Concurrency Testing of the HotStuff Distributed Consensus Algorithm

Wenkai Li[1]

Supervisor(s): Burcu Külahçıoğlu Özkan[1]**, Ege Berkay Gülcan**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Wenkai Li
Final project course: CSE3000 Research Project
Thesis committee: Burcu Külahçıoğlu Özkan, Ege Berkay Gülcan, Johan Pouwelse

Abstract

Concurrency bugs are easy to introduce but difficult to detect, especially in implementations of distributed algorithms where concurrency nondeterminism is an inherent problem. These bugs may only be identified under very specific orderings of execution events, making them challenging to reproduce. Controlled concurrency testing techniques have been proposed to address the testing challenge, by taking over the scheduling of events, and effectively searching the state space of schedules to identify the concurrency bugs.

In this paper, we investigate and compare the performance of two concurrency exploration algorithms, probabilistic concurrency testing (PCT) and delay bounding strategy on our implementation of the HotStuff consensus protocol, which has been quite popular and was adopted by Meta for its Diem blockchain project. Our results suggest that can indeed find concurrency bugs more frequently in our HotStuff implementation than the baseline random strategy. However, using different bound parameters for PCT and delay bounding does not have significant effect on bug finding performance on our benchmarks.

1 Introduction

Imagine checking your bank account twice but seeing your balance suddenly plunge the second time — this is what could happen when concurrency bugs manifest. Ensuring the correctness of concurrent programs is a challenging task. Concurrency can introduce non-deterministic behaviours in programs and result in concurrency bugs. It can be very difficult to identify, reproduce and fix the bugs in concurrent programs.

Concurrency non-determinism is inherent in the context of distributed systems. The execution order of different processes is arbitrary, and messages sent over the network can be delayed and reordered. Distributed consensus algorithms are therefore essential for distributed server nodes to achieve agreements despite unpredictable network conditions or even when some nodes can fail.

HotStuff [1] has been a popular Byzantine fault tolerant (BFT) consensus algorithm in recent years, and is the basis of the DiemBFT [2] algorithm that Meta adopted for its crypotcurrency project Diem. HotStuff addresses the scalability challenges in past algorithms due to high communication complexity like the seminal PBFT [3], and achieves linear complexity in optimistic cases. It also offers optimistic response time that depends on the actual network delay, rather than a theoretical upper bound in Tendermint [4]. More importantly, HotStuff is the first algorithm that decouples the safety and liveness property on the algorithmic level, offering high flexibility in customizing the liveness conditions according to different applications.

Controlled concurrency testing (CCT) [5] techniques have been proposed to address the challenge of effectively finding concurrency bugs. This approach takes over the scheduling of threads, and then explores the possible interleavings of events for buggy schedules. However, the number of possible interleavings could be very large, posing a new challenge of effective schedule exploration.

Research on the characteristics of concurrency bugs provides an important observation that most bugs can be found with just a small number of ordering constraints [6] or context switches [7]. This critical observation has given rise to effective schedule exploration strategies like probabilistic concurrency testing (PCT) [8] and delay bounding [9]. PCT bounds the minimum number of reorderings required to find the bug, and delay bounding bounds the number of a schedule deviates from the decision of a deterministic scheduler. Both strategies can explore a small subset of possible schedule where concurrency bugs are most likely to manifest, and thus can be more effective than systematic or naive random strategy.

Previous work [10] has examined the performance of different concurrency exploration techniques including PCT and delay bounding on finding concurrency bugs in implementations of the Paxos [11] and Raft [12] algorithm. This paper aims to expand such effort to examining the performance of PCT and delay bounding strategies on finding concurrency bugs in our HotStuff implementation.

The main questions that this research aims to investigate and answer are as follows:

RQ1 Can PCT and delay bounding strategy find bugs more frequently in our HotStuff implementation, compared to the baseline random scheduler?

RQ2 Which bound parameter gives the best performance in PCT and delay bounding?

The paper will be structured as follows. In section 2, we will provide further details on relevant CCT techniques and the HotStuff algorithm. In section 3, we will introduce the experimental setup for evaluating the CCT techniques and report our results. Section 5 presents our considerations on the ethical and reproducibility aspect of this research. Discussion of the results will be in section 4. We conclude our research and suggest possible directions of future work in section 6.

2 Methodology

In this section, we will provide more details on the techniques and algorithms used to answer our research question. We will further explain controlled concurrency testing strategies in section 2.1 and the HotStuff consensus algorithm in section 2.2.

2.1 Controlled Concurrency Testing

Controlled concurrency testing aims to control the nondeterminism in typical system schedulers, and generate deterministic and reproducible schedules.

Systematic Scheduling

With systematic concurrency testing (SCT) [13; 14], a program is tested repeatedly where each test iteration explores a different schedule of executions, until all schedules haven been explored. This method is very effective and gives no false-positives, but it does pose two major challenges. First,

the state space of interleavings grows exponentially with the number of executions and makes it often infeasible to explore all schedules. Second, many schedules are equivalent to each other in terms of bug finding (e.g., re-ordering of two trivial events that are not relevant to exposing a bug).

Randomized Scheduling

Randomized exploration strategies are proposed to address the first problem of state space explosion. A subset of interleavings is randomly sampled and explored, rather than all possible ones.

Random walk (RW) strategy is a straightforward approach. At each scheduling point, it randomly chooses a next operation to execute. Despite its simple idea, it has been shown to be quite effective and is therefore a good baseline strategy to be compared with other techniques [5].

```
   ---- Thread A ----
1
2
3
4
5  assert(x==0)
```

```
   ------ Thread B ------
1  exec(1)
2  ...
3  exec(k-1)
4  x = 1
5
```

Figure 1: An execution where RW provides much worse probabilistic guarantee than PCT due to irrelevant homogeneous schedules. Variable x is initialized to 0. Example taken from [15].

However, random walk strategy can be ineffective when there are many homogeneous schedules (that do not expose a bug). Consider the execution in Figure 1. In order for the assertion to fail and to catch the bug, a scheduler needs to schedule all k operations in thread B before thread A, each has $1/2$ of the chance to be picked at a scheduling point. Therefore, there is only a probability of $1/2^k$ that RW will explore the buggy schedule in this scenario.

Probabilistic concurrency testing (PCT) [8] is a popular randomized scheduler that addresses the problem of too many equivalent schedules. It relies on the observation that most concurrency bugs are only caused by a small number of reordered events [6]. The intuition is that only these few events need to be scheduled correctly to find a bug, while the numerous possible schedules of other irrelevant events do not need to be explored, thus increasing the probability of hitting the bug. The minimum number of ordering constraints for a bug to manifest is defined as *bug depth* d in PCT.

PCT assigns a unique priority to each thread, and chooses the thread with the highest priority at each scheduling point. It also randomly decides $d-1$ priority change points where the executing thread will be given a lower priority. In an execution with n threads and k steps, PCT can detect concurrency bugs of depth d with a probability of at least $1/nk^{d-1}$. To find the bug in Figure 1, we only need to schedule the assertion in thread A after the assignment in thread B. The bug is therefore only of depth 1 and PCT gives a probabilistic guarantee of $1/2$, much stronger than RW.

Schedule Bounding

With the intuition that most concurrency bugs are due to only a few reorderings, PCT effectively reduces the search space to a small subset of schedules with high probability of finding a bug. Many other strategy have been proposed using similar intuition [7; 16] to put an effective bound on schedules using different parameters. Delay bounding is one such technique [9] that can be applied to various deterministic schedulers.

Delay means blocking a thread at scheduling point, and giving execution to the next thread. This can also be seen as a deviation from a given deterministic schedule, and delay introduces nondeterminism into an otherwise deterministic scheduler. Delay bounding technique uses the total number of such delays to bound the search space of a scheduler. In an execution of k steps and a delay bound of d, the search space can be bound to k^d.

2.2 HotStuff Consensus Algorithm

HotStuff aims to achieve consensus among a set of decentralized nodes. It guarantees correctness when there are $n \geq 3f+1$ nodes, where f is the number of nodes that can exhibit Byzantine faults.

In the HotStuff algorithm, each server node stores a tree of *blocks*. Each block contains a command requested by client, a reference to its parent block and other bookkeeping variables. A consensus is reached when a block is committed. To commit a block, the leader of the current *view* needs to collect the majority vote from a quorum of $n-f$ nodes (known as a *quorum certificate*, or *QC*) in three phases: Prepare, Pre-Commit, and Commit. The leader then broadcasts the decision in the Decide phase. Execution of the HotStuff protocol produces a monotonically growing chain of blocks. A typical execution of the multi-phased Basic HotStuff (Algorithm 2 in [1]) is illustrated in Figure 2.

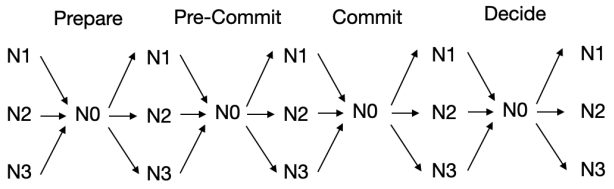

Figure 2: Execution of multi-phased Basic HotStuff protocol.

Since each phase of the Basic HotStuff is doing very similar work, it can then be optimized similar to a CPU pipeline. When a leader has collected enough Prepare votes and produced a quorum certificate in a view, it can relay the QC to the leader in the next view, delegating the responsibility of the Pre-Commit phase. The next leader does not actually start a Pre-Commit phase, however, but instead initiates another Prepare phase and make its own proposal. This Prepare phase simultaneously serves as the Pre-Commit phase for the previous view. Therefore, when a block carries a QC that refers to its direct parent, we know that the parent has completed its Prepare phase. This is called a *One-Chain*. If the direct parent also carries a QC referring to its direct parent, this forms a *Two-Chain* and implicates that the direct grandparent has completed its Pre-Commit phase. Finally, a block becomes a committed decision when a *Three-Chain* has formed. Execution of the Chained HotStuff algorithm (Algorithm 3 in [1])

is illustrated in Figure 3. The block of $QC3$ forms a Three-Chain and the block of $cmd1$ can be committed.

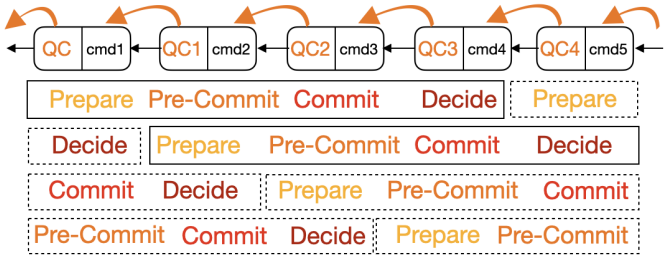

Figure 3: Execution of pipelined Chained HotStuff protocol.

3 Experimental Setup and Results

For the experiment, we implemented the event-driven HotStuff (see Algorithm 4 in [1], cryptographic component is omitted for simplicity) in C#, using the Coyote Actor framework v1.7.8. The event-driven version is based on the Chained HotStuff, which is then further simplified and has its liveness mechanism decoupled into the Pacemaker class. The Pacemaker is implemented with a predefined rotating leader scheme (shared among all nodes) in Round-Robin style. Two additional classes are implemented for testing: a client that sends requests and waits for responses, a cluster manager that coordinates communications with the client and among the servers.

3.1 Correctness Specification

The correctness of a consensus algorithm is specified with two properties: agreement and termination. Agreement is a safety property, specifying that no two correct nodes decide different values. Termination is a liveness property, specifying that every correct node eventually decides a value.

Safety and liveness specification of the HotStuff implementation is implemented with the Monitor API in Coyote library. The SafetyMonitor specifies that one client request cannot be decided in two different blocks and reports a bug upon violation. It is invoked each time the cluster manager receives a decision from the servers. The LivenessMonitor checks that client eventually receives responses to all the requests they have sent, and reports a bug in case of insufficient responses. Both monitors are registered to Coyote runtime during test initialization.

Note that liveness checking is conceptually very difficult, if not impossible [17], since it can be impossible to tell whether a message will not arrive or will only arrive after a very long delay. Liveness checking is however possible in a controlled execution environment like Coyote, because delays are not unbounded. The liveness violation threshold is sufficiently long (50000 steps) for any correct execution to complete (1200 steps, with 4 servers and 1 client sending 10 requests), and there are no heavy and long operations that could cause the starvation of other threads in case of an unfair scheduler.

3.2 Performance Measurement and Data Collection

Performance of concurrency exploration techniques are measured in two ways. First, we measure how frequently a strategy can find the bug, more precisely, the number of test iterations that a strategy identifies the bug out of 1000. Second, we measure how quickly a strategy can hit the bug. For this, we record the elapsed time from test start to the first bug found (if there is one) for each iteration, and compute an average.

Test results are collected through additional logging due to limitations of the Coyote library. For the safety property, a timer is started with each test iteration, and the elapsed time (in milliseconds) is written to a text file when the SafetyMonitor reports a violation for the first time in that iteration. Therefore, the number of lines in the text file represents the number of iterations that identified a bug, and each line represents the time it took to find the bug in that iteration. For the liveness property, a new line is written to another text file when an iteration runs to completion. Therefore, the number of violations is equal to the number of total iterations minus the number of lines in the log file.

3.3 Test Setup

Experiments are performed on macOS 13.2 with a 12 core Apple Silicon CPU. Before running the tests, Coyote needs to be invoked to rewrite the compiled binary to inject hooks for controlling the concurrency.

Tests involve six mock objects: a client, four servers running our HotStuff implementation, and a cluster manager that coordinates communications with the client and among the servers. Tests are run with default number of max steps of 10000 unless otherwise specified. A correct implementation will not hit this bound with a reasonable number of requests.

For each test iteration, the client sends one or more requests to the cluster manager. The cluster manager enqueues the requests, and broadcasts one to the servers to kick-start the consensus process. The leader sends the quorum certificate to the cluster manager after it has collected enough votes. The cluster manager will then broadcast this QC along with the next client request. After a consensus is reached (i.e. a block is committed), the decision is sent to the cluster manager and forwarded to the client.

3.4 Strategies and Bugs

We measure the performance of the following concurrency exploration strategies implemented in Coyote. Tests are run with default parameters unless otherwise specified.

S1 **Random Walk (RW)**. It picks a random operation at each scheduling point and is used as a baseline for comparison [5].

S2 **PCT**. Coyote's implementation of the PCT algorithm in [8]. Default depth is set to 10, an empirically best parameter suggested in [10].

S3 **Fair-PCT (F-PCT)**. Same as above, but falls back to Random Walk strategy when exceeding the maximum number of unfair scheduling steps, and thus avoids starvation of some threads.

S4 **Delay-Bounding (DB)**. Coyote's implementation of the Delay-Bounding algorithm in [9]. Default bound is set to 10, also an empirically best parameter suggested in [10]..

S5 **Fair-Delay-Bounding (FDB)**. Same as above, but falls back to Random Walk strategy when exceeding the maximum number of unfair scheduling steps, and thus avoids starvation of some thread.

In order to benchmark the performance of different exploration strategies, we seeded the following concurrency bugs into our implementation. Only one concurrency bug is present in the implementation in any test iteration.

B1 **Safety violation**. This bug is due to the lack of deduplication of client requests combined with an incorrect implementation of re-sending client requests. The client tries to re-send a request too eagerly, which can result in servers sometimes receiving and committing duplicate requests. This violates the SafetyMonitor since duplicate requests are committed in different blocks.

B2 **Liveness violation**. This bug is caused by a specific ordering of events. During test initialization, a message is sent to the cluster manager to kickstart the consensus process. If this message arrives earlier than client requests, the cluster manager has no client request to broadcast to servers to start the consensus. But when the client requests finally arrive, the cluster manager cannot kickstart the consensus and will just freeze, since the message is only sent once.

B3 **Liveness violation**. This bug has a similar root cause as B2, but is much harder to be caught by the schedulers. The bug manifests after a first round of quorum certificate has been produced and sent to the cluster manager, but the cluster manager does not receive further client requests to start the next round of consensus. It also freezes and does not progress when client requests come in the future. This bug likely has a higher depth, which could explain why it is much harder to find.

3.5 Results

We first examine how frequently can PCT and delay bounding technique find a concurrency bug. The experiments are run with 4 servers and 1 client. Client sends 1 request in B1 (the result is more distinct for different strategies) and 10 requests in B2 and B3. (Fair-)PCT and (Fair-)Delay Bounding uses the default bound of 10.

Table 1 shows the number of buggy schedules found by each strategy out of 1000 explorations. (Fair-)PCT strategies are the only ones that found all three concurrency bugs. Fair delay bounding strategy found B1 and B2 in almost every schedule, but fails to find a buggy schedule for B3. Therefore, both PCT and delay bounding can find concurrency bugs more frequently than the baseline random scheduler, and PCT is the most effective strategy here, being the only strategy that successfully finds B3.

We then try to examine what is the best bound parameter for PCT and delay bounding to find bugs in our HotStuff implementation. Based on the intuition that most concurrency bugs are caused by a small number of reorderings, only values smaller than the default 10 are experimented with.

	RW	PCT	F-PCT	DB	F-DB
B1	609	9	982	0	**1000**
B2	507	837	832	987	**990**
B3	0	9	**22**	0	0

Table 1: The number of buggy schedules out of 1000 explored

Table 2 shows the number of buggy schedules found by (fair) PCT strategies in B2 and B3, using different values of bug depth. Table 3 shows the number of buggy schedules found by (fair) delay bounding strategies in B2 and B3, using different values of the bound parameter.

The number of buggy schedules found using different bound parameters do not differ significantly from each other, nor do they exhibit any clear trend. It is therefore difficult to make any arguments from this data.

One explanation for the small difference between different bounds could be that our implementation and seeded bugs are not complicated enough, that they do not require a bound larger than 2 or even 1. Therefore, the different bounds for PCT and delay bounding do not make visible difference in terms of bug finding performance on our benchmarks.

d	1	2	3	4	6	8	10
B1-F	989	988	988	991	**995**	990	982
B2	842	831	839	**857**	846	831	837
B2-F	845	848	839	**850**	842	832	832
B3	18	16	20	13	14	**22**	9
B3-F	9	**26**	21	12	20	15	22

Table 2: The number of buggy schedules found out of 1000 explorations using different bug depth value for (fair) PCT. "-F" suffix indicates the fair version was used.

d	1	2	3	4	6	8	10
B1-F	1000	1000	1000	1000	1000	1000	1000
B2	**997**	996	996	994	995	992	987
B2-F	**998**	996	993	994	994	992	990
B3	0	0	0	0	0	0	0
B3-F	0	0	0	0	0	0	0

Table 3: The number of buggy schedules found out of 1000 explorations using different bounds for (fair) delay bounding. "-F" suffix indicates the fair version was used.

4 Discussion

Our results show that PCT and delay bounding strategy can indeed find bug more frequently than the baseline random scheduler. This is consistent with the previous experiments on other benchmarks [10; 5].

However, when exploring the best parameters for PCT and delay bounding, our experiment show no significant different for parameter ranging from 1 to 10. On the other hand, the

results in previous research are not very consistent either. [5] suggests the best parameter for PCT is $d = 3$, which is a reasonable number based on the intuition that most bugs are caused by small number of reorderings. [10] suggests they observed best result with $d = 10$, which does not correspond very well to the observed characteristics of concurrency bugs [6]. The best parameter may largely depend on the benchmarks, and thus can vary a lot in different scenarios.

5 Responsible Research

The research itself carries minimal ethical implications. Experiments on concurrency testing are conducted on a simple implementation of HotStuff consensus developed for this project, rather than a production one. Therefore, the usual concern of finding and reporting bugs in a critical real world system is not a problem in this research.

Reproducibility is also not an issue for this research, although the topic of concurrency and nondeterminism could leave a different impression. Since we are investigating *controlled* concurrency testing techniques, the nondeterministic schedules are well controlled by the scheduler, and turned into serialized, deterministic and fully reproducible schedules. Therefore, it is actually quite easy and straightforward to reproduce the results in this research.

Furthermore, we will take the following approaches to provide all the necessary details to reproduce our experiments and results. First, the experimental setup and parameters, especially the seed values for random schedule generation, used in the experiments will be reported in full detail. Second, all of the code in the research, including our HotStuff implementation, test setup and the seeded bugs, will be published to a public repository. With these two measures, we hope that everyone would be able to replicate our experiments and results in the exact same setup.

6 Conclusions and Future Work

In this paper, we experimented with two controlled concurrency testing techniques, PCT and delay bounding, investigated whether they can find bugs more frequently than the baseline random scheduler in our implementation of the HotStuff consensus algorithm, and explored what is the best bound parameter for the two strategies. To conduct the experiment, we implemented the popular HotStuff algorithm using Coyote framework, and seeded several concurrency bugs into the implementation as test benchmarks.

Our experiments show that PCT and delay bounding strategies can indeed find buggy schedules more frequently in our HotStuff implementation compared to the random walk strategy. In particular, one of the seeded bugs is only found by PCT but not the other two strategies. It is therefore the most effective exploration strategy on our benchmarks.

On the other hand, using different bound parameters does not make a significant difference on the number of buggy schedules found in our HotStuff implementation. This is likely due to the characteristic of the bugs we seeded in the implementation, such that they are not very sensitive to the bounding parameters.

Seeding concurrency bugs is one major issue of this research. The HotStuff consensus provides a very strong safety guarantee despite its simplicity. Coyote's event driven Task Asynchronous programming model also eliminates some sources of common concurrency bugs like synchronization. Therefore, the seeded bugs are somewhat limited in variety and may not be very good test benches that can provide insightful comparison among different strategies.

For the purpose of benchmarking concurrency testing strategies, it may be of interest for future work to test them on lower level implementations like the SCTBench [5] where more concurrency bugs may be exposed. It may also be of interest to experiment the exploration strategies on sophisticated production systems, where the concurrency bugs may have more diverse characteristics and are more suitable to evaluate the performance of using different bound parameters.

References

[1] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pp. 347–356, 2019.

[2] T. D. Team, "Diembft v4: State machine replication in the diem blockchain," 2021.

[3] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OsDI*, vol. 99, pp. 173–186, 1999.

[4] E. Buchman, *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[5] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using controlled schedulers: An empirical study," *ACM Transactions on Parallel Computing (TOPC)*, vol. 2, no. 4, pp. 1–37, 2016.

[6] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pp. 329–339, 2008.

[7] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 446–455, 2007.

[8] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 167–178, 2010.

[9] M. Emmi, S. Qadeer, and Z. Rakamarić, "Delay-bounded scheduling," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, pp. 411–422, 2011.

[10] P. Deligiannis, A. Senthilnathan, F. Nayyar, C. Lovett, and A. Lal, "Industrial-strength controlled concurrency

testing for c# programs with coyote," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 433–452, 2023.

[11] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[12] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 305–319, 2014.

[13] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs.," in *OSDI*, vol. 8, 2008.

[14] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 221–230, 2011.

[15] X. Yuan, J. Yang, and R. Gu, "Partial order aware concurrency sampling," in *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, pp. 317–335, Springer, 2018.

[16] M. Musuvathi and S. Qadeer, "Partial-order reduction for context-bounded state exploration," tech. rep., Tech. Rep. MSR-TR-2007-12, Microsoft Research, 2007.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.