

SDNRacer: Concurrency Analysis for Software-Defined Networks

IOS (UE17CS302) Assignment

Review of the research paper in PLDI (2016) by El-Hassany, et al.

Bhavna Arora, Raunak Sengupta, Sarthak Gupta, Saioni Chatterjee, Manasa HK
Section 5G, B. Tech (CSE), PES University (Batch of 2017-2021)

Table Of Contents

Table Of Contents	1
1. Summary	2
Some Keywords	2
Need For The SDNRacer	3
Performance Of The SDNRacer	3
Introduction	3
Contributions Of The Paper	4
Overview	4
Formal Model Of SDN Operation	6
Implementation	9
Evaluation	9
Setup	9
Filtering Efficiency	10
Consistency Checks	10
Time	11
Related Work	11
Conclusion	11
2. Reference Reading	12
3. Internet Research	12
4. Insights From The Paper	14
5. Understanding From Code	14
Environment Setup	14
Package Dependencies	14
Setup	14
Implementation	14
Tests	15
6. Individual Contributions And Team Review	19
7. Conclusion	19
8. Appendix	19

1. Summary

SOME KEYWORDS

- I. **SDN**: enables consistent management of the network, which may be made up of complex technology parts. It is a network architecture approach that enables the network to be intelligently and centrally controlled, or 'programmed,' using software applications.
- II. **Concurrency**: When components that operate concurrently interact by messaging or by sharing accessed data (in memory or storage), a certain component's consistency may be violated by another component. Concurrency control provides rules, methods, design methodologies, and theories to maintain the consistency of components operating concurrently while interacting, and thus the consistency and correctness of the whole system. Introducing concurrency control into a system means applying operation constraints which typically result in some performance reduction.
- III. **SDN Switches**: are memory locations which are read / modified by various events.

NEED FOR THE SDNRACER

SDNRacer, is a comprehensive dynamic and controller-agnostic concurrency analyzer for production-grade SDN controllers. It can ensure a network is free of harmful errors such as data races or per-packet incoherences.

SDNRacer is based on two key ingredients:

- I. A precise **happens-before model** for SDNs that captures when events can happen concurrently
- II. A set of sound, **domain-specific filters** that reduce the reported violations by orders of magnitude.

Performance of the SDNRacer

The Authors' of the paper claim that SDNRacer is practically effective.

It quickly (within 30 seconds in 90% of the cases) pinpoints harmful concurrency violations (including unknown bugs) without overwhelming the user with false positives.

INTRODUCTION

Two key principles for SDN:

1. SDN argues for a physical separation between the control-plane, which decides how to forward data packets, and the data-plane, which forwards packets according to control-plane decisions.
2. Second, SDN argues for a (logical) centralization of the control logic which relies on standardized APIs, such as OpenFlow to program forwarding state in each network device (SDN switch).

Two places where concurrent interference can occur:

1. Within the SDN control software itself (e.g., if it is multi-threaded or distributed). These can be detected using standard approaches.
2. At the interface between the control software and the SDN switches. These are harder to detect due to specific unpredictable events.

Detecting these interferences is important as they are typically at the root of deeper semantic problems such as blackholes, forwarding loops or non-deterministic forwarding.

Contributions of the paper

1. First dynamic **controller agnostic concurrency Analyser** for production grade SDN controller
2. Checks for a **variety of errors** such as high level data races, packet coherence violation, update isolation violation
3. HB (**Happens Before**) **model** for commonly used OpenFlow features
4. **Commutativity Specification**: Captures precise conditions under which 2 operations on the network switch commute
5. The **specification and filters** reduce the number of reported issues by several orders of magnitude.

OVERVIEW

Concurrency issues in SDN Programming

SDN controller: Maintain, populate forwarding table of each SDN switch in the network.

Forwarding table: Ordered list of forwarding entries, boolean predicate (a set of packets to which the corresponding forwarding action is applied), forwarding actions (sending the packet to the controller or to a given output port).

Reason for the arise of concurrency issues: 2 unordered accesses to the switch flow table, one of which is a write produced by the controller

Example: a non-deterministic forwarding loop in a load balancer

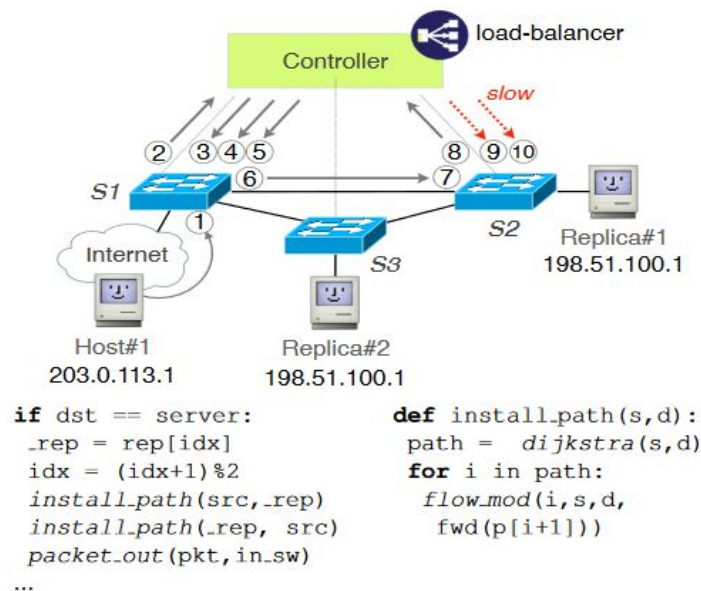


Fig1: Sequence of events leading to forwarding loop and hence concurrency errors

Host 1 sends a request to web server replicas. This request hits the first switch in the network S1. This being a new request, S1 forwards it to the controller telling that a packet has come in (3) & (4). Controller then elects replica 1, computes shortest path between S1 and replica 1, and initiates two write events (one for each traffic direction) on S1 (9) & (10). Similarly on S2 to forward the packets from host 1 to replica1, these are to be installed on S2 causing the packets to be sent back to controller. S1 sends request to S2, packet hits S2 before rules are installed on S2. Thus, leading to concurrency issues.

Reducing concurrency issues:

Reduce the no. of false positive issues that are harmless using 2 distinct filters.

No. of races reduced by the filters $\approx 99.97\%$

- **Filter I - Commuting Events:** if 2 events are interfering with each other (via low level reads and writes) and the network state ends up being identical. Such an interference is therefore harmless and can be filtered out.

The race is harmless if the two events are for non-overlapping entries of the forwarding table, the forwarding table would end being identical irrespective of which event occurs before. Thus these 2 events are said to commute.

- **Filter II - Time:** If a read and a write event are separated by, δ seconds then they are unlikely to be reordered in practice.

SDN developer specifies a time δ after which two events cannot interfere anymore. This δ can easily be estimated based on the maximum network delay and the maximum switch processing time.

Detection of high level properties:

SDNRacer is capable of detecting violations of higher level properties such as inconsistent packet forwarding during a network update.

Update consistency means that packets are either forwarded by the old or the new version of the forwarding state, but not by an interleaving of the two.

FORMAL MODEL OF SDN OPERATION

Operations and events:

- Set of events which succinctly encapsulate the relevant operations performed by the controller, the network switches, and hosts in the network.
- Depending on the event type, set of attributes are chosen out of which only a subset of attributes is used: $\langle \text{pid}, \text{mid}, \text{outpids}, \text{outmids}, \text{msgtype}, \text{sw}, \text{ops} \rangle$ where pid is the identifier of the packet processed by the event.

The HB model uses the packet ids to link causally related events. Flow message processed by the event is of type *msgtype*.

The relevant message types for our analysis are:

- PACKETIN,
- PACKETOUT,

- BARRIERREQUEST,
- BARRIERREPLY,
- PORTMOD,
- FLOWREMOVED and
- FLOWMOD.

Finally, *sw* is a switch identifier, and *ops* is the set of flow table operations the event contains.

SDN FlowTable:

A packet contains a header and a payload. The header consists of a set of fields.

Each flow table entry contains the fields *match*, *priority*, *actions*, *counters*, and *timeouts*. The match can be either an exact match or a wildcard match.

- **Flow table operations for *HandleMsg* events:**
 - *add(eread, strict)*: tries to add a new entry *eadd* to the flow table
 - *mod(eread, strict)*: modifies existing entries in the flow table
 - *del(eread, strict)*: deletes all entries that match the entry *edel* in the flow table
- **Flow table operations for *HandlePkt* events:**
 - *read(pkt)*: a packet *pkt* is matched against the flow table to determine the highest priority flow table entry *eread* that should be applied. If not found set *eread* to none.

HAPPENS BEFORE

For a finite trace consisting of a sequence of events $\pi = \alpha_0 \cdot \alpha_1 \cdot \dots \cdot \alpha_n$, we use $\alpha < \pi \beta$ to denote that event “ α ” occurs before event β in π . We use *HandleMsgs* to denote the set of all the events of type *HandleMsg*.

SPECULATIVE TIME-BASED RULES: Adds edges between events that are highly unlikely to be reordered due to the physical limits of the network. The value of δ depends on the specific parameters of the network

SWITCHDATAPLANE: $\frac{\alpha \in \text{HandlePkts} \cup \text{HandleMsgs} \quad \beta \in \text{SendPkts} \quad \beta.\text{pid} \in \alpha.\text{out_pids}}{\alpha \prec \beta}$	CONTROLPLANETO: $\frac{\alpha \in \text{SendMsgs} \quad \beta \in \text{CtrlHandleMsgs} \quad \beta.\text{mid} \in \alpha.\text{out_mids}}{\alpha \prec \beta}$
SWITCHCONTROLPLANE: $\frac{\alpha \in \text{HandlePkts} \cup \text{HandleMsgs} \cup \text{RemovedFlows} \quad \beta \in \text{SendMsgs} \quad \beta.\text{mid} \in \alpha.\text{out_mids}}{\alpha \prec \beta}$	CONTROLPLANEFROM: $\frac{\alpha \in \text{CtrlSendMsgs} \quad \beta \in \text{HandleMsgs} \quad \beta.\text{mid} \in \alpha.\text{out_mids}}{\alpha \prec \beta}$
SWITCHBUFFER: $\frac{\alpha \in \text{HandlePkts} \cup \text{HandleMsgs} \quad \beta \in \text{HandleMsgs} \quad \beta.\text{pid} \in \alpha.\text{out_pids}}{\alpha \prec \beta}$	BARRIERPRE: $\frac{\alpha, \beta \in \text{HandleMsgs} \quad \alpha.\text{msgtype} = \text{BARRIER_REQUEST} \quad \alpha.\text{sw} = \beta.\text{sw} \quad \alpha <_{\pi} \beta}{\alpha \prec \beta}$
HOST: $\frac{\alpha \in \text{HostHandlePkts} \quad \beta \in \text{HostSendPkts} \quad \beta.\text{pid} \in \alpha.\text{out_pids}}{\alpha \prec \beta}$	BARRIERPOST: $\frac{\alpha, \beta \in \text{HandleMsgs} \quad \beta.\text{msgtype} = \text{BARRIER_REQUEST} \quad \alpha.\text{sw} = \beta.\text{sw} \quad \alpha <_{\pi} \beta}{\alpha \prec \beta}$
CONTROLLER: $\frac{\alpha \in \text{CtrlHandleMsgs} \quad \beta \in \text{CtrlSendMsgs} \quad \beta.\text{mid} \in \alpha.\text{out_mids}}{\alpha \prec \beta}$	TIME1: $\frac{\alpha \in \text{HandlePkts} \cup \text{HandleMsgs} \quad \beta \in \text{HandleMsgs} \quad \beta.t - \alpha.t > \delta}{\alpha \prec \beta}$
DATAPLANE: $\frac{\alpha \in \text{SendPkts} \cup \text{HostSendPkts} \quad \beta \in \text{HandlePkts} \cup \text{HostHandlePkts} \quad \beta.\text{pid} \in \alpha.\text{out_pids}}{\alpha \prec \beta}$	TIME2: $\frac{\alpha \in \text{HandleMsgs} \quad \beta \in \text{HandlePkts} \cup \text{HandleMsgs} \quad \beta.t - \alpha.t > \delta}{\alpha \prec \beta}$

Fig2. Happens-before rules capturing ordering of packets and OpenFlow messages for a trace π

COMMUTATIVITY SPECIFICATION

- Used to improve concurrency of multicore systems as well as to enhance the precision of program analysis dealing with interference (it is important to reduce the number of reported false positives).
- To check commutativity :
 - Compare the results of 2 operations: flow table state (consider two flow table are in the same state if all their flow table entries contain identical priority, match and action fields) and returned valued of the participating operations.
- There are some non-trivial rules – (add, add) , (add, mod) , (del, mod) ,(add, del) , (mod, mod) , (read, add/del/mod).
- Commutativity rules for read are specialized based on the trace order, which is a direct consequence of depending on the state in which the operations were performed.

For a pair of operations a and b, the predicate ϕ_{ab} evaluates to true if operations commute and false otherwise.

- ★ Commutativity specification of an OpenFlow switch: Two read or two del operations always commute.
- ★ The read operations our commutativity specification incorporates parts of the flow table state by using the returned values.

- ★ Commutativity rules for read are specialized based on the trace order, which is a direct consequence of depending on the state in which the operations were performed.

CONSISTENCY PROPERTIES

If we establish the properties holding on a single trace, it follows that the properties hold for all traces which contain the same events where the traces uses the same input.

- **Network Update-** SDN applications update more than one flow rule in the network. Network updates are either triggered reactively by message switches or proactively by an external event.
- **Update isolation-** Executing the updates defined by each policy in any interleaving results in a network state equivalent to the one obtained by some serial execution.
- **Packet Coherence-** A packet trace is coherent if each packet is processed entirely using one consistent global network configuration.

There can be packet coherence even in the presence of races, if the re-ordering of the single event in the races does not negatively affect packet coherence.

Further explanation [Refer Appendix]

IMPLEMENTATION

The implementation carried out in Python consisted of:

1. **SDN troubleshooting system (STS):** to simulate a complete network and track packets, messages and switch operations.
2. **Controller Frameworks (POX, Floodlight, ONOS):** wrapper around the event handlers for incoming messages, links the incoming and outgoing messages. And instrumentation of several **controller frameworks** (POX, FloPOX uses cooperative threading, running only one task at a time while Floodlight and ONOS are multi-threaded and they context-switch threads.
3. **Concurrency analyzer:** that implements the happens-before rules, commutativity checks, and consistency checks. SDNRacer reads events from a trace file, builds the HB graph and then runs the concurrency analysis on top of it.

EVALUATION

Setup:

16GB of RAM + 2.5GHz 4-core processor.

Run on network traces collected from a set of SDN controllers, each between 93 and 24,612 events spanning between 26 and 74 seconds, collected from 200 STS simulation steps.

Tested against all 3 controllers, each on 3 topologies- Single, Linear, and BinTree.

5 Applications- MAC-learning, Forwarding, Circuit Pusher, Admission Control, Load Balancer.

For Circuit Pusher, randomly install a new circuit every second and remove 1 with $P(x) = 0.5$.

For Admission Control, allow 80% of the hosts to communicate.

For Load Balancer, create replica pools with 2 hosts and assign them a VIP, which all hosts send traffic to. Only run on larger topologies.

Filtering Efficiency:

No. of races depends on the no. of read and write events which depend on the controller.

Can't report all races to the developer. Must filter based on commuting events, timing and race coverage.

1. Commutativity- reduces > 33% races in most traces and > 73% in 65.5% traces. Best in traces that have many unrelated reads and writes (different hosts sharing the same path)
2. Time-based- reduces > 20% in half of the traces ($\delta = 2s$). Higher δ gives high FPR.
3. Covered races- reported interferences that can't happen because of high-level dependencies. Only 2.4% of the races.

Consistency Checks:

Detected consistency violations in all our experiments. Mostly subtle (and some are unknown) bugs.

Update consistency violations in every app except LB. For LB, the source is a serious bug.

1. Floodlight Load Balancer distributes flows inconsistently.
Bug: Upon packet reception, the controller was sending the packet back into the network without waiting for the flow rule to be committed to the switch. Further packets went to the controller and triggered the process again. The same flow may be assigned to different replicas.
Fix: Force LB to request a barrier before pushing packets back into the network and buffering any packets for the same connection.
2. **Bug:** POX forwarding module deletes rules installed by other modules.
Fix: Ensure that the Forwarding application only deletes its own flow rules.

Packet coherence violations due to sending a packet without waiting for the flow rules to be committed first. Waiting for writes to be committed slows down network

operations. Consistency/Speed tradeoff. In general, violating per-packet coherence may not always be harmful.

Time:

Took < 32 seconds for most traces. Timed for:

1. Loading the trace
2. Building the HB graph
3. Applying all filters
4. Performing all consistency analysis.

FloodLight Load Balancer on BinTree worst case 3.7 minutes, due to bug in the app.

RELATED WORK

- **Data plane verification:** Anteater, HSA and Libra collect snapshots of the network forwarding state to check for violations. VeriFlow and NetPlumber build on this by allowing real-time checking. A VeriFlow extension allows using assertions to check network properties during controller execution. STS extends these works by considering the minimal sequence of events responsible for a given invariant violation. SDNRacer reports strictly more violations by generalizing the observed trace to all traces obtainable from the same inputs. STS uses network-wide snapshots to check various properties while SDNRacer considers all relevant events. STS outputs the minimal sequence of input events that reproduce an invariant violation while SDNRacer outputs the exact pairs of read/write events that caused the property violation.
- **Controller verification:** Controller bugs are eliminated by synthesizing provably correct controllers. In FlowLog, rulesets are partially compiled to NetCore policies and then verified. NICE uses concolic execution of Python controller programs with symbolic packets and then runs a model checker to determine invariant violations. Kuai applies partial order reduction techniques to reduce the states for exploration. Vericon converts programs into first-order logic formulas and uses a theorem prover for verification. SDNRacer is a dynamic analyzer that operates on actual controller traces and can quickly detect concurrency issues.

CONCLUSION

1. Identify previously unknown and harmful bugs in existing SDN controllers.
2. A precise formal happens-before model of SDN (OpenFlow) concurrency
3. Efficient filters including a commutativity specification of a network switch,

4. A thorough experimental evaluation illustrating that our techniques for filtering races and identifying high-level (consistency) violations work in practice.

2. Reference Reading

OpenSwitch[\[1\]](#)

- Production quality, multilayer virtual switch
- Designed to enable massive network automation through programmatic extension
- Designed to support distribution across multiple physical servers.
- POX – SDN controller
- Components are python programs that uses POX API to modify the state of the switch and respond to information those switches sends to the POX controller.

Floodlight[\[2\]](#)

- Offers a module loading system that make it simple to extend and enhance.
- Easy to set up with minimal dependencies
- Supports a broad range of virtual- and physical- OpenFlow switches
- Can handle mixed OpenFlow and non-OpenFlow networks – it can manage multiple “islands” of OpenFlow hardware switches
- Designed to be high-performance – is multithreaded from the ground up
- Support for OpenStack (link) cloud orchestration platform

POX[\[3\]](#)

- Networking software platform written in Python
- Started as an OpenFlow controller, now an OpenFlow switch, and can be useful for writing networking software.
- POX currently communicates with OpenFlow 1.0 switches and includes special support for the Open vSwitch/Nicira extensions.

3. Internet Research

Some key terms used in the paper:

- **OpenFlow Networks:**
 - enables network controllers to determine the path of network packets across a network of switches.
 - The controllers are distinct from the switches. This separation of the control from the forwarding allows for more sophisticated traffic management than

is feasible using access control lists (ACLs) and routing protocols. Also, OpenFlow allows switches from different vendors — often each with their own proprietary interfaces and scripting languages — to be managed remotely using a single, open protocol. The protocol's inventors consider OpenFlow an enabler of software-defined networking (SDN).

- OpenFlow allows remote administration of a layer 3 switch's packet forwarding tables, by adding, modifying and removing packet matching rules and actions.
- Routing decisions can be made periodically or ad hoc by the controller and translated into rules and actions with a configurable lifespan, which are then deployed to a switch's flow table, leaving the actual forwarding of matched packets to the switch at wire speed for the duration of those rules.
- Unmatched packets are forwarded to the controller

- **Data Plane:**

- Also known as Forwarding Plane
- Forwards traffic to the next hop along the path to the selected destination network according to control plane logic
- Data plane packets go through the router
- The routers/switches use what the control plane built to dispose of incoming and outgoing frames and packets

- **Control Plane:**

- Makes decisions about where traffic is sent
- It is the Signalling of the network
- Control plane packets are destined to or locally originated by the router itself
- Since the control functions are not performed on each arriving individual packet, they do not have a strict speed constraint and are less time-critical

- **Happens Before:**

- In computer science, the happened-before relation (denoted: \rightarrow) is a relation between the result of two events, such that if one event should happen before another event, the result must reflect that, even if those events are in reality executed out of order (usually to optimize program flow).
- This involves ordering events based on the potential causal relationship of pairs of events in a concurrent system, especially asynchronous distributed systems.

- The happened-before relation is formally defined as the least strict partial order on events such that:
 - If events a and b occur on the same process, $a \rightarrow b$, if the occurrence of event a preceded the occurrence of event b
 - If event a ; is the sending of a message and event b , is the reception of the message sent in event a , $a \rightarrow b$

4. Insights from the paper

SDNRacer would detect if there any concurrency violations. These violations are not false positives, and thus a sound HB concurrency analyzer will report them. There are that the number of violations would be much high when SDNcontrollers run more than one application. Many violations originate from the same cause (i.e.,the same bug). The user wouldn't know the root cause of violation and would be given a number of violations.

5. Understanding from code

Environment Setup

We install a 64-bit Ubuntu 14.04 image with default partitioning settings, provisioning 8GB RAM, 32GB HDD, 2 CPUs for it.

Package dependencies:

- General: git, build-essential
- Hassel (header space analysis framework for STS): python-dev
- Floodlight (open source SDN controller): ant, openjdk-7-jdk
- SDN Troubleshooting System (STS): python-docutils
- SDNRacer: python-networkx
- Viewing .dot files: xdot, graphviz
- Plotting: matplotlib, scipy

Setup:

- Clone SDNRacer Repository and checkout HB branch for Happens-before model.
- Verify STS, POX and Hassel repositories within it, install hassel python library for STS.
- Clone Floodlight repository and build using ant.

Implementation:

How is the HB model is loaded: Each line in hb.json contains an event captured while running STS. SDNRacer will load the trace using `def load_trace(self, filename)` defined in `hb_graph.py` and add each event to a graph of HB relations. Each node in the graph is an event while each edge is an HB relation between two events. SDNRacer skips irrelevant messages, such as the handshake messages between the controller and the switches. These messages are defined in `SKIP_MSGS` in `hb_graph.py`.

Race Detector: After loading the graph of HB relations, SDNRacer invokes the `RaceDetector` module (see `./sts/happensbefore/hb_race_detector.py`). The method `detect_ww_races` will search the list of events loaded from the trace and report any two races that don't have HB relations nor they commute as a possible race. The same for `detect_rw_races`, but for read/write events.

Commutativity Specification: This functionality is implemented in `./sts/happensbefore/hb_race_detector.py` and invoked by the `RaceDetector` to check if two events commute or not.

Extracting Packet Traces: In order to check packet coherence property, SDNRacer need to extract the packet traces out of the entire network trace. This is implemented in `def extract_traces(self, g)`.

Checking Packet Coherence: Packet Coherence property is implemented in `find_per_packet_inconsistent` in `hb_graph.py`.

Tests:

- **Circuit Pusher:** This purely proactive application automatically installs paths between two hosts identified by their MAC addresses, as well as the switch and port they are connected to. It uses an external Floodlight process as the controller, and adds/removes circuits through the REST interface exposed by Floodlight. BinaryLeafTreeTopology tree in this example has 3 levels of switches under the root, resulting in a total of $1+2+4+8=15$ switches and $8*2=16$ hosts.

```
$ ./simulator.py -L logging.cfg -c
config/trace_floodlight_circuitpusher.py
```

The above command runs the config file and produces a trace after 200 rounds, which is stored in the hb.json file stored in the traces directory. The race detection is then run using SDNRacer's happens-before model.

```
$ ./sts/happensbefore/hb_graph.py
traces/trace_floodlight_circuitpusher-BinaryLeafTreeTopology1-steps200
/hb.json
```

* Race analysis *

```
Total number of events in the trace: 606
Total number of events with read operations: 136
Total number of events with write operations: 79
Total number of events with read or write operations: 215
Total number of observed races without any filters: 4608
Total number of commuting races: 4214
Total number of races filtered by Time HB edges: 338
Total number of races covered by data dependency: 0
Remaining number of races after applying all enabled filters: 56 (1.22%)
```

* Properties analysis *

```
Number of observed network updates: 12
Number of update isolation violations: 6
Total number of packets in the traces: 65
Number of packet coherence violations: 20
Number of packet coherence violations filtered due covered races: 0
Number of packet coherence but only on the first switch in the update: 11
Number of packet coherence violations after filtering covered races: 9
```

* Timing information *

```
Done. Time elapsed: 0.611974000931 s
load_trace: 0.191139936447 s
detect_races: 0.170279026031 s
extract_traces_time: 0.010400056839 s
find_reactive_cmds_time: 0.00101590156555 s
find_proactive_cmds_time: 0.0764169692993 s
find_covered_races_time: 9.53674316406e-07 s
per_packet_inconsistent_time: 0.00304412841797 s
find_inconsistent_update_time: 0.000123977661133 s
```

These are the produced results. The below command is used to produce a pdf of the visualisation of the graphviz file generated.

```
$ dot -Tpdf
traces/trace_floodlight_circuitpusher-BinaryLeafTreeTopology1-steps200
/hb.dot -o
```


traces/trace_floodlight_circuitpusher-BinaryLeafTreeTopology1-steps200/hb.pdf

- **MAC-learning:** A purely reactive application builds and maintains a dynamic MAC address table for each switch. This table maps known MAC addresses to the physical port on which they can be reached. This simulation will terminate after 100 rounds. The BinaryLeafTreeTopology tree in this example has 1 level of switches under the root, resulting in a total of $1+2=3$ switches and $2*2=4$ hosts.

```
$ ./simulator.py -L logging.cfg -c config/demo_pox_l2_learning.py
```

```
$ ./sts/happensbefore/hb_graph.py
experiments/demo_pox_l2_learning/hb.json
```

* Race analysis *

Total number of events in the trace: 491

Total number of events with read operations: 85

Total number of events with write operations: 55

Total number of events with read or write operations: 140

Total number of observed races without any filters: 1901

Total number of commuting races: 1804

Total number of races filtered by Time HB edges: 77

Total number of races covered by data dependency: 0

Remaining number of races after applying all enabled filters: 20 (1.05%)

* Properties analysis *

Number of observed network updates: 57

Number of update isolation violations: 0

Total number of packets in the traces: 44

Number of packet coherence violations: 13

Number of packet coherence violations filtered due covered races: 0

Number of packet coherence but only on the first switch in the update: 10

Number of packet coherence violations after filtering covered races: 3

* Timing information *

Done. Time elapsed: 0.40608716011 s

load_trace: 0.134392023087 s

detect_races: 0.081650018692 s

extract_traces_time: 0.00996494293213 s

find_reactive_cmds_time: 0.00221800804138 s

find_proactive_cmds_time: 0.0501770973206 s

find_covered_races_time: 1.90734863281e-06 s

per_packet_inconsistent_time: 0.00192213058472 s

find_inconsistent_update_time: 2.50339508057e-05 s

- **Load Balancer:** This application performs stateless load balancing among a set of replica identified by a virtual IP address (VIP). Upon receiving packets destined to a VIP, the application selects a particular host and installs flow rules along the entire path.

Violation from paper: Upon packet reception, the controller selects a replica, pushes flow rules to direct traffic to it, sends the packet back in the network towards the replica without waiting for the flow rule to be committed to the switch. Concretely, this means that multiple load-balancing decisions can be taken for the same flow.

```
$ ./simulator.py -L logging.cfg -c config/trace_floodlight_loadbalancer.py
```

```
$ ./sts/happensbefore/hb_graph.py --data-dep
./paper/trace_floodlight_loadbalancer-fixed-BinaryLeafTreeTopology2-steps200/hb.json
```

* Race analysis *

Total number of events in the trace: 21402

Total number of events with read operations: 1849

Total number of events with write operations: 5545

Total number of events with read or write operations: 7394

Total number of observed races without any filters: 3654183

Total number of commuting races: 3610031

Total number of races filtered by Time HB edges: 42177

Total number of races covered by data dependency: 0

Remaining number of races after applying all enabled filters: 1975 (0.05%)

* Properties analysis *

Number of observed network updates: 1363

Number of update isolation violations: 906

Total number of packets in the traces: 203

Number of packet coherence violations: 80

Number of packet coherence violations filtered due covered races: 0

Number of packet coherence but only on the first switch in the update: 8

Number of packet coherence violations after filtering covered races: 72

* Timing information *

Done. Time elapsed: 138.014732122 s

load_trace: 32.3178610802 s

detect_races: 91.5820109844 s

extract_traces_time: 0.587380886078 s

find_reactive_cmds_time: 2.78195405006 s

find_proactive_cmds_time: 2.9247610569 s

find_covered_races_time: 0.785151958466 s

per_packet_inconsistent_time: 0.651242017746 s

find_inconsistent_update_time: 1.43894696236 s

SDNRacer extract a subgraph of the HB relations that point directly to the violation (for ease of use). This subgraph can be visualized using:

```
$ dot -Tpdf
paper/trace_floodlight_loadbalancer-fixed-BinaryLeafTreeTopology2-step
s200/isolation_violation_100.dot -o
paper/trace_floodlight_loadbalancer-fixed-BinaryLeafTreeTopology2-step
s200/isolation_violation_100.pdf
```


6. Individual Contributions and Team Review

	Paper Reading	Reference Reading	Paper Summarization	Code Understanding	Report / PPT	Overall
Bhavna Arora PES1201700062	10	9	9.5	8.5	9	9.2
Raunak Sengupta PES1201700072	10	8.5	9.5	9	8.5	9.1
Sarthak Gupta PES1201700077	10	8.5	9.5	9	9	9.2
Saioni Chatterjee PES1201700118	10	9	9	8	8	8.8
Manasa HK PES1201701886	10	8.5	9	8	8	8.7

7. Conclusion and Future Directions

Conclusion:

1. Happens-Before Model for SDN: Captures asynchrony of SDN
2. Flow Table Commutativity Spec: Captures Interference
3. Concurrency Analysis:
 - a. Race Freedom
 - b. Network Update Isolation
 - c. Packet Coherence

- 
4. Implementation and Evaluation: Found bugs in existing apps: ONOS, POX, Floodlights

Future Directions:

The software for the current implementation has very outdated dependencies (Ubuntu 14, JDK 7, Python 2.7). All with having ended official support or ending in 2019. Also, the code in python is not very optimised. A future direction could be porting the code to updated versions of the dependencies.

8. Appendix

References Read:

- [1] [OpenFlowSwitchSpecification.Version1.0.0.https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf](https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf)
- [2] <http://www.projectfloodlight.org/floodlight>
- [3] <https://noxrepo.github.io/pox-doc/html/>

SDNRacer: Concurrency Analysis for SDN's

Summary

~ Bhavna, Raunak, Sarthak, Saioni, Manasa

1

Some Keywords

SDN: A network architecture approach that enables the network to be intelligently and centrally controlled, using software applications.

SDN Switches: Memory locations read / modified by various events.

SDNRacer: A dynamic, controller-agnostic concurrency analyzer for SDN controllers.

2

SDNRacer. What it is. What it does.

- It is a comprehensive dynamic and controller-agnostic concurrency analyzer for production-grade SDN controllers.
- Can ensure a network is free of harmful errors such as data races or per-packet incoherences.
- Can quickly (< 30 secs) pinpoint harmful concurrency violations.

3

How it does it.

1. Loading the trace
2. Building the Happens Before graph
3. Applying all filters.
4. Performing all consistency analysis.

4

Why it is needed.

Concurrent interference can occur within the SDN control software itself, and at the interface b/w the control software and the SDN switches.

These interferences are at the root of deeper semantic problems such as black holes, forwarding loops or non-deterministic forwarding.

5

Two key ingredients

A precise **happens-before model** for SDNs.
Captures when events can happen concurrently

Sound, **domain-specific filters**. Reduces the reported violations by orders of magnitude.

Two key principles

Physical separation b/w the control-plane, (network layer), and the data-plane, (transport layer).

Logical centralization of the control logic which relies on standardized APIs, such as OpenFlow to program forwarding state in each SDN switch.

6

Concurrency issues in SDN Programming

We maintain and populate forwarding table of each SDN switch in the network.

A forwarding table contains the ordered list of forwarding entries, boolean predicate, forwarding actions.

2 unordered accesses to the switch flow table, one of which is a write produced by the controller results in concurrency issues.

7

Update Consistency

SDNRacer is capable of detecting violations of higher level properties such as inconsistent packet forwarding during a network update.

SDNRacer provides update consistency, i.e. that packets are either forwarded by the old or the new version of the forwarding state, but not by an interleaving of the two.

8

Filters - Reducing concurrency issues

Commuting Events: A race is harmless if the two events are for non-overlapping entries of the forwarding table, the forwarding table would end being identical irrespective of which event occurs before. Thus these 2 events are said to commute.

Time: If a read and a write event are separated by, δ seconds then they are unlikely to be reordered in practice.

Such interferences ($\cong 99.97\%$) are harmless and can be filtered out.

9

Happens Before (HB)

In computer science, the happened-before relation (denoted: \rightarrow) is a relation between the result of two events, such that if one event should happen before another event, the result must reflect that, even if those events are in reality executed out of order (usually to optimize program flow).

This involves ordering events based on the potential causal relationship of pairs of events in a concurrent system, especially asynchronous distributed systems.

10

Happens Before (HB)

The happened-before relation is formally defined as the least strict partial order on events such that:

If events a and b occur on the same process, $a \rightarrow b$ if the occurrence of event a preceded the occurrence of event b

If event a is the sending of a message and event b is the reception of the message sent in event a , $a \rightarrow b$

11

Commutativity Specification

Used to improve concurrency of multicore systems as well as to enhance the precision of program analysis dealing with interference (it is important to reduce the number of reported false positives).

Two read or two del operations always commute.

(add, add) , (add, mod) , (del, mod) , (add, del) , (mod, mod) , $(read, add/del/mod)$ are trivial rules.

Commutativity rules for read are specialized based on the trace order, which is a direct consequence of depending on the state in which the operations were performed.

12

Consistency Properties

Network updates- Update more than one flow rule in the network. Either triggered reactively by message switches or proactively by an external event.

Update isolation- Executing the updates defined by each policy in any interleaving results in a network state equivalent to the one obtained by some serial execution.

Packet Coherence- A packet trace is coherent if each packet is processed entirely using one consistent global network configuration.

13

What the code includes.

SDN troubleshooting system (STS): to simulate a complete network and track packets, messages and switch operations.

Controller Frameworks: wrapper around event handlers, links the incoming and outgoing messages. And instrumentation of several controller frameworks.

Concurrency analyzer: that implements the happens-before rules, commutativity checks, and consistency checks. SDNRacer reads events from a trace file, builds the HB graph and then runs the concurrency analysis on top of it.

14

Setup used for evaluation

16GB of RAM + 2.5GHz 4-core processor.

Run on network traces collected from a set of SDN controllers, each between 93 and 24,612 events spanning between 26 and 74 seconds, collected from 200 STS simulation steps.

Tested against all 3 controllers, each on 3 topologies- Single, Linear, and BinTree.

5 Applications- MAC-learning, Forwarding, Circuit Pusher, Admission Control, Load Balancer.

15

Filtering

No. of races depends on the no. of read and write events which depend on the controller. Can't report all races to the developer.

Must filter based on commuting events, timing and race coverage.

16

Filtering in SDNRacer

1. **Commutativity**- reduces > 33% races in most traces and > 73% in 65.5% traces. Best in traces that have many unrelated reads and writes.
2. **Time-based**- reduces > 20% in half of the traces ($\delta = 2s$). Higher δ gives high FPR.
3. **Covered races**- reported interferences that can't happen because of high-level dependencies. Only 2.4% of the races.

17

Conclusion

1. Identify previously unknown and harmful bugs in existing SDN controllers.
 2. A precise formal happens-before model of SDN (OpenFlow) concurrency
 3. Efficient filters including a commutativity specification of a network switch,
 4. A thorough experimental evaluation illustrating that our techniques for filtering races and identifying high-level (consistency) violations work in practice.
-

18

Conclusion

1. Happens-Before Model for SDN:
Captures asynchrony of SDN
 2. Flow Table Commutativity Spec:
Captures Interference
 3. Concurrency Analysis:
 - a. Race Freedom
 - b. Network Update Isolation
 - c. Packet Coherence
 4. Implementation and Evaluation: Found bugs in existing apps: ONOS, POX, Floodlights
-

19

Insights

SDNRacer would detect if there any concurrency violations. These violations are not false positives, and thus a sound HB concurrency analyzer will report them. There are that the number of violations would be much high when SDN controllers run more than one application. Many violations originate from the same cause (i.e., the same bug). The user wouldn't know the root cause of violation and would be given a number of violations.

20

Future Directions

- The software for the current implementation has very outdated dependencies (Ubuntu 14, JDK 7, Python 2.7).
- All with having ended official support or ending in 2019.
- Also, the code in python is not very optimised. A future direction could be porting the code to updated versions of the dependencies.