work was supported by NSF CNS-10-40391 and DARPA MRC.

References

[1] G. Altekar and I. Stoica. Focus Replay Debugging Effort on the Control Plane. In *HotDep*, 2010.

[2] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *OSDI*, 2010.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[6] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[7] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security*, 2004.

[8] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable Deterministic Multithreading through Schedule Memoization. In *OSDI*, 2010.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI*, 2002.

[10] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global Comprehension for Distributed Replay. In *NSDI*, 2007.

[11] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the Debugger for my Software-Defined Network? In *HotSDN*, 2012.

[12] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble. DDOS: Taming Nondeterminism in Distributed Systems. In *ASPLOS*, 2013.

[13] IBM. PDB parallel debugger. `http://www-03.ibm.com/systems/software/parallel/index.html`.

[14] D. R. Jefferson. Virtual Time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[15] D. R. Jefferson and A. Motro. The Time Warp Mechanism for Database Concurrency Control. In *ICDE*, 1986.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC*, 2005.

[17] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*, 2010.

[18] C.-C. Lin, V. Jalaparti, M. Caesar, and J. Van der Merwe. DEFINED: Deterministic Execution for Interactive Control-Plane Debugging. Technical report, UIUC, 2013.

[19] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. In *SOSP*, 2011.

[20] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[21] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[22] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[23] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz. Route Flap Damping Exacerbates Internet Routing Convergence. In *SIGCOMM*, 2002.

[24] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *SOSP*, 2005.

[25] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[26] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[27] S. R. Sarangi, B. Greskamp, and J. Torrellas. CADRE: Cycle-Accurate Deterministic Replay for Hardware Debugging. In *DSN*, 2006.

[28] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *SIGCOMM*, 2002.

[29] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A LightWeight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX ATC*, 2004.

[30] The tcpdump Team. tcpdump. `http://www.tcpdump.org/`.

[31] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*, 2002.

[32] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC*, 2011.

[33] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A Comprehensive Source-Level Debugger for Wireless Sensor Networks. In *SENSYS*, 2007.