

# Experiences from Testing and Verifying “Real-World” Concurrent and Distributed Systems

**Kostis Sagonas**



UPPSALA  
UNIVERSITET

**UP/MARC**

Uppsala Programming for  
Multicore Architectures  
Research Center

# Analyzing concurrent software

- Concurrent software (e.g., Linux kernel) is central to the infrastructure of our society
- Important to ensure correctness
  - including absence of concurrency errors
- We need techniques that
  - are automatic
  - provide high coverage
  - and scale.

# Concurrent programming is HARD

- Concurrent execution is difficult to reason about and get right – even for experts!
- Rare process interleaving results in bugs that are:
  - hard to anticipate;
  - hard to find, reproduce and debug (“Heisenbugs”);
  - hard to be sure whether they are really fixed.
- Big productivity problem: it can waste significant developers’ time and resources.
- Can have severe consequences.
- Weak memory makes things harder...



# Stateless Model Checking (SMC)

aka **Systematic Concurrency Testing**

A technique to **detect** concurrency errors or **verify** their absence by exploring all possible ways that scheduling non-determinism can influence a program's safety property.

- + Fully automatic.
- + Has low memory requirements.
- Applicable to data-deterministic programs with finite executions.

# How SMC works

Assume that you only have one 'scheduler'.

Run an arbitrary execution of the program...

Then:

Backtrack to a point where some other thread could have been chosen to run...

From there, continue with another execution...

Repeat until all choices have been explored.

# Systematic Exploration Example

Initially:  $x = y = 0$

Thread 1

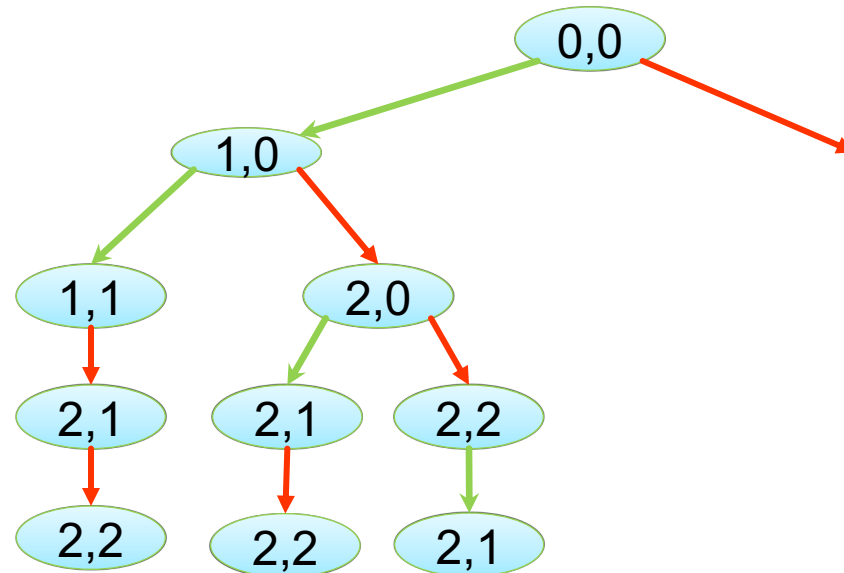
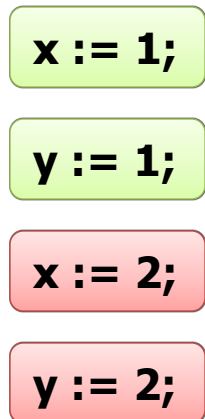
**$x := 1;$**   
 **$y := 1;$**

Thread 2

**$x := 2;$**   
 **$y := 2;$**

Correctness Property (at the end)

**$\text{assert}(x == y);$**



Exploration can stop early when a property is violated.

# Systematic Exploration Example

Initially:  $x = y = 0$

Thread 1

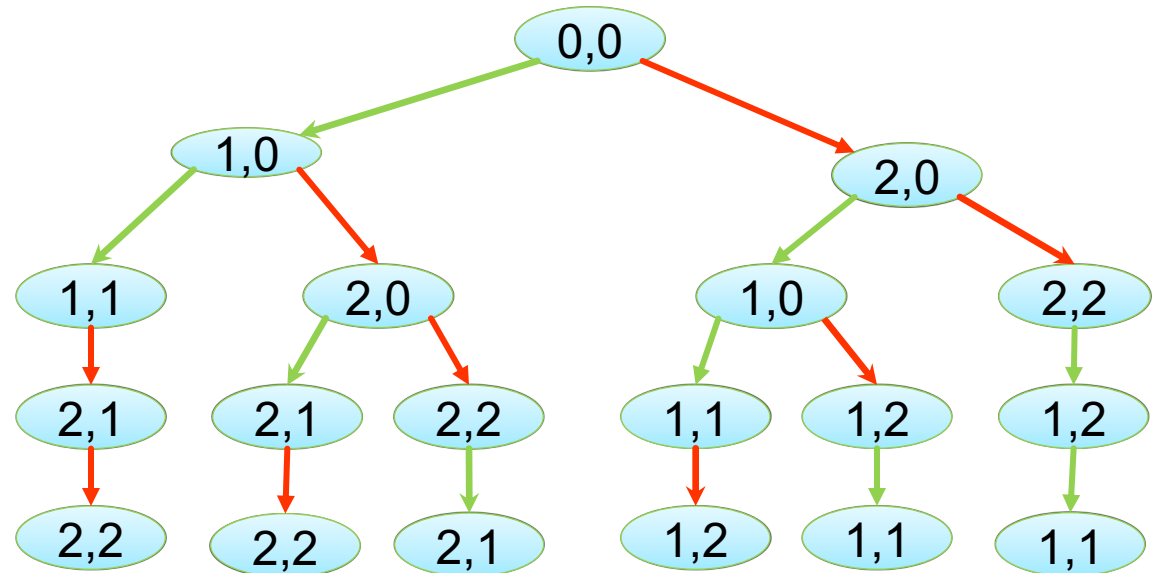
$x := 1;$   
 $y := 1;$

Thread 2

$x := 2;$   
 $y := 2;$

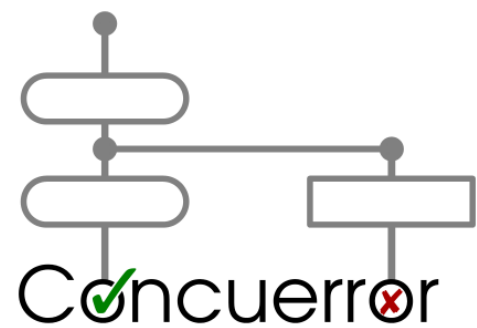
Correctness Property (at the end)

**$\text{assert}((x + y) < 7);$**



Exploration needs to visit the complete set of traces for properties that hold.

# Concuerror



A **stateless model checker** for **Erlang** that systematically explores **all** possible behaviours of a program annotated with some assertions, to

- either detect concurrency errors  
(in which case it reports the erroneous trace)
- or verify their absence  
(i.e., that the properties in the assertions hold)



# Concuerror Paper

## Systematic Testing for Detecting Concurrency Errors in Erlang Programs

Maria Christakis\*, Alkis Gotovos\* and Konstantinos Sagonas<sup>†‡</sup>

\* *Department of Computer Science, ETH Zurich, Switzerland*

<sup>†</sup> *Department of Information Technology, Uppsala University, Sweden*

<sup>‡</sup> *School of Electrical and Computer Engineering, National Technical University of Athens, Greece*

*maria.christakis@inf.ethz.ch   alkisg@ethz.ch   kostis@it.uu.se*

ICST 2013

**Abstract**—We present the techniques used in Concuerror, a systematic testing tool able to find and reproduce a wide class of concurrency errors in Erlang programs. We describe how we take advantage of the characteristics of Erlang’s actor model of concurrency to selectively instrument the program under test and how we subsequently employ a stateless search strategy to systematically explore the state space of process interleaving sequences triggered by unit tests. To ameliorate the problem of combinatorial explosion, we propose a novel technique for avoiding process blocks and describe how we can effectively combine it with preemption bounding, a heuristic algorithm for reducing the number of explored interleaving sequences. We also briefly discuss issues related to soundness, completeness and effectiveness of techniques used by Concuerror.

our work, which allows finding important concurrency errors even with a small number of preemptions between processes.

This paper presents the techniques used in Concuerror, a stateless model checking tool, that, given an Erlang program and its (existing) test suite, systematically explores process interleaving and presents detailed interleaving information about errors (such as abnormal process exits, stuck processes and assertion violations) that occur during the execution of these tests. Concuerror is extremely easy to use and can support the test-driven development of Erlang programs [2].

The focus of this paper is on Concuerror’s implementation technology. More specifically, we describe how we take ad-

# Systematic $\neq$ Naive

Literally explore “all traces”?? Too many!

Not all pairs of events are conflicting.

Each explored trace should be **different**.

# Partial Order Reduction

Combinatorial explosion in the number of interleavings.

Initially:  $x = y = \dots = z = 0$

Thread 1:  
 $x := 1$

Thread 2:  
 $y := 1$

Thread N:  
 $z := 1$

- Interleavings under naïve exploration:  **$N!$**
- Interleavings needed to cover all behaviors: **1**

## Partial Order Reduction (POR)

- ✓ Explore just a subset of all interleavings
- ✓ Still cover all behaviors

# Optimal DPOR [POPL'14, JACM'17]

The exploration algorithm

- ... monitors **actual conflicts** between events;
- ... explores additional interleavings **as needed**;
- ... completely avoids **equivalent** interleavings.

**Dynamic:** at runtime, using concrete data.

**Optimal:**

- explores only one interleaving per equivalence class;
- does not even initiate redundant ones.

# Optimal DPOR Exploration

Initially:  $x = y = 0$

Thread 1

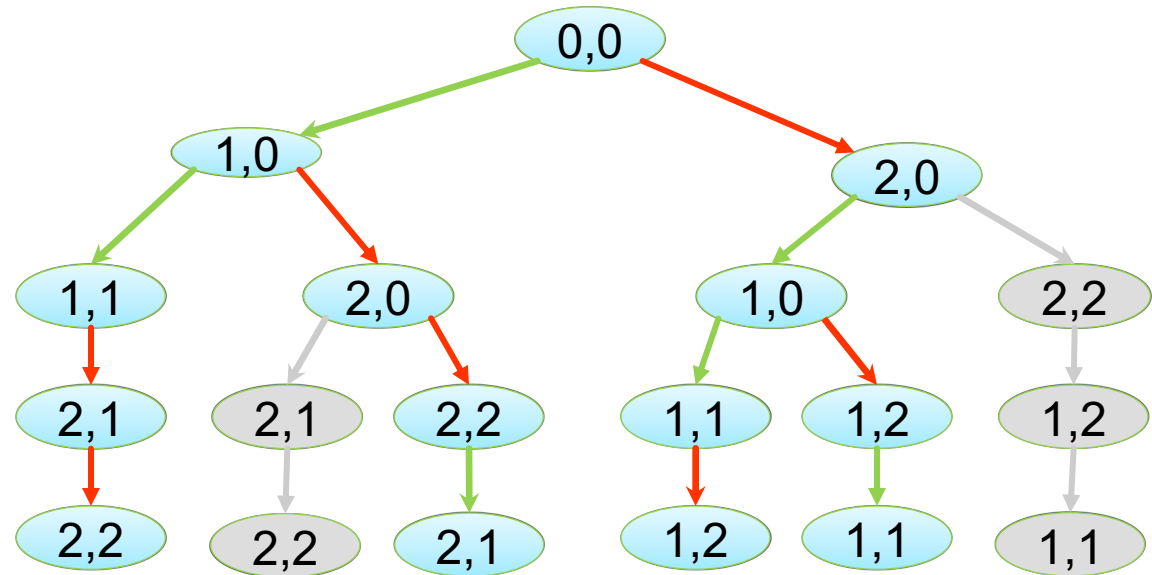
$x := 1;$   
 $y := 1;$

Thread 2

$x := 2;$   
 $y := 2;$

Correctness Property (at the end)

$\text{assert}((x + y) < 7);$



Optimal DPOR will not explore the grey nodes.

# Bounding Techniques

Explore only a few traces based on some bounding criterion.

E.g., number of times threads can be preempted, delayed, etc.

- Very effective for testing!
- Not suitable for verification.

# Exploration Using Preemption Bounding

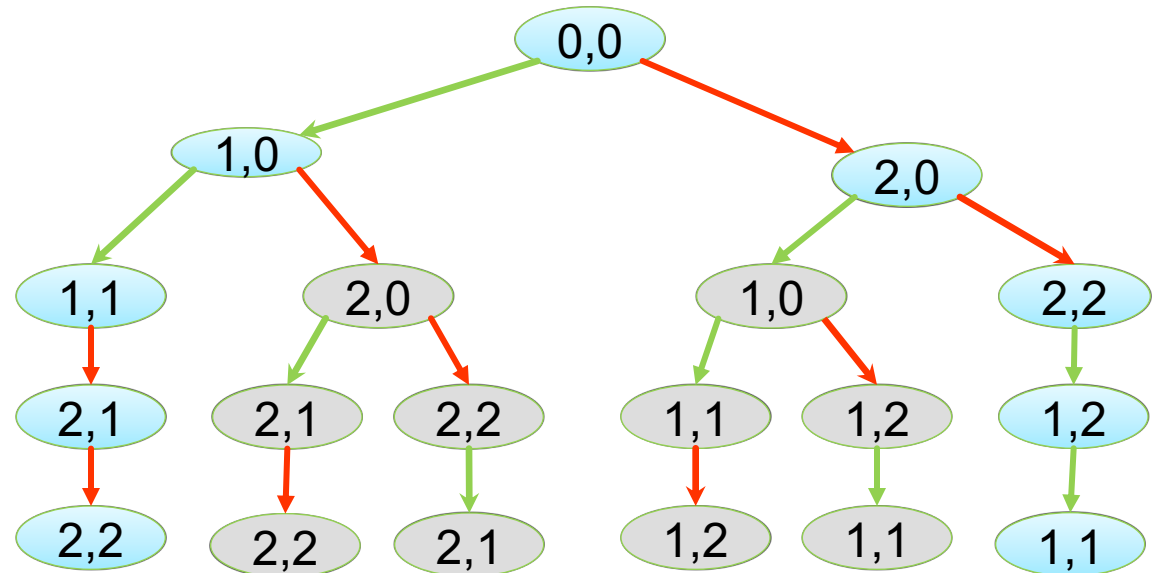
Initially:  $x = y = 0$

Thread 1

$x := 1;$   
 $y := 1;$

Thread 2

$x := 2;$   
 $y := 2;$



With a **preemption bound of 0**,  
the grey nodes will not be explored.

# Exploration Using Preemption Bounding

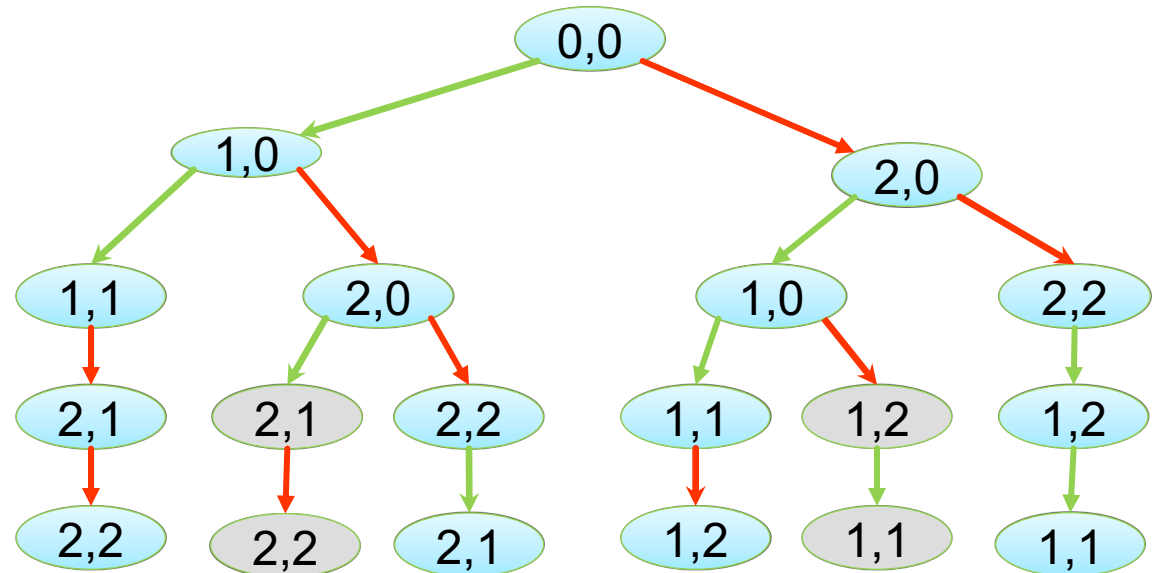
Initially:  $x = y = 0$

## Thread 1

```
x := 1;  
y := 1;
```

## Thread 2

```
x := 2;  
y := 2;
```



With a **preemption bound of 1**,  
the grey nodes will not be explored.



# Nidhugg

A **stateless model checker** for **C/threads** that

- either detects bugs caused by concurrency
  - assertion violations
  - robustness violations
  - memory bugs (e.g., double-free errors)
- or verifies their absence.

Implements effective DPOR algorithms with extensions for checking effects of weak memory models (TSO, PSO and POWER).

# 1. Nidhugg on RCU

## Stateless Model Checking of the Linux Kernel's Hierarchical Read-Copy-Update (Tree RCU)

Michalis Kokologiannakis

National Technical University of Athens, Greece

Konstantinos Sagonas

Uppsala University, Sweden

National Technical University of Athens, Greece

### ABSTRACT

Read-Copy-Update (RCU) is a synchronization mechanism used heavily in key components of the Linux kernel, such as the virtual filesystem (VFS), to achieve scalability by exploiting RCU's ability to allow concurrent reads and updates. RCU's design is non-trivial, requires significant effort to fully understand it, let alone become convinced that its implementation is faithful to its specification and

### 1 INTRODUCTION

The Linux kernel is used in a surprisingly large number of devices: from PCs and servers, to routers and smart TVs. For example, in 2015 more than one billion smart phones used a modified version of the Linux kernel [5], and in 2016 almost all modern supercomputers used Linux as well [22]. It is self-evident that the correct and reliable operation of the Linux kernel is of great importance, which renders

**Extended version published at STTT 2019 journal**

# The Linux kernel

- Ubiquitous: billion+ instances deployed!  
A once-in-a-million bug occurs several times a day!
- Used on many platforms with weak memory.
- Large codebase.
- Fast releases: new release every 2 months!
- Concurrency bugs often manage to survive after years of intensive stress testing.  
There is a need for better concurrency error testing.

# Our contribution

- We verified the fundamental guarantee of the Linux kernel's [Read-Copy-Update](#) synchronization mechanism using Nidhugg.
- We used the actual code from the Linux kernel (~15kLoC) and a plethora of different versions.
- Scaffolded a Linux-kernel SMP environment
  - which is as precise as possible;
  - but also allows the state space to be small.
- Verification is **fast!**

# Times are very small :-)

**SPIN 2017**

v3.0			v3.19			v4.3			v4.7			v4.9.6		
Time		Traces	Time		Traces	Time		Traces	Time		Traces	Time		Traces
SC	TSO	Explored	SC	TSO	Explored	SC	TSO	Explored	SC	TSO	Explored	SC	TSO	Explored
<b>227.37</b>	<b>260.38</b>	<b>19398</b>	<b>594.33</b>	<b>651.32</b>	<b>24760</b>	<b>1041.85</b>	<b>1147.20</b>	<b>28996</b>	<b>411.25</b>	<b>416.03</b>	<b>11076</b>	<b>1033.12</b>	<b>1125.30</b>	<b>28996</b>
2.09	2.30	145	1.47	1.66	37	1.70	1.77	29	1.93	2.09	29	2.06	2.16	29
2.11	2.27	146	1.51	1.67	41	1.99	1.97	33	2.09	2.26	33	2.13	2.32	33
0.43	0.38	4	0.63	0.66	3	0.80	1.03	3	1.18	1.22	3	1.14	1.27	3
27.60	31.02	2372	399.89	433.38	13264	334.13	316.14	8114	290.50	329.10	8114	307.72	338.26	8114
1.35	1.46	84	3.01	3.13	79	1.87	2.08	24	3.09	3.17	43	2.99	3.26	43
58.15	64.80	4888	1.18	0.98	9	1.22	1.39	9	1.57	1.66	9	1.60	1.65	9
1.14	1.41	1	3.34	3.33	2	5.28	5.13	2	10.40	10.34	2	10.80	10.93	2
24.91	26.42	2024	10.73	11.32	608	11.27	10.88	488	10.15	11.64	488	10.77	11.85	488
50.28	52.37	3888	10.46	11.13	608	12.30	11.95	516	11.55	12.49	516	11.80	13.18	516
26.38	28.98	2184	12.48	13.63	688	11.37	11.23	488	11.66	12.75	532	11.89	13.27	532

All tests run in less than 20minutes on a 5 year-old desktop!

# Read-Copy-Update (RCU)

- A synchronization mechanism used in the Linux kernel since 2002.
- Allows concurrent reads and updates, without
  - performance degradation
  - scalability limitations (up to 16,777,216 CPUs)
  - deadlocks and memory leaks.
- Allows read paths to be extremely fast.  
(Zero overhead in non-preemptible kernels!)

# How does RCU work?

By maintaining multiple versions of data structures which are not freed until it is certain that they are no references to them.

**Basic idea:** Split updates in two phases:

- removal phase;
- reclamation phase.

Capitalizes on the fact that, on modern CPUs, writes to a single aligned pointer are atomic.

# RCU readers

- Have no memory footprint.
- Not permitted to block or sleep.
  - So, when a CPU executes e.g. a context switch, any prior RCU read-side critical sections will have completed.
  - As soon as each CPU has executed at least one context switch, all pre-existing readers have completed.
- “Wait for readers” mechanism.



# Formalizing aspects of RCU

Statements that are not within an RCU read-side critical section are said to be in a **quiescent state**.

Any time period during which each CPU resides at least once in a quiescent state is called a **grace period**.

*The “wait for readers” mechanism has to wait for at least one grace period to elapse.*

# Grace-Period Guarantee

*If any statement in a given RCU read-side critical section precedes a grace period, then all statements in that RCU read-side critical section must complete – including memory operations – before that grace period ends.*

# Kernel environment modeling

- Multiple CPUs: Modeled using mutexes.
- RCU relies on timer interrupts.
  - However, *exact* timing is not important.
  - An interrupt lock for each CPU was used.
- Per-CPU variables are modeled with arrays.
- Many other kernel primitives were “faked”, e.g., wait queues, synchronization primitives, etc.

# Verification times

Kernel	Time (SC)	Time (TSO)	Traces Explored
v3.0	3m47s	4m20s	19398
v3.19	9m54s	10m51s	24760
v4.3	17m22s	19m07s	28996
v4.7	6m51s	6m56s	11076
v4.9.6	17m13s	18m45s	28996

Memory needed was at most 35MB!

# Verified, really?

Actually, we only verified some RCU properties, most notably:

The Grace-Period guarantee  
for non-preemptible kernels.

But can we really trust these results?

- Anybody can write a program that prints “Verified!”
- We injected bugs to RCU’s code, inspired from real kernel failures observed throughout the past years.

# Bug detection times (secs)

	Time (SC)	Time (TSO)	Traces Explored
Bug1	2.06	2.16	29
Bug2	2.13	2.32	33
Bug3	1.14	1.27	3
Bug4	307.72	338.26	8114
Bug5	2.99	3.26	43
Bug6	1.60	1.65	9
Bug7	10.80	10.93	2
Bug8	10.77	11.85	488
Bug9	11.80	13.18	516
Bug10	11.89	13.27	532

- Most injected bugs are found within seconds!
- In contrast, CBMC needs days and many GBs.

# Some results

- Examined reported “bug” for multi-node hierarchies in kernel v2.6.31:
  - + Proved that there is no such interleaving that can cause the scenario described in the commit log and in the relevant LKML mails
  - + We constructed a test case which shows that the conditions needed to provoke the bug cannot happen for any interleaving
- We also found the real cause behind these failures
  - + We constructed a test case showing that a read-side critical section can span a grace period (forbidden)

Test case showed that the bug is not present in kernel v3.0!

# In short: Nidhugg on RCU

- Verified the basic properties for Tree RCU, the main RCU implementation used in the Linux kernel.
- Found the real cause for most relevant failures observed in the past.
- Demonstrated that a previously reported bug does not qualify as a bug.



# 2. Concuerror on CORFU

*iFM 2017*

## Testing and Verifying Chain Repair Methods for CORFU Using Stateless Model Checking

Stavros Aronis<sup>1</sup>(✉), Scott Lystig Fritchie<sup>2</sup>, and Konstantinos Sagonas<sup>1</sup>

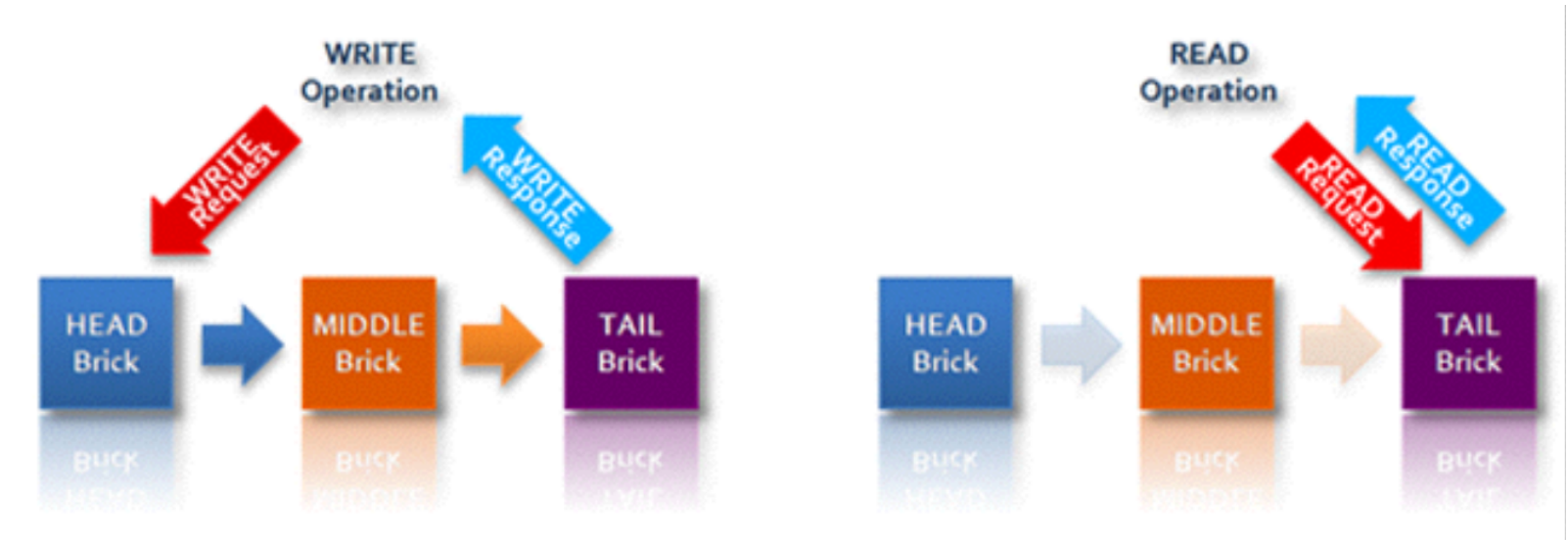
<sup>1</sup> Department of Information Technology, Uppsala University, Uppsala, Sweden  
{stavros.aronis,kostis}@it.uu.se

<sup>2</sup> VMware, Cambridge, MA, USA  
sfritchie@vmware.com

**Abstract.** CORFU is a distributed shared log that is designed to be scalable and reliable in the presence of failures and asynchrony. Internally, CORFU is fully replicated for fault tolerance, without sharding data or sacrificing strong consistency. In this case study, we present the modeling approaches we followed to test and verify, using Concuerror, the correctness of repair methods for the Chain Replication protocol suitable for CORFU. In the first two methods we tried, Con-

# Chain Replication [OSDI'04]

A variant of master/slave replication.  
Strict chain order:



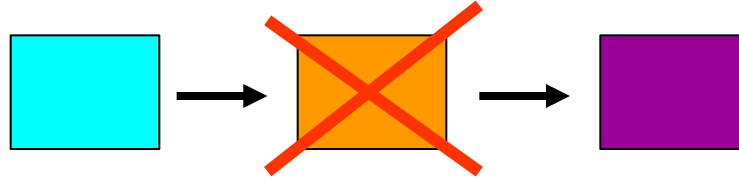
Sequential read @ tail.

Linearizable read @ all.




Dirty read @ head or middle.

# Chain Repair


Suppose chain of three servers:



Naive offline repair method:

1. Stop all surviving servers in the chain  → 
2. Copy tail's update history to the repairing node 
3. Restart all nodes with the new configuration



A better repair method for CR systems places the repairing node directly on the chain and reads go to  (the old tail).

# CORFU [SIGOPS'12, NSDI'17]

Uses Chain Replication with three changes:

1. Responsibility for replication is moved to the client.
2. CORFU's servers implement write-once semantics.
3. Identifies each chain configuration with an epoch #.
  - All clients and servers are aware of the epoch #.
  - The server rejects clients with a different epoch #.
  - A server temporarily stops service if it receives a newer epoch # from a client.

# Engineers @ VMWare (1)

Investigated methods for chain repair in CORFU

- Method #1: ~~Add to the tail~~

# Chain Repair in CORFU

A repair during epoch #5: a client is writing a *new* value to the cluster for a data with *old* value.

epoch #5	$S_{head}^a$ value= <i>new</i>	$S_{tail}^b$ value= <i>old</i> or value= <i>new</i>	$S_{repair}^c$ value= <i>old</i>
epoch #6	$S_{head}^a$ value= <i>new</i>	$S_{middle}^b$ value= <i>new</i>	$S_{tail}^c$ value= <i>old</i> or value= <i>new</i>

There is a race condition here, which can lead to a linearizability violation.

# Engineers @ VMWare (2)

Investigated methods for chain repair in CORFU

- Method #2: ~~Add to the head~~



**Scott L. Fritchie**

@slfritchie

Following



I was all ready to have a celebratory "New algorithm works!" tweet. Then the DPOR model execution w/Concuerror found an invalid case. Ouch.

RETWEET

1

LIKES

5



9:16 AM - 23 Jun 2016

# Modeling CORFU in Erlang

## Initial model:

- Some (**one** or **two**) servers undergo a chain repair to add **one** more server to their chain.
- Concurrently, **two** other clients try to write **two** different values to the same key.
- While a third client tries to read the key **twice**.



# Modeling CORFU (cont.)

- Servers and clients are modeled as Erlang processes.
- All requests are modeled as messages.

## Processes used by the model:

- Central coordinator
- CORFU log servers (2 or 3)
- Layout server process
- CORFU reading client
- CORFU writing clients (2)
- Layout change and data repair process

# Correctness Properties

## **Immutability:**

Once a value has been written in a key, no other value can be written to it.

## **Linearizability:**

If a read sees a value for a key, subsequent reads for that key must also see the same value.

# Three Repair Methods

1. Add repair node at the tail of the chain.
2. Add repair node at the head of the chain.
3. Add repair node in the middle.
  - a. Configuration with two healthy servers.
  - b. Configuration with one healthy server which is “logically split” into two.

# Results in (old) Concuerror

Method	Bounded Exploration			Unbounded Exploration		
	Bug?	Traces	Time	Bug?	Traces	Time
1 (Tail)	Yes	638	57s	Yes	3 542 431	144h
2 (Head)	Yes	65	7s	Yes	389	26s
3 (Middle)	No	1257	68s	No	>30 000 000	>750h

# Model Refinements

## 1. Conditional read:

Avoid issuing read operations that are sure to not result in violations.

## 2. Convert layout server process to an ETS table (instead of a process).

# Optimization (in DPOR)

- Treat blocking receives, whose message patterns are all known, specially.
- Avoids exploring an exponential number of "unnecessary" interleavings from sends.

## Optimal Dynamic Partial Order Reduction with Observers

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas

Dept. of Information Technology, Uppsala University, Uppsala, Sweden



TACAS 2018

**Abstract.** Dynamic partial order reduction (DPOR) algorithms are used in stateless model checking (SMC) to combat the combinatorial explosion in the number of schedulings that need to be explored to guarantee soundness. The most effective of them, the Optimal DPOR algorithm, is optimal in the sense that it explores only one scheduling per Mazurkiewicz trace. In this paper, we enhance DPOR with the notion of *observability*, which makes dependencies between operations conditional on the existence of future operations, called *observers*. Observers naturally

# Effect of Model Refinements

Method #3 (repair node in the middle)

Concuerror verified its correctness:

- in 48 hours;
- after exploring 3 931 412 traces.

Method #1 (repair node in the tail)

Even *without* bounding, an error was found in just 19 seconds (212 traces).

# Concluding Remarks

Stateless model checking tools, like Concuerror, can be used not only for programs, but also for protocols and algorithms at their design phase to

- detect bugs in them
- or verify their correctness.

Writing a correct and efficient model can be quite challenging.

It is still very difficult (for users) to reason why their correct model is hard to verify.