TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Randomized Testing of Distributed Systems

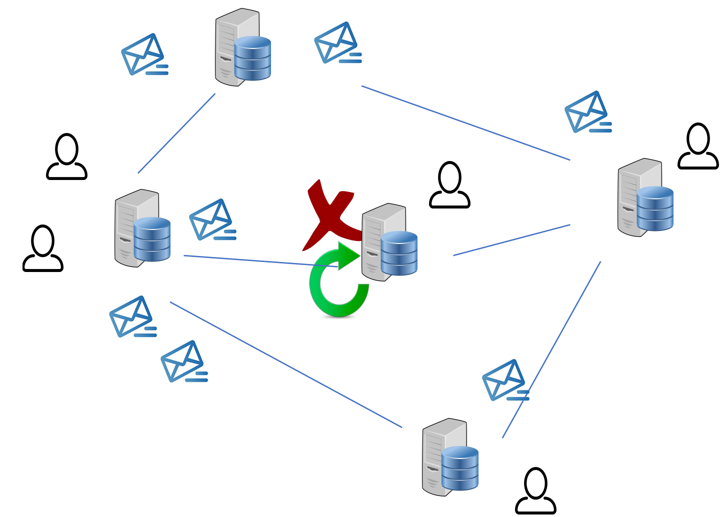
Burcu Kulahcioglu Ozkan

TU Kaiserslautern

Summer Term 2019

Distributed systems are prone to bugs!

- Distribution
- Asynchrony
- Replication
- …



They are difficult to test!

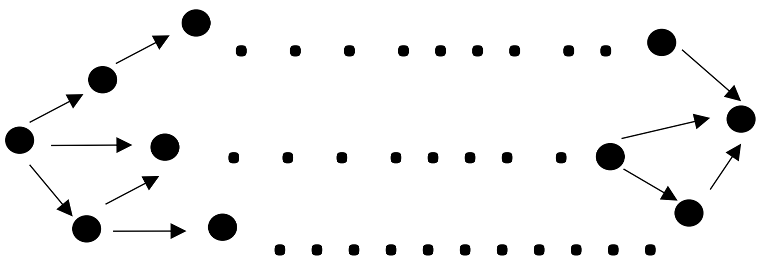
- Many components, many sources of nondeterminism

Cassandra / CASSANDRA-14702
Cassandra Write failed

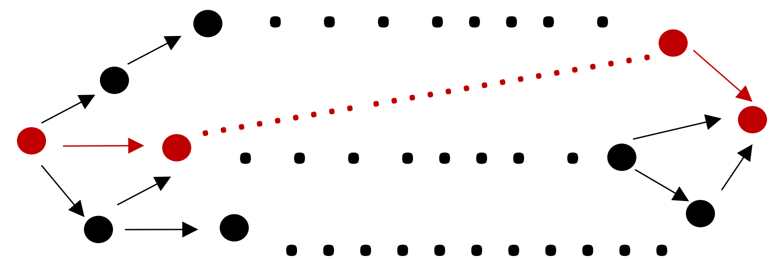
HBase / HBASE-20368
Fix RIT stuck when a rsgroup has no online servers

ZooKeeper / ZOOKEEPER-2930
Leader cannot be elected due to network timeout

Testing is a practical approach



Systematic testing - infeasible



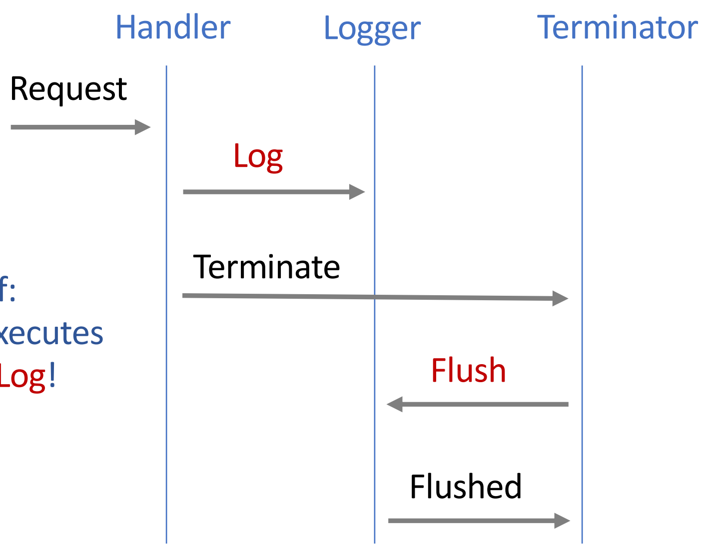
Random testing – no guarantees

Randomized Testing with Probabilistic Guarantees

(joint work with Rupak Majumdar, Filip Niksic, Simin Oraee, Mitra Tabaei Befrouei, Georg Weissenbacher)

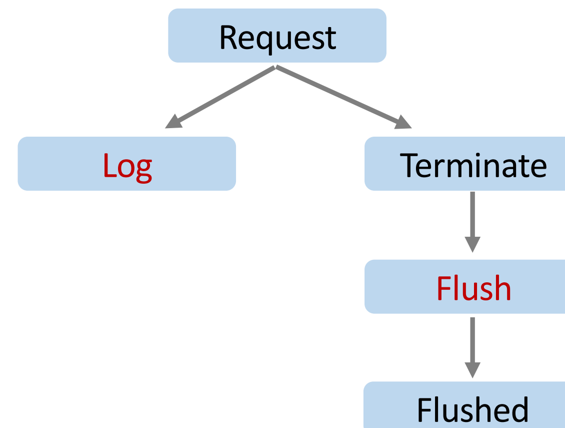
- We propose a randomized scheduling algorithm:
  - for arbitrary partially ordered sets of events revealed online as the program is being executed
  - Guaranteeing a lower bound on the probability of exposing a bug

PCTCP on an example



Buggy if:
Flush executes
before Log!

Upgrowing Poset:



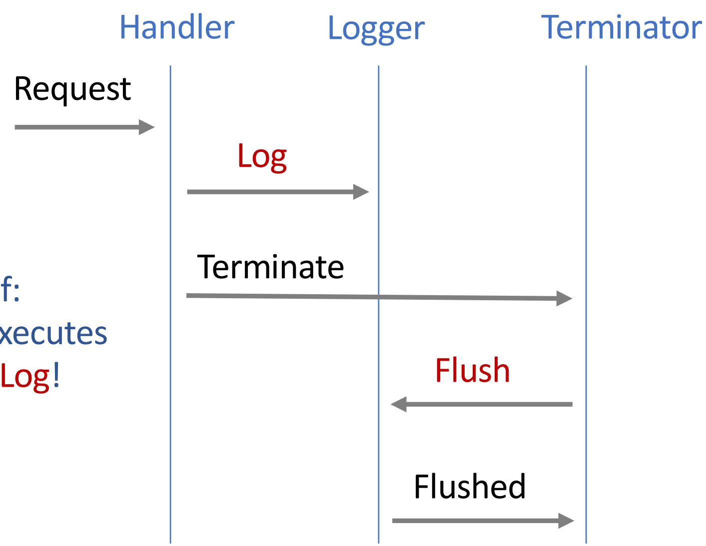
The program is decomposed into causally dependent chains of events:

Online chain partitioning:

$$C1 = [Request, Log]$$
$$C2 = [Terminate, Flush, Flushed]$$

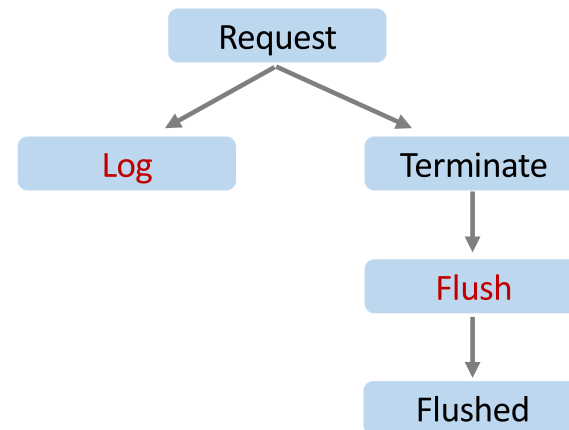
$$priority(C1) > priority(C2)$$

PCTCP on an example



Buggy if:
Flush executes before Log!

Upgrowing Poset:



Online chain partitioning:

$C1 = [Request, Log]$
$C2 = [Terminate, Flush, Flushed]$

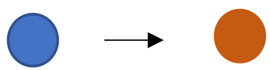
$priority(C2) > priority\ (C1)$

The bug is detected with probability:

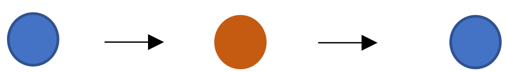
$PCTCP:\ 1/2$
$Random\ walk:\ 1/4$

Bug depth: Minimum tuple of events to expose the bug

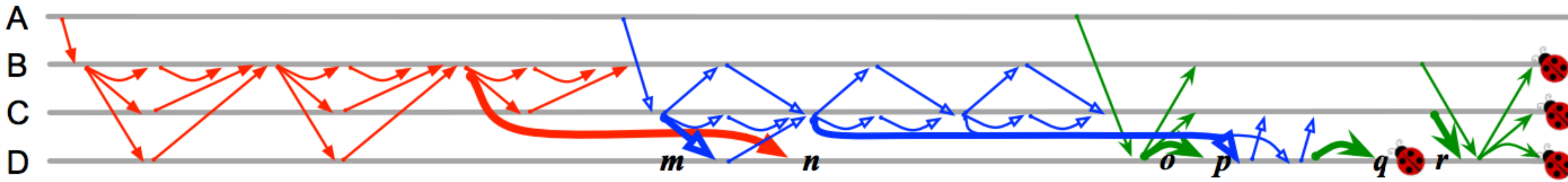
- $d = 2 \ \langle e_1, e_2 \rangle$ e.g. order violation



- $d = 3 \ \langle e_1, e_2, e_3 \rangle$ e.g. atomicity violation



- $d = n \ \langle e_1, \ldots, e_n \rangle$ more complicated bugs

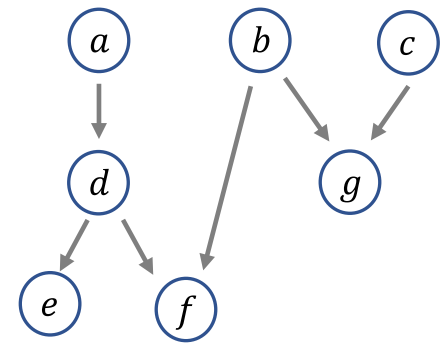


Bug in Cassandra 2.0.0 *(img. from Leesatapornwongsa et. al. ASPLOS'16)*

Coverage: Strong d-Hitting families of schedules

A schedule α strongly hits $\langle e_0, \ldots, e_{d-1} \rangle$ if for all $e \in P$:

$e \geq_\alpha e_i$ implies $e \geq e_j$ for some $j \geq i$



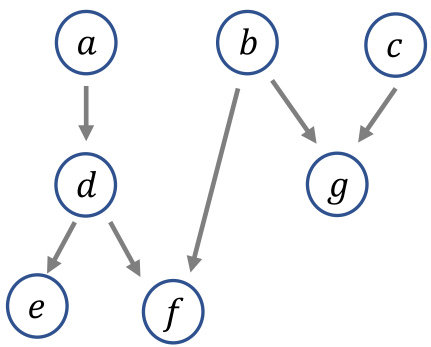
$\alpha 1 = \; a, b, c, d, f, e, g$
strongly hits 1–tuple $\langle g \rangle$, 2–tuple $\langle e, g \rangle$

$\alpha 2 = \; a, b, c, d, f, g, e$
strongly hits 1–tuple $\langle e \rangle$, 2–tuple $\langle g, e \rangle$, 3-tuple $\langle d, g, e \rangle$

For each d-tuple, a **strong d-hitting family** has a schedule which strongly hits it.

Challenge: How to sample uniformly at random from strong d-hitting family for distributed systems?

- Events in a distributed message passing system:
  upgrowing poset, revealed during execution
- Mutual dependency to the schedule



- Build a schedule online
- For an arbitrary ordering

Use combinatorial results for posets!

Schedule: $a\ d\ e\ b\ f\ c\ g$

Realizer and dimension of a poset

Realizer of P is a set of linear orders:

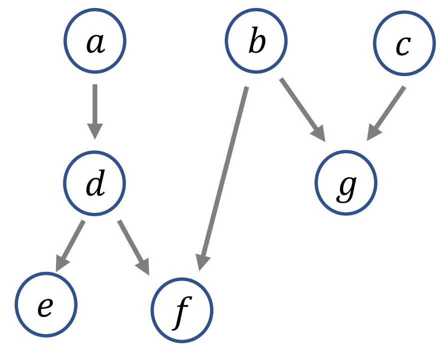
$$F_R = \{L_1, L_2, \ldots, L_n\}$$

such that: $L_1 \cap L_2 \ldots \cap L_n = P$

Dimension of P is the minimum size of a realizer

Realizer of size $\dim(P)$

- Covers all pairwise orderings!



$L_1 = a\ d\ e\ b\ f\ c\ g$

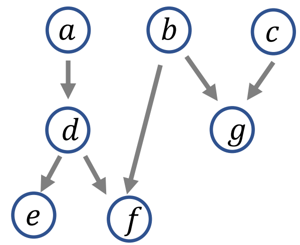
$L_2 = c\ a\ d\ e\ b\ g\ f$

$L_3 = c\ b\ g\ f\ a\ d\ e$

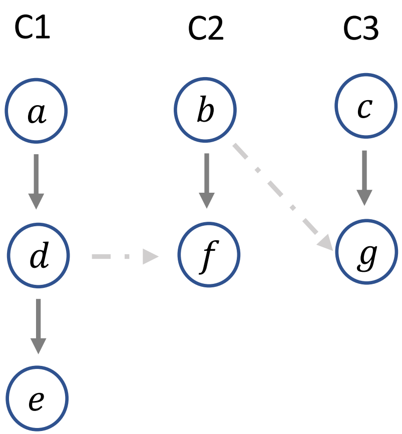
$\dim(P) = 3$

Adaptive chain covering ~ Online dimension algorithm

- Decompose P into chains



- Compute linear extensions of P



$L1 = b\ c\ g\ f\ a\ d\ e$

$L2 = c\ a\ d\ e\ b\ g\ f$

$L3 = a\ d\ e\ b\ f\ c\ g$

This is a strong 1-hitting family!

Adaptive chain covering ~ Strong 1-hitting family ~ Online dimension algorithm

Adaptive chain covering ~ Online dimension algorithm

[Felsner'97, Kloch'07]

Strong $\boldsymbol{d}$-hitting family ~ Adaptive chain covering

[Felsner, Kloch] Strong 1-hitting family ~ Adaptive chain covering

$$hit(w) = adapt(w)$$

[Our main result] Strong $\boldsymbol{d}$-hitting family ~ Adaptive chain covering

$$hit_d(w,n) \leq adapt(w)\binom{n}{d-1}(d-1)!$$

n: number of events
d: bug depth

Index the schedules in the strong d-hitting family by: $\langle \lambda, n_1, n_2, \ldots, n_{d-1} \rangle$ strongly hits $e_0 \in Chain(\lambda)$ and $e_1, e_2, \ldots, e_{d-1}$

λ: chain id

$n_1, n_2, \ldots, n_{d-1}$: steps in which $e_1, e_2, \ldots, e_{d-1}$ were added

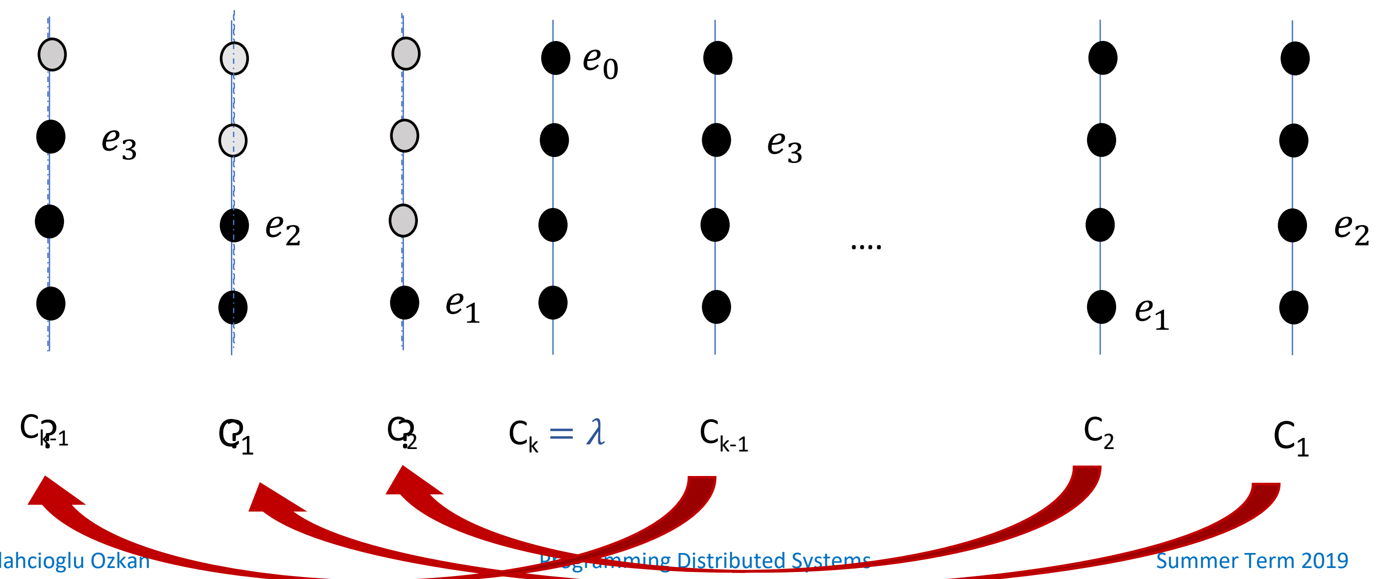
Sample from this set of schedules!

PCTCP : PCT + Chain Partitioning

Generates randomly a schedule index $\langle \lambda, n_1, n_2, \ldots, n_{d-1} \rangle$:

- Randomly generate a $(d-1)$-tuple: $\langle n_1, n_2, \ldots, n_{d-1} \rangle$
- Partition P into chains online
- Assign random distinct initial priorities $> d$
- Reduce priority at: $\langle e_1, e_2, \ldots, e_{d-1} \rangle$ to $(d-i-1)$ for e_i

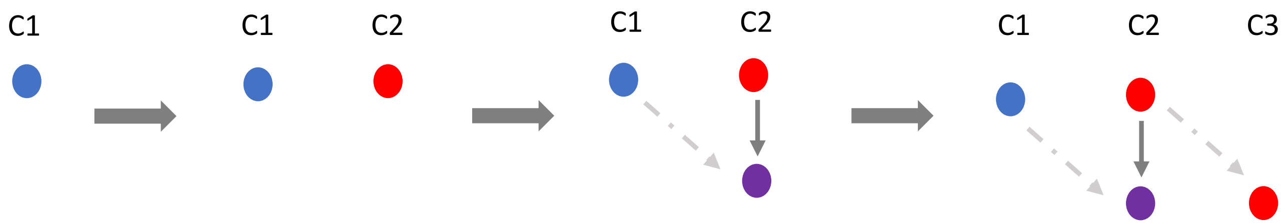
strongly hits $e_0 \in Chain(\lambda)$ and $e_1, e_2, \ldots, e_{d-1}$



The prob. of hitting a bug – Generalizes the PCT result

$$hit_d(w,n) \leq \ adapt(w)\binom{n}{d-1}(d-1)! \leq \underbrace{adapt(w)}_{\text{online width of the poset of width } w}\, n^{d-1}$$

- Not possible to partition P of width w into w chains online in general:



- [Felsner, 95] The best possible on-line partitioning algorithm partitions upgrowing P of width w into $\binom{w+1}{2}$ chains!

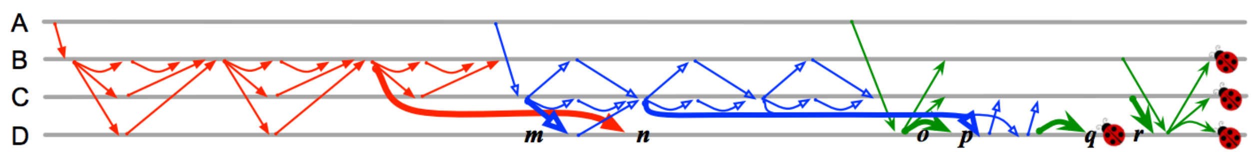
We sample from at most $w^2 n^{d-1}$ schedules,

hitting a bug of depth d with a probability of at least $\frac{1}{w^2 n^{d-1}}$

n: number of events
d: bug depth

Experimental results - Cassandra

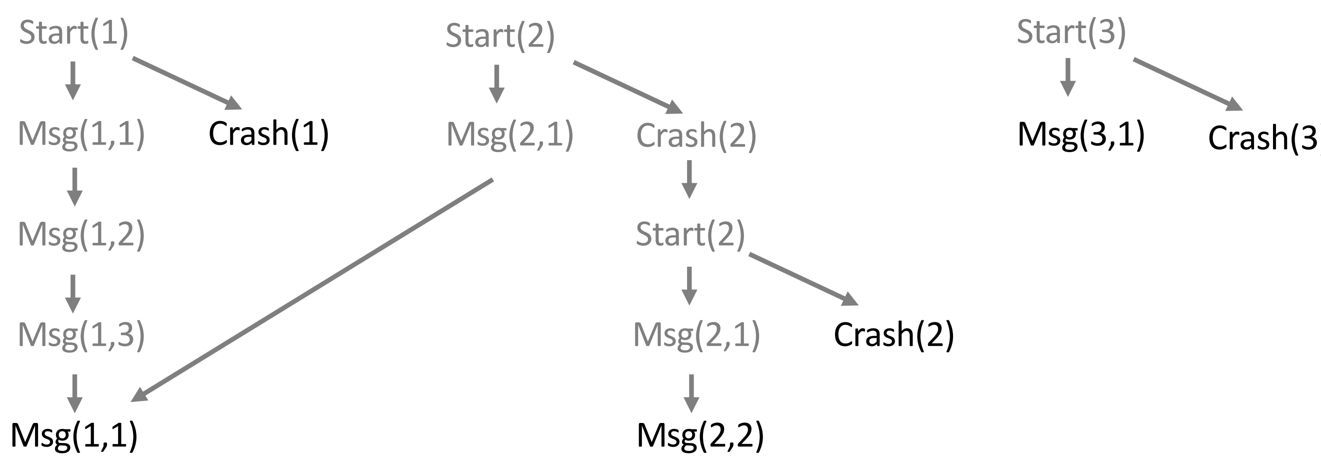
	# Event Labels (d)	Max # Events (n)	Avg of Max # Chains	Max # Chains	# Runs	#Buggy	Time(s)
Random Walk	-	54	6.97	11	1000	0	481.95
PCTCP	d = 4	54	5.65	11	1000	0	505.73
PCTCP	d = 5	54	5.73	11	1000	1	503.81
PCTCP	d = 6	54	5.80	11	1000	1	512.00



Bug in Cassandra 2.0.0 *(img. from Leesatapornwongsa et. al. ASPLOS'16)*

Source code at: https://gitlab.mpi-sws.org/fniksic/PSharp
Source code at: https://gitlab.mpi-sws.org/burcu/pctcp-cass
Source code at: https://gitlab.mpi-sws.org/rupak/hitmc

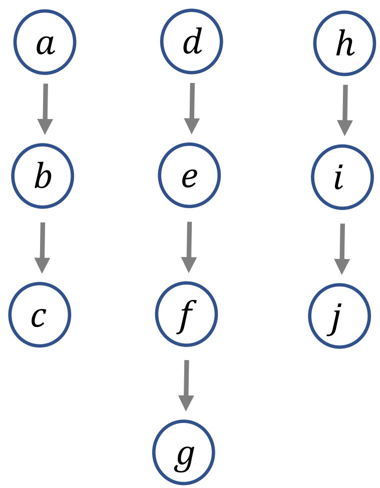
Experimental results - ZooKeeper



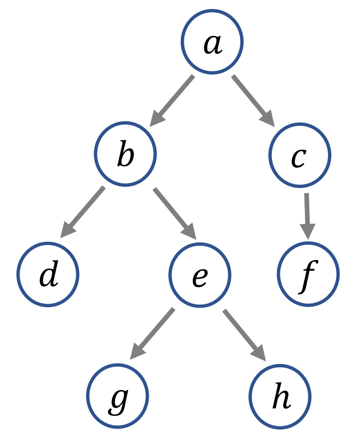
Source code at: https://gitlab.mpi-sws.org/fniksic/PSharp
Source code at: https://gitlab.mpi-sws.org/burcu/pctcp-cass
Source code at: https://gitlab.mpi-sws.org/rupak/hitmc

Related Work

PCT for multithreaded programs, linear orders
[Burckhardt, Kothari, Musuvathi, Nagarakatte, 2010]



d-Hitting families of schedules, trees
[Chistikov, Majumdar, Niksic, 2016]



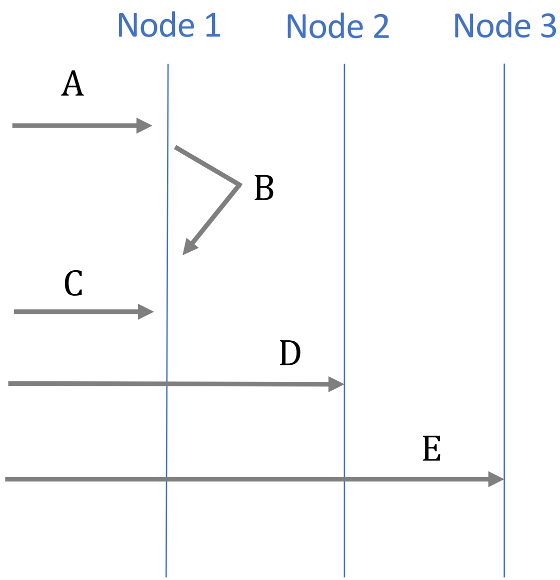
Our method hits a bug with a prob. $\frac{1}{adapt(w)n^{d-1}}$

Generalizes the PCT result $\frac{1}{k\ n^{d-1}}$

Our method samples from hitting families for **any arbitrary upgrowing poset**

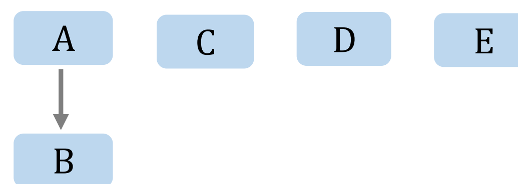
TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Current Work: Partial Order Reduction for Hitting Families



Some schedules in strong hitting family are equivalent :

Upgrowing Poset:

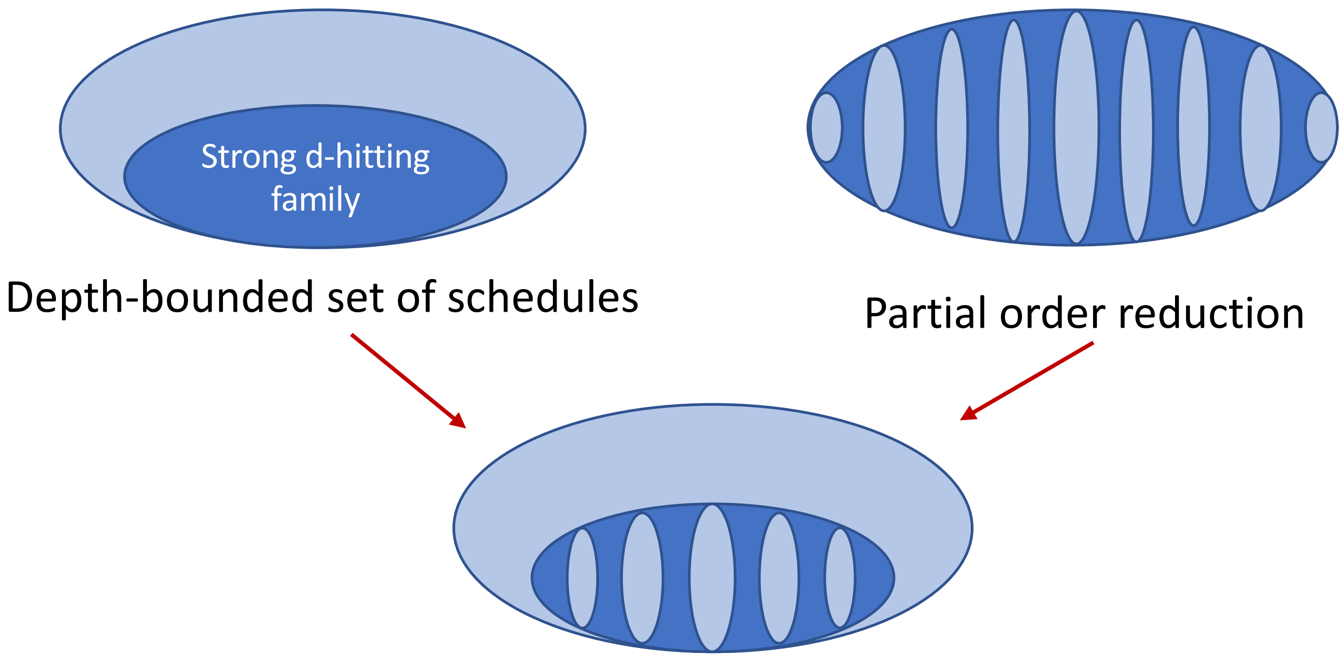


e.g. Two schedules strongly hitting $\langle E \rangle$ and $\langle D \rangle$:

$$A\ B\ C\ D\ E \equiv A\ B\ C\ E\ D$$

Can we use POR techniques for randomized testing?

Depth-Bounded + Dependency-Aware Random Testing



Sample from a smaller set of schedules!

Summary – PCTCP :

A randomized testing method PCTCP with probabilistic guarantees for distributed message passing systems

- Depth-bounded sampling from strong d-hitting families of schedules
  - Combinatorial results on dimension theory, adaptive chain covering
  - Indexing strong d-hitting families of schedules of size $hit_d(w,n) \leq adapt(w)n^{d-1}$
- Our result generalizes the PCT guarantee:
  - Hitting a bug with prob. of at least $1 / (adapt(w)n^{d-1})$

Randomized Testing with Jepsen

- Test tool for safety of distributed databases, queueing systems, consensus systems etc.
- Black-box testing by randomly inserting network partition faults
- Developed by Kyle Kingsbury, available open-source
- Approach:
    1. Generate random client operations
    2. Record history
    3. Verify that history is consistent with respect to the model

Example: Jepsen Analysis for MongoDB

- MongoDB is a document-oriented database
- Primary node accepting writes and async replication to other nodes

Test scenario:

- 5 nodes, n1 is primary
- Split into two partitions (n1, n2 and n3, n4, n5), n5 becomes new primary
- Heal the partition

How many writes get lost?

- In Version 2.4.1. (2013)
  - Writes completed 93.608 seconds 6000 total 5700 acknowledged 3319 survivors 2381 acknowledged writes lost!
- Even when imposing writes to majority:
  - 6000 total 5700 acknowledged 5701 survivors 2 acknowledged writes lost! 3 unacknowledged writes found!
- In Version 3.4.1 all tests are passed (when using the right configuration with majority writes and linearizable reads) !!

Why Is Random Testing Effective for Partition Tolerance Bugs? (Majumdar & Niksic, 2018)

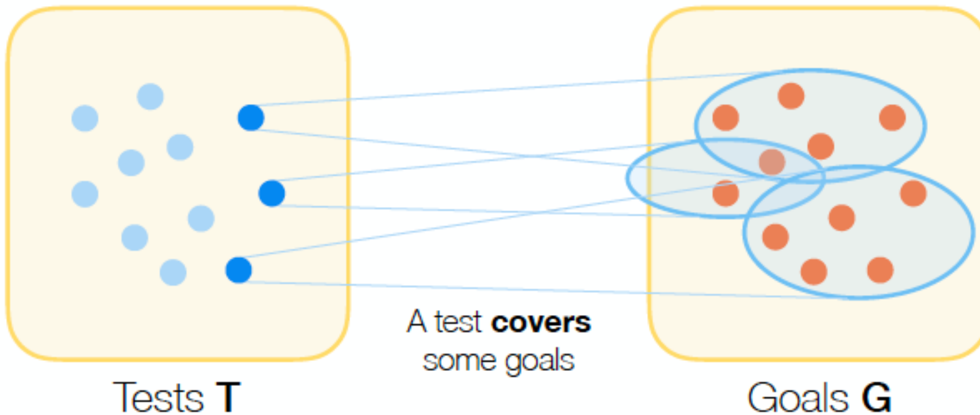
Coverage notions for network partitions:

- k-Splitting
  - Split network into k distinct blocks (typically k = 2 or k = 3)
- (k,l)-Separation
  - Split subsets of nodes with specific role
- Minority isolation
  - Constraints on number of nodes in a block (e.g. leader is in the smaller block of a partition)

With high probability, O(log n) random partitions simultaneously provide full coverage of partitioning schemes that incur typical bugs.

Why Is Random Testing Effective for Partition Tolerance Bugs? (Majumdar & Niksic, 2018)

Tests and goal coverage:



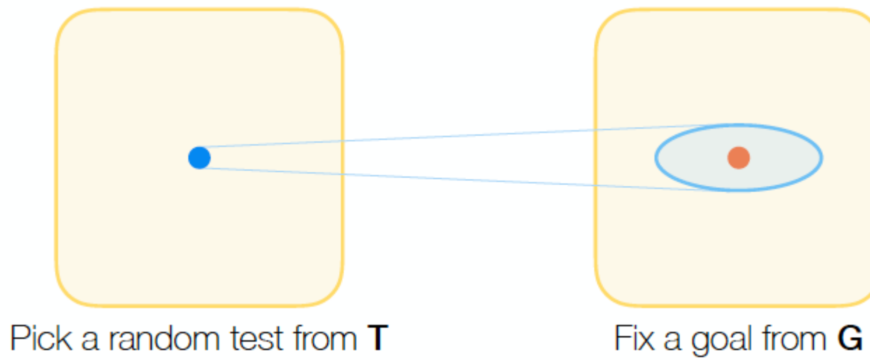
Covering family = Set of tests **cover all goals**

Small covering families = Efficient testing

(from Filip Niksic's presentation @ POPL'18)

Why Is Random Testing Effective for Partition Tolerance Bugs? (Majumdar & Niksic, 2018)

Random Testing



(from Filip Niksic's presentation @ POPL'18)

Why Is Random Testing Effective for Partition Tolerance Bugs?

- Let G be the set of goals and P[random T covers G] ≥ p
- Theorem: There exists a covering family of size $p^{-1} \log|G|$.
  - P[T random does not cover G] ≤ 1 – p
  - P[K independent T do not cover G] ≤ $(1 - p)^K$
  - P[K independent T are not a covering family] ≤ $|G|\ (1 - p)^K$

For $K = p^{-1} \log|G|$, this probability is strictly less than 1.
Therefore, there must exist K tests that are a covering family!

(from Filip Niksic's presentation @ POPL'18)

ChaosMonkey

Unleash a wild monkey with a weapon in your data center (or cloud region) to randomly shoot down instances and chew through cables[1]

- Built by Netflix in 2011 during their cloud migration
- Testing for fault-tolerance and quality of service in turbulent situations
- Random selection of instances in the **production environment** and deliberately put them out of service
  - Forces engineers to built resilient systems
  - Automation of recovery

[1] http://principlesofchaos.org

Principles of Chaos Engineering[2]

Discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production

- Focus on the measurable output of a system, rather than internal attributes of the system
  - Throughput, error rates, latency percentiles, etc.
- Prioritize disturbing events either by potential impact or estimated frequency.
  - Hardware failures (e.g. dying servers)
  - Software failures (e.g. malformed messages)
  - Non-failure events (e.g. spikes in traffic)
- Aim for authenticity by running on production system
  - But reduce negative impact by minimizing blast radius
- Automatize every step

[2] http://principlesofchaos.org

The Simian Army[3]

- Shutdown instance. Shuts down the instance using the EC2 API. The classic chaos monkey strategy.
- Block all network traffic. The instance is running, but cannot be reached via the network
- Detach all EBS volumes. The instance is running, but EBS disk I/O will fail.
- Burn-CPU. The instance will effectively have a much slower CPU.
- Burn-IO. The instance will effectively have a much slower disk.
- Fill Disk. This monkey writes a huge file to the root device, filling up the (typically relatively small) EC2 root disk.

[3] https://github.com/Netflix/SimianArmy/wiki/The-Chaos-Monkey-Army

The Simian Army (cont.)

- Kill Processes. This monkey kills any java or python programs it finds every second, simulating a faulty application, corrupted installation or faulty instance.
- Null-Route. This monkey null-routes the 10.0.0.0/8 network, which is used by the EC2 internal network. All EC2 <-> EC2 network traffic will fail.
- Fail DNS. This monkey uses iptables to block port 53 for TCP & UDP; those are the DNS traffic ports. This simulates a failure of your DNS servers.
- Network Corruption. This monkey corrupts a large fraction of network packets.
- Network Latency. This monkey introduces latency (1 second +- 50%) to all network packets.
- Network Loss. This monkey drops a fraction of all network packets.

Summary - Random Testing of Distributed Systems:

- A randomized testing method PCTCP with probabilistic guarantee
  - Generalizes PCT for multithreaded programs

- Jepsen testing framework
  - Random testing is effective for partition tolerance bugs

- ChaosMonkey
  - Failure testing on production environment