FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems

Jeffrey F. Lukman\textsuperscript{1}, Huan Ke\textsuperscript{1}, Cesar A. Stuardo\textsuperscript{1}, Riza O. Suminto\textsuperscript{1}, Daniar H. Kurniawan\textsuperscript{1}, Dikaimin Simon\textsuperscript{2}, Satria Priambada\textsuperscript{3}, Chen Tian\textsuperscript{4}, Feng Ye\textsuperscript{4}, Tanakorn Leesatapornwongsa\textsuperscript{5}, Aarti Gupta\textsuperscript{6}, Shan Lu\textsuperscript{1}, and Haryadi S. Gunawi\textsuperscript{1}

\textsuperscript{1}University of Chicago, \textsuperscript{2}Surya University, \textsuperscript{3}Bandung Institute of Technology, \textsuperscript{4}Huawei US R\&D Center, \textsuperscript{5}Samsung Research America, and \textsuperscript{6}Princeton University

Abstract

We present a fast and scalable testing approach for datacenter/cloud systems such as Cassandra, Hadoop, Spark, and ZooKeeper. The uniqueness of our approach is in its ability to overcome the path/state-space explosion problem in testing workloads with complex interleavings of messages and faults. We introduce three powerful algorithms: state symmetry, event independence, and parallel flips, which collectively makes our approach on average $16 \times$ (up to $78 \times$) faster than other state-of-the-art solutions. We have integrated our techniques with 8 popular datacenter systems, successfully reproduced 12 old bugs, and found 10 new bugs — all were done without random walks or manual checkpoints.

1 Introduction

Datacenter systems such as distributed key-value stores, scalable file systems, data-parallel computing frameworks, and distributed synchronization services, are the backbone engines of modern clouds, but their complexities and intricacies make them hard to get right. Among all types of issues in such systems, complex interleavings of messages, crashes, and reboots are among the most troublesome\cite{39, 50, 61, 62, 79}. Such a non-deterministic order of events across multiple nodes cause “distributed concurrency” bugs to surface (or “DC bugs” for short). Developers deal with DC issues on a monthly basis\cite{45, 48}, or worse on a weekly basis for newly developed protocols\cite{6}. They are hard to reproduce and diagnose (take weeks to months to fix the majority) and lead to harmful consequences including whole-cluster unavailability, data loss/inconsistency, and failed operations\cite{59}.

Ideally, bugs should be unearthed in testing, not in deployment\cite{35}. One systematic testing technique that fits the bill is stateless/software model checking that runs directly on implementation-level distributed systems\cite{46, 49, 55, 58, 73, 80, 81}. These software model checkers\cite{1} attempt to exercise many possible interleavings of non-deterministic events such as messages and fault timings, hereby pushing the target system into unexplored states and potentially revealing hard-to-find bugs.

One nemesis of checkers is the path explosion problem. As an illustration, given 10 concurrent messages (events)\{a, b, \ldots, j\}, a naive checker such as depth-first search (DFS) has to exercise $10!$ (factorial) unique execution paths (ab..ij, ab..ji, and so on). Figure 1 illustrates further this explosion problem. The gray “DFS” bar shows almost 100 paths to explore (in y-axis) under a simple workload such as an instantiation of a Raft update protocol (“Raft-1”)\cite{67}.

To tame this problem, checkers employ path reduction algorithms. For example, MoDIST\cite{81} and some others\cite{73, 80} adapted the popular concept of Dynamic Partial Order Reduction (DPOR)\cite{38, 41}, for example “a message to be processed by a given node is independent of other concurrent messages destined to other nodes [hence, need not to be interleaved].” SAMC\cite{58} also extended DPOR further. As a result, reductions significantly improve upon a naive DFS method, as shown by the “mDPOR” and “SAMC” bars on Raft-1 in Figure 1.

1In this paper, “checkers” specifically represent the distributed system software model checkers, as cited above, not including “local” thread-scheduling checkers\cite{22, 66} or the classical model checkers\cite{37, 42}.
Despite these early successes, we found that the path explosion problem remains untamed under more complex workloads. For example, under two or three concurrent Raft updates (Raft-2 and -3 workloads in Figure 1), the number of paths to explore still increases significantly in MODIST and SAMC. Not to mention a much more complex workload such as Paxos [57] where the path explosion is larger (e.g., Paxos-1 to -3 workloads in Figure 1).

To sum up, existing checkers fail to scale under more complex distributed workloads. Yet in reality, some real-world bugs are still hidden behind complex interleavings (§7). For example, the Paxos bug in Cassandra in Figure 2 can only surface under a workload with three concurrent updates with 54 events in total. These kinds of bugs will take weeks to surface with existing checkers, wasting testing compute resources and delaying bug finding and fixes. For all the reasons above, to find DC bugs, some checkers mix their algorithms with random walks [81] or manual checkpoints [49], hoping to faster reach “interesting” interleavings that would lead to DC bugs. However, this approach becomes unsystematic – the random and manual approaches lead to poorer coverage than a systematic coverage of all states relevant to observable events.

We present FLYMC, a fast, scalable, and systematic software/stateless model checker that covers all states relevant to observable events for testing distributed systems implementations. FLYMC achieves scalability by leveraging the internal properties of distributed systems as we illustrate below with three FLYMC’s algorithms.

(1) Communication and state symmetry: Common in cloud systems, many nodes have the same role (e.g., follower nodes, data nodes). The state transitions of such symmetrical nodes usually depend solely on the order and content of messages, irrespective of the node IDs/addresses. Thus, FLYMC reduces different paths that represent the same symmetrical communication or state transition into a single path.

(2) Event independence: While state symmetry significantly omits symmetrical paths, many events must still be permuted within the non-symmetrical paths. FLYMC is able to identify a large number of event independencies that can be leveraged to alleviate a wasteful reordering. For example, FLYMC automatically marks concurrent messages that update disjoint sets of variables as independent. FLYMC can also find independence among crash-related events.

(3) Parallel flips: While the prior methods reduce message interleavings to every node, in aggregate many flips (reordering of events) must still be done across all the nodes. The problem is that in existing checkers, only one pair of events is flipped (reordered) at a time. To speed this up, parallel flips perform simultaneous reorderings of concurrent messages across different nodes to quickly reach hard-to-reach corner cases.

Finally, not only path reduction but wall-clock speed also matters. Existing checkers must wait a non-negligible amount of time in between every pair of enabled events for some correctness and functionality purposes. The wait time is reasonable under simple workloads, but it significantly hurts the aggregate testing time of complex workloads. FLYMC optimizes this design with local ordering enforcement and state transition caching which will be explained later (§5).

Collectively, the algorithms make FLYMC on average $16 \times$ (up to $78 \times$) faster than other state-of-the-art systematic and random-based approaches, and the design optimizations improve it to $28 \times$ (up to $158 \times$). FLYMC is integrated with 8 widely-used systems, the largest number of integration that we are aware of. We model checked 10 protocol implementations (Paxos, Raft, etc.), successfully reproduced 12 old bugs, and found 10 new DC bugs, all confirmed by the developers and all were done in a systematic way without random walks or manual checkpoints. Some of these bugs cannot be reached by prior checkers within a reasonable time. We have released our FLYMC publicly [20].

The following sections detail our four contributions:

1. Highly scalable checker algorithms that provide systematic state coverage for given workloads (§3).
2. A checker design that is backed with static analysis help developers extract information from the target system and use it to write the system-specific parts of the algorithms (§4).
3. Additional optimizations that improve the checker’s wall-clock speed in exploring paths (§5).
4. A comprehensive integration with challenging applications, and detailed evaluations that demonstrate the checker’s effectiveness. (§6-7).

For interested readers, we provide an in-depth technical report [21].
2 Background

• Checker architecture: The concept of checkers and how they work in detail can be found in existing checker literature [58, §2.1][81, §2] [73, §3][21, §2]. This section briefly discusses the important components and terms.

As shown in Figure 3, a checker runs the target workload (e.g., in nodes A and B) and intercepts all the in-flight messages (e.g., the concurrent messages a1, a2, b1, and b2 intercepted by the gray “hooks”) to control their timings. The checker’s server then enables one message event at a time (e.g., enable b1). The checker’s hooks wait for the target system to quiesce (after b1 is processed) and the node to pass the new resulting global state (e.g., S1) to the checker’s server which records it as the state-event history (e.g., S0+b1→S1).

The developers decide which global state variables to check (e.g., role, leader, ballot number). The server then runs assertions to find any safety violation in the new state.

New events generated by every enabled event will be intercepted again by the checker (e.g., a3 generated in response to b1, not shown in the figure). This whole process repeats (S0+b1→S1, S1+b2→S2, ...+a3→...) until it reaches a termination point — when a specification is violated or the workload ends without any violation (e.g., no more messages observed). This forms an explored path (e.g., b1b2...a3). A path implies a unique total ordering of events; it is also known as “trace” or “execution sequence” [44, 54, 81].

Given a previously exercised path, the checker will permute the possible interleavings (to-explore paths) and restart the workload. For example, in the next run, it will flip b2 before b1, hence executing a new path b2b1... within the same workload. Paths can also contain crash/reboot events; for example, a path b1b2B↑b2... implies a crash B and a reboot B↑ on node B are injected after b1 is processed but before b2 arrives. The whole test completes (exhausts the state space) if there are no more paths left to explore.

3 FLYMC Algorithms

By considering the properties of distributed systems, we equip FLYMC with two reduction algorithms: communication and state symmetry (§3.1) and event independence (§3.2); and one prioritization algorithm: parallel flips (§3.3).

The two reduction algorithms reduce unnecessary interleavings (redundant paths) that would lead to the same states already explored before. While the prioritization algorithm prioritizes interleavings that would reach corner cases faster.

Throughout this section, we describe each of the algorithms in the following format: (a) the specific path explosion issue being addressed, (b) the intuition for the reduction or prioritization, (c) the algorithm in a high-level description, and (d) a comparison to existing solutions. Later in Section 4, we discuss the intricacies of implementing these algorithms correctly and how our static analyses support can help developers in this regard.

3.1 Communication and State Symmetry

• PROBLEM: Let us imagine a simple communication in Figure 4a where message k triggers l, x triggers y, and k and x are messages of the same type (e.g., a write request). Figures 4b and 4c show two possible reordered paths klxy and xylk. While these paths seem to be different, their communication structures in Figures 4b-c hint at a possibility for symmetrical reduction.

A method to implement the symmetrical reduction in local concurrency literature is to abstract the system property [32, 33, 75]. Applying this to distributed systems, we initially attempted to abstract only the communication structure, specifically by abstracting the sender and destination node IDs (e.g., IP addresses) to a canonical receiving order; for example in Figure 4b, as node B is the first to receive, its node ID is abstracted to node “1” (e.g., kA→B becomes k2→1). Similarly in Figure 4c, as node A is the first to receive, its node ID is abstracted to node “1” (e.g., XB→A becomes x2→1), hence the two figures exhibit a communication symmetry as k and x are messages of the same type from node “2” to “1”.

Unfortunately, this approach is not always effective because most messages carry a unique content. For example, in Paxos, messages k and x carry different ballot numbers, hence cannot be treated the same. Thus, while the communication structures (the arrows) in Figures 4b and 4c look symmetrical, abstracting only the messages does not lead to a massive reduction.

• INTUITION: Fortunately, in many cloud systems, many nodes have the same role (e.g., follower nodes, data nodes) although their node IDs are different. Furthermore, the state transitions of such symmetrical nodes usually depend solely
on the order and content of the messages, irrespective of the sending/receiving node IDs.

To illustrate this, let us consider the two communication structures in Figures 5a-b, which represent the first phase of a (much simplified) Paxos implementation with two concurrent updates (solid and dashed lines). Node A broadcasts its prepare messages (the solid lines), a1 to itself and b1 to node B, with “1” representing a ballot number 1. Similarly, node B broadcasts b2 to itself and a2 to node A with ballot number 2 (dashed lines).

If we compare the two communication structures in Figures 5a-b, they are not symmetrical, unlike the previous example in Figure 4. But let’s analyze the state transition of every node, such as the highest ballot number the node has received, as shown in the middle table of Figure 5. In this Paxos example, every node only accepts a higher ballot and discards a new lower one, hence the node prepare status monotonically increases (e.g., 1 to 2). In the left ordering, b1a2b2a1 in Figure 5a, node A’s state transition is 00222 and B’s is 01122. In the ordering on the right, a1b2a2b1, the state transition is symmetrical (mirrored), 01122 in A and 00222 in B.

To sum up, while the two paths do not exhibit communication symmetry (Figures 5a-b), their state transitions are symmetrical (the middle table). Thus, state symmetry can be effective for path pruning (e.g., if b1a2b2a1 is already explored, then a1b2a2b1 is redundant).

- **Algorithm:** To implement symmetry, first, we keep a history of state-event transitions (§2) that have been exercised in the past, in the following format: S_i + e_j → S_j where “S” denotes the global state (i.e., collection of per-node states) and e_j is the next enabled event. So, when e_j is enabled, the global state transitions from state S_i to S_j.

  In addition, we keep a history of \{absState+absEv\} transitions where absState denotes the abstracted global state (in alphabetical order) that excludes the node IDs for symmetrical nodes such as datanodes (and similarly absEv for events). Using the example in Figure 5a, the first event will generate \{00+1\} where 00 represents the abstracted state of datanodes A and B (with just the highest ballot numbers, excluding the node IDs) and 1 represents the abstracted a1 message. Subsequently, we record \{01+2\}, \{12+2\}, and \{22+1\} to the history. Important to note that state 12 is from the alphabetically ordered state 21; that is, symmetry implementation requires alphabetical/numerical sorting.

  With this history, the second ordering a1b2a2b1 in Figure 5b will be marked symmetrical; when a1 is to be enabled (abstracted to +1) when the system is at state 00, a historical match \{00+1\} will be found. Similarly, for b2 (abstracted to +2) when the system is at state 01, a match \{01+2\} will be found. One caveat is that state symmetry works less effective in earlier paths as the history is still being built up, but after a few initial paths, the “cache hit rate” increases significantly (more in §5).

- **Comparison:** In classical (stateful) model checking, symmetry is commonly used, e.g., for symmetrical processors [32, 75]. In distributed checkers, we found none that employs symmetry [46, 49, 55, 58, 73, 80, 81], except SAMC [58]. However, SAMC only uses symmetry for reducing unnecessary crash timings, but not for concurrent messages. FLYM’s symmetry is more powerful as it also generalizes for crash timings. More specifically, a crash is abstracted as a crash event targeted to a particular node; for example, \{12+2\} implies a crash injected at the node with ballot number 2 (regardless of the datanode IDs).

### 3.2 Event Independence

- **Problem:** While state symmetry omits symmetrical paths, there are many other events to reorder within the non-symmetrical paths. For example, if four messages a1...a4 of different types are concurrent to node A, the permutation will lead to 4! times more paths. As some concurrent messages to every node must still be reordered, more reduction is needed.

- **Intuition:** In this context FLVM adapts the concept of DPOR’s “independence” (aka. commutativity) as mentioned in the introduction. In DPOR, two events e1 and e2 are independent if S_i+e1+e2→S_j and S_i+e2+e1→S_j. That is, if e1e2 or e2e1 result in the same global state transition from S_i to S_j, the pair of events e1 and e2 do not have to be flipped when the system is at S_i, hence reducing the number of paths to explore. An example of independence in distributed systems is when many concurrent messages (to a destination node) update different variables. For example, in some distributed systems such as ZooKeeper, the atomic broadcast protocol might be running concurrently with the leader election protocol (because of a crashed node), but some of the messages in these two protocols do not update the same variables (when the system is at a specific state S_i), hence it is not necessary to flip them.

- **Algorithm:** While the concept of DPOR/independence arose from stateful model checkers (with known state transitions) [29, 38, 41, 69], adapting it to stateless distributed checkers is not straightforward – how can a checker have
prior knowledge that $S_i + e_1 e_2$ and $+ e_2 e_1$ would lead to the same future state $S_i$ before exercising the events? For this, FLyMC helps developers identify disjoint updates ahead of time with the static analyses (more details in §3.1).

Essentially, for every message $n_i$ to a node $N$, our static analyses builds the live readSet and updateSet, a set of to-be-read and -updated variables, within the flow of processing $n_i$ at $N$’s current state. That is, our approach incorporates the fact that $n_i$’s read and update sets can change as node $N$ transitions across different states. Given such an information, two messages $n_i$ and $n_j$ to a node $N$ are marked independent if $n_i$’s readSet and updateSet do not overlap with $n_j$’s updateSet at the current state $S_i$, and vice versa. In addition, if the updateSets of two messages intersect completely and all the variables in the sets are in/decremented by one (e.g., a common acknowledgment increment “ack+” in distributed systems), then the two messages are marked independent/commutative.

Beyond reducing unnecessary message interleavings, a scalable checker must reduce unnecessary crash injections at different timings. 50% of DC bugs can only surface with at least one crash injection and 12% require at least two crash events at specific timings [59] (e.g., the complex ZooKeeper bug in Figure 6), which exacerbates further the path explosion problem (imagine different fault timings such as $...a_1A,...,a_1A,...,a_2A$ where “A” denotes crashing of node $A$). Thus, another uniqueness of FLyMC’s adaptation of independence is building the sets above for crash events. For example, if a follower node is crashed ($B$) and the leader node $A$ reacts by reducing the live-nodes count (e.g., $\text{liveNodes}--$), then $B$’s updateSet will include $A$’s $\text{liveNodes}$ variable.

**COMPARISON:** Prior checkers adopted DPOR’s independence, but only to a limited extent, hence are not scalable under complex interleavings. For example, MoDfS [81, §3.6], CrystalBall [80, §2.2] and dBug [73, §2] only adopted DPOR with the following rule: “a message to be processed by a given node is independent of other concurrent messages destined to other nodes”. But because they are black-box checkers that do not analyze the target source code, they cannot find more independencies. On the other hand, being a white-box checker, FLyMC exploits access to source code in today’s DevOps-based cloud development where developers are testers and vice versa [60].

![Parallel Flips](image)

**Figure 7: Parallel Flips.** Figures (a+b) and (c) are explained in the “Problem” and “Algorithm” segments of §3.3, respectively.

FLyMC is another example of a white-box checker, but developers need to manually analyze their target system code and follow the SAMC simple principles. Hence, SAMC only introduces cautious and rigid reduction algorithms which are less powerful than FLyMC’s (note that in FLyMC, the content of the sets mentioned above will be automatically constructed through the static analysis support). For example, FLyMC generalizes a discarded message ([58, §3.3.1]) as an empty updateSet, such a message automatically does not conflict with any other messages, hence not need to be reordered. As another example, a crash that does not lead to new messages (e.g., a quorum is still maintained after crashing a follower node $B$) will not be interleaved with all the outstanding messages [58, §3.3.2] as FLyMC automatically identifies that the crash event $B$’s updateSet (e.g., $\text{liveNodes}$ in the leader node) does not conflict with updateSets of in-flight messages.

### 3.3 Parallel Flips

**Problem:** While our previous methods reduce messages reordering to every node, in aggregate many flips must still be done across all the nodes. The problem is that in existing checkers, to create a new reordered path, only one pair of events is flipped at a time. For example, in Figure 7a, two concurrent messages $a_1$ and $a_2$ are in transit to node $A$ and four messages $b_1...b_4$ to node $B$. Figure 7b illustrates how existing approaches flip only one non-independent pair of events at a time; for example, after path #1 $a_1 a_2 b_1 b_2 b_3 b_4$, the next path #2 is created by sliding $b_1$ before $b_2$, then a subsequent path #3 with $b_1$ before $b_2$, and so on. Now, let us suppose that a bug is induced by the $a_2 a_1$ ordering (i.e., $a_2$ must be enabled before $a_1$). In the standard approach above, it will take 4! reorderings (of the four messages to $B$) before we have the chance to flip $a_2$ before $a_1$.
• **Intuition:** We observed such patterns when analyzing our bug benchmarks. For example, to hit the Paxos bug in Figure 2, Prepare#2 message must be enabled before Commit#1, both to node C, but there are 8 earlier in-flight messages to other nodes that must be reordered. Worse, after that, Prepare#3 message must arrive before Propose#2, but there are 5 earlier messages to flip. Thus, the bug-inducing flips are not exercised early.

This problem motivates us to introduce parallel flips. That is, rather than making one flip at a time, parallel flips of pairs of events are allowed. Parallel flips also bode well with a typical developers’ view that mature cloud systems are generally robust in the “first order” (under common interleavings) [27] but simultaneous “uncommon” interleavings across all the nodes may find bugs faster.

• **Algorithm:** For the next to-explore path, we flip a pair of two messages in every node, hence N simultaneous flips across all the N nodes (but we do not perform multiple flips within a node). For example, in Figure 7c, after executing path #1 a1a2b1...b4, in path #2 we make both a2a1 and b1b3 flips. This is permissible because the in-flight messages to node A are independent of those to node B (per our DPOR adoption in §3.2). If no parallel flips are possible, we revert back to single flips (e.g., only b1b2 flip in path #3).

We emphasize that parallel flips is a prioritization algorithm rather than a reduction algorithm. That is, this algorithm helps developers to unearth bugs faster but does not reduce the state space. Thus, in the implementation, FLYMC retains the single flips paths prior to the parallel flips into a set of low-priority-to-explore paths so that FLYMC stays systematic. Later when evaluating coverage completeness (§7.3), parallel flips are not included.

• **Comparison:** We are not aware of any distributed checkers that employ an algorithm such as parallel flips. However, in the software testing literature [31, 44], we found that our approach is in spirit similar to “branch flipping” where multiple branch constraints are flipped simultaneously to cover more corner cases faster.

4 FLYMC Static Analyses and Design Challenges

There are several challenges in applying FLYMC algorithms correctly in the context of distributed systems. First, we describe FLYMC static analyses that automatically extract the knowledge about the target system (§4.1). Next, we describe the challenges in applying the FLYMC algorithms (§4.2). Additional correctness sketches and pseudo-code are available in an anonymized technical report [1].

2In Section 3, we have described the algorithms in a way that is easy to understand. For correctness details, we apologize that we cannot fit the entire discussion in Section 4. We hope that our technical report addresses related questions that might arise.

4.1 Static Analyses Support

While FLYMC’s algorithms are generic, the details (e.g., the if-else predicates for reduction) are specific to a target system. Furthermore, the required predicates can become quite complex, which makes it harder for developers to derive them manually. For this reason, we provide static analyses support in FLYMC, which automatically builds the required predicates from simple annotations provided by the developers. For example, it automatically builds readSet, updateSet, sendSet and diskSet (§3, §4.2) containing variable names specific to the target system implementation. Below we describe the input and output formats.

• **Input (Annotation):** To use FLYMC’s static analyses, developers only need to annotate a few data structures: (a) node states, (b) messages, and (c) crash handling paths. Annotating node states that matter (e.g., ballot, key, value) is a common practice [68, 81], for example:

```java
public class Commit {
    ...
    @FlyMCNodeState
    public final UUID ballot;
    ...
}
```

Annotating message class declarations such as “MessageIn” in Cassandra is relatively simple (note that we only need to annotate the class declarations, but not every instantiation, hence a light annotation). Crash handling paths are typically in the catch blocks of failed network IOs, for example:

```java
try{
    binaryOutput.writeRecord(quorumPacket, ...);
    ...
} catch { @crashHandlingPath ...
}
```

In addition, our static analyses also maintains a dictionary of disk IO library calls. On average, the annotation is only 19 LOC per target system that we have studied.

• **Output:** The output of the analysis is all the variable sets mentioned above, along with the symbolic paths. For example, for Cassandra, the analysis outputs are as follows:

```java
(A) if (m.type == "PROP" && m.ballot > n.ballot)
    updateSet = n.key, n.value, n.ballot
    readSet = n.ballot
(B) if (m.type == "PROP_RESP" && m.resp == true &&
     n.proposeCounter < majority)
    updateSet = n.proposeCounter
    readSet = --
(C) ...
```

With this output, we can track the relationship of every two concurrent messages n1 to n2 to node N. For example,
if both messages are type \( \lambda \), they conflict with each other and must be reordered. However, if \( n_i \) is of type \( \lambda \) while \( n_j \) is of type \( \beta \), they exhibit disjoint updates and therefore, do not need to be reordered (but they are independent only at specific states that satisfy the if-predicates above).

To create such an output given the input annotation, our static analyses performs basic data- and control-flow analyses. The detailed steps and pseudo-code are presented in [21, §4.2]. Our static analyses do not cover multi-variable correlation and pointer/heap analysis (as not needed in our target cloud systems so far; e.g., a message simply arrives, gets processed, and then is deallocated).

### 4.2 Design Challenges

- **State Symmetry**: In reality, not only one variable (e.g., ballot number) is included in the abstracted state, which then raises the question of which variables should be included/excluded in the abstracted information. For example, if the protocol processes the sender IDs (node addresses) of the messages, then excluding sender IDs from the abstracted event is not safe, as this can incorrectly skip unique event reorderings. Thus, for state symmetry, our static analyses outputs a list of message variables that state transitions depend on, hence cannot be abstracted (excluded). For example, for Cassandra Paxos, neither the sender nor destination datanode IDs are used by the protocol, hence can be safely excluded from the abstracted information.

- **Event Independence**: We address two challenges in implementing event independence.

  First, as we target storage-backed distributed systems, two messages, \( n_i \) and \( n_j \) to node \( N \), might modify two different variables that perhaps will eventually be logged to the same on-disk file. It is not safe to consider them independent as the same file is updated but potentially in different orders. Thus, the two disjoint messages are truly independent only if they are not logged to the same file, which our static analyses tracks (§4.1).

  The second challenge is similar but more subtle. In distributed settings, reordering of messages to one node cannot be seen as a local impact only, as an arriving message can trigger new messages. This non-local impact must be put into consideration. For example, let us consider two messages \( a_1 \) and \( a_2 \) concurrently arrive at node \( A \) whose local state is \( \{ x=0, y=0 \} \). Now, let us suppose \( a_1 \) makes \( x=1 \) and \( a_2 \) makes \( y=2 \). Here, the two messages seem to be disjoint. However, if after processing each message, node \( A \) sends its state \( \{ x, y \} \) to other nodes (e.g., \( B \)), then the two messages are actually not independent. Making them independent would lead to an unsafe reduction. Let us consider the following sequence:

  1. \( A \)'s state is \( x=0, y=0 \)
  2. \( A \) receives \( a_1 \), so \( x=1 \)
  3. \( A \) sends \( x=1, y=0 \) to \( B \)
  4. \( A \) receives \( a_2 \), hence \( y=2 \)
  5. \( A \) sends \( x=1, y=2 \) to \( B \)

  The example scenario shows \( a_1 \) is enabled before \( a_2 \), (2) before (4). If we (incorrectly) declare them as independent, \( a_2a_1 \) ordering will be skipped, thus we will never see \( \{ y=2, x=0 \} \) sent to \( B \). If node \( B \) has a logic such as “if (\( y=2 \) & \( x=0 \)) panic()”, then we will miss this \( a_2a_1 \)-induced bug. For this reason, in addition to `readSet` and `updateSet`, we keep track of the `sendSet` (§4.1), the variables that are sent out after a message is processed. In the example, because \( a_1 \)'s and \( a_2 \)'s `sendSets` overlap with their `updateSets` (i.e., \( x, y \)), \( a_1 \) and \( a_2 \) are not independent.

- **Parallel Flips**: We only allow parallel flips if none of the events within the flips are causally-dependent on one another (i.e., exhibit a happens-before relationship). For example, let us consider \( a_1a_2b_1b_2 \) in the first case on the right figure, where \( b_1 \) is causally-dependent on (happens after) \( a_1 \), and \( b_2 \) on \( a_2 \). If we carelessly make the two flips to \( a_2a_1b_2b_1 \), it is possible that \( b_1 \) will never happen (as shown in the lower figure) because the new ordering \( a_2a_1 \) that node \( A \) receives does not generate \( b_1 \). In this case, this new path will make FlyMC hang. Thus for correctness, FlyMC’s parallel flips are backed with happens-before analysis via vector clocks [21, §6.1].

  We would like to emphasize that we have provided sketches for the correctness claims of our reduction algorithms based on symmetry and independence [21, §4.1]. However, parallel flips do not provide a sound reduction of state space in general, and it is tricky to identify conditions where they might, as also described in [21, §4.1.3]. Thus, we treat parallel flips as a prioritization heuristic to reach buggy states earlier, but it is based on a well-grounded intuition (§3.3) as opposed to randomness or manual checkpoints.

### 5 FlyMC Design Optimizations

In wall-clock time, an execution of one path can take 8-40 seconds [58, 73]. Since one of the checker’s goals is to quickly unearth bugs, thus, per-path wall-clock speed matters. Below we describe our solutions to the bottlenecks.

- **Local Ordering Enforcement**: In a path execution, stand-alone checkers that intercept events at the application layer with “hooks” (e.g., dBug [73], SAMC [58], FlyMC) must wait a non-negligible amount of time before enabling the next event, for two purposes: to prevent concurrency issues within itself and to wait for new updated state from the target system.

  To illustrate the former, consider two concurrent incoming messages \( a_1 \) and \( a_2 \) to node \( A \), and the checker’s server decides to enable \( a_1 \) then \( a_2 \). If the wait time is removed between the two actions, the probability that node \( A \) accidentally processes \( a_2 \) before \( a_1 \) increases. This is because `enable(a1)` and `enable(a2)` actions themselves are concurrent.
rent messages from the checker’s server to node A whose timings are not controlled. This wait time is too expensive for such rare cases. To remove it, we enhance FlyMC’s interposition mechanism at the target system side to enforce local action ordering; for example, enabling $a_2$ includes information about the previous enabled event, $a_1$, such that at node A’s side, $a_2$ waits for $a_1$, if needed.

**State-event caching:** Although the wait time has been removed, FlyMC must perform “history tracking” for collecting past state-event transitions $S_1 + e_j \rightarrow S_j$. Here, after enabling $e_j$ and before enabling the next event $e_k$, FlyMC must collect $S_j$ from the target system, another expensive round-trip time. To optimize this, we use the state-event history as a cache. That is, if $e_j$ is to be enabled at $S_1$ and the state transition $S_1 + e_j$ already exists in the history, then no wait is needed between enabling $e_j$ and $e_k$. In this case, FlyMC will automatically change its view state of the target system to $S_j$. This strategy is highly effective; the “cache hit rate” reaches 90% quickly after 35 paths explored.

In summary, a checker itself is a complex system with many opportunities for optimization. Our optimizations above successfully deliver further speed-ups to quickly find DC bugs (§7.1). For interested readers, in [21, §5], we describe further explanation regarding the latency analysis of various types of checkers (OS-supported, runtime-supported, and stand-alone checkers).

### 6 Implementation and Integration

FlyMC is implemented in around 10 KLOC (including mechanisms such as fault injection, deterministic replay, interposition hooks, path execution/history management, state caching/snapshotting, etc., as illustrated earlier in Figure 3). The four core algorithms described throughout Section 3 are however only 2420 LOC and the hooks to a target system are only 147 LOC on average. The static analyses support is written in 1799 LOC in Eclipse AST parser for Java programs [17], which covers many of our target systems including Cassandra, ZooKeeper, and Hadoop. For LogCabin Raft and Kudu Raft (in C++) and Spark (in Scala), we manually build the sets (§4.1). We leave porting to other language front-end parsers as future work.

FlyMC has been integrated with 8 widely-popular systems: Cassandra [56], Ethereum Blockchain [19], Hadoop [1], Kudu [24], Raft LogCabin [25, 67], Spark [83], ZooKeeper [51], and a 2-year old production system “X” of a large company (to the best of our knowledge, the largest checker integration compared to prior works). Within these systems, we model checked 10 unique protocol implementations: Cassandra Paxos, ZooKeeper leader election and atomic broadcast, Hadoop task management, Kudu Raft, LogCabin Raft leader election and snapshot, Spark core, Ethereum fast synchronization, and “X” leader election.

<table>
<thead>
<tr>
<th>BugName</th>
<th>Issue#</th>
<th>#Ev</th>
<th>#Cr</th>
<th>#Rb</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASS-1</td>
<td>6023</td>
<td>54</td>
<td>–</td>
<td>–</td>
<td>Paxos</td>
</tr>
<tr>
<td>CASS-2</td>
<td>6013</td>
<td>30</td>
<td>–</td>
<td>–</td>
<td>Paxos</td>
</tr>
<tr>
<td>CASS-3</td>
<td>5925</td>
<td>15</td>
<td>–</td>
<td>–</td>
<td>Paxos</td>
</tr>
<tr>
<td>ZOOK-1</td>
<td>335</td>
<td>46</td>
<td>3</td>
<td>3</td>
<td>LE, AB</td>
</tr>
<tr>
<td>ZOOK-2</td>
<td>790</td>
<td>39</td>
<td>1</td>
<td>1</td>
<td>LE</td>
</tr>
<tr>
<td>ZOOK-3</td>
<td>1419</td>
<td>41</td>
<td>3</td>
<td>3</td>
<td>LE</td>
</tr>
<tr>
<td>ZOOK-4</td>
<td>1492</td>
<td>24</td>
<td>1</td>
<td>–</td>
<td>LE</td>
</tr>
<tr>
<td>SPRK-1</td>
<td>19623</td>
<td>42</td>
<td>–</td>
<td>–</td>
<td>Spark Core</td>
</tr>
<tr>
<td>SPRK-2</td>
<td>15262</td>
<td>23</td>
<td>–</td>
<td>–</td>
<td>Spark Core</td>
</tr>
<tr>
<td>MAPR-1</td>
<td>5505</td>
<td>36</td>
<td>1</td>
<td>1</td>
<td>TA</td>
</tr>
<tr>
<td>RAFT-1</td>
<td>174</td>
<td>21</td>
<td>2</td>
<td>2</td>
<td>LE, Snapshot</td>
</tr>
<tr>
<td>ETHM-1</td>
<td>15138</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>Fast Sync</td>
</tr>
</tbody>
</table>

Table 1: Bug benchmarks (complex DC bugs). The table lists DC bugs used to benchmark checkers scalability. In the first column: “CASS” represents Cassandra, “ZOOK” ZooKeeper, “SPRK” Spark, “HADO” Hadoop MapReduce, “RAFT” Raft LogCabin, and “ETHM” Ethereum Blockchain. For the Protocols column: “LE” stands for leader election, “AB” atomic broadcast, and “TA” Task Assignment. “#Ev”, “#Cr” and “#Rb” stands for #Events, #Crashes and #Reboots that interleave to reach the bugs.

For reproducibility, FlyMC and some integrated systems are open-sourced [20].

### 7 Evaluation

We now evaluate FlyMC in terms of speed in reproducing DC bugs (§7.1), scalability (§7.2), coverage completeness (§7.3), and effectiveness in finding new bugs (§7.4).

**Bug benchmarks:** A popular method to evaluate a checker is how fast it can reproduce (reach) a DC bug given the corresponding workload. Table 1 shows the bug benchmarks that we use, including the number of events needed to hit the bugs (i.e., the bug “depth”). Most papers did not report bug depths [55, 73, 81], but it is important to pick deep real-world DC bugs for scalability evaluation. Interested readers can find the detailed bug descriptions in our technical report. [21, §7.1].

**Techniques compared:** We exhaustively compare FlyMC against six existing solutions as listed in Table 2: a purely random technique (Rand), two systematic techniques (m-DP and SAMC), and three hybrid systematic+random+bounded techniques (b-DP, r-DP, br-DP). The last category highlights how current approaches incorporate random and bounded flips to reach bugs faster.

We do not compare with DEMETER [49] and LMC [46] as they mainly reduce thread (local) interleavings in the context of reducing global interleavings (§8). We also do not show the results of depth-first-search (as used in MACEMC [81]) as DFS is extremely unscaleable [81, Fig. 9].
Figure 8: **FLYMC speed.** The top and bottom figures show the number of paths to explore (in log scale) and the wall-clock time, respectively, to find the buggy paths that make the bugs surface, as explained in Section 7.1. For the legend labels, please see Table 2. “↑” implies that the bug is not reached after 10,000 paths. Rand numbers are averaged from five tries.

<table>
<thead>
<tr>
<th>Label</th>
<th>Technique Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-DP</td>
<td>MoDiSt’ systematic DPOR reduction rule [81, §3.6] as discussed in §1 and §3 (comparison segments). Note that this reduction is also used in other checkers such as dBug [73, §2] and CrystalBall [80, §2.2].</td>
</tr>
<tr>
<td>SAMC</td>
<td>SAMC reduction algorithms [58, §3.3] as discussed in the comparison segments of §3.</td>
</tr>
<tr>
<td>b-DP</td>
<td>MoDiSt’ bounded+DPOR rule [81, §3.6] – run DPOR evaluation up until certain depth (i.e., #events).</td>
</tr>
<tr>
<td>r-DP</td>
<td>MoDiSt’s random+DPOR rule [81, §4.5] – execute random path on every 50 paths and then use DPOR to evaluate the path.</td>
</tr>
<tr>
<td>br-DP</td>
<td>Combination of the last two approaches above (bounded+random+DPOR).</td>
</tr>
<tr>
<td>Rand</td>
<td>A purely random exploration.</td>
</tr>
</tbody>
</table>

Table 2: **Techniques comparison.** The table lists all the techniques compared against FLYMC.

7.1 Speed

Figure 8a (in log scale) shows the number of paths explored to hit each of the bugs in Table 1 across different methods listed in Table 2. Figure 8b shows the wall-clock time. For readability, in each bar group, we put FLYMC bar in the middle (striped blue), systematic approaches on the left (m-DP and SAMC in patterned bars), hybrid and random on the right (br-DP, r-DP, b-DP, and Rand in solid colors). The horizontal blue markers are the height of FLYMC bars.

This evaluation method reflects a checker’s speed in helping developers reproduce hidden DC bugs. That is, imagine the users supply a workload that occasionally (non-deterministically) fails, the checker then should find the buggy interleaving(s) such that the developers can easily (and deterministically) replay them. Note that some methods fail to find the buggy paths after exploring 10,000 paths (marked with ↑ in Figure 8a). We stop at 10,000 paths to prioritize other evaluations. From the figure, we make the following observations:

(a) Within the systematic group, MoDiSt’ DPOR (m-DP) is not effective for 5 of the bugs (CASS-1, ZOOK-1, SPRK-1, MAPR-1, and RAFT-1), which is due to the limitations of black-box methods in pruning redundant paths.

(b) SAMC is faster than m-DP up to 25×. However, for two of the bugs (CASS-1 and ZOOK-1) SAMC cannot reach them within 10,000 paths and for the other two cases (MAPR-1 and RAFT-1) SAMC is relatively slow. Again, this is because SAMC does not have any static analysis support. Instead, developers need to manually implement their own reduction algorithms by following the SAMC principles. Therefore, in practice, SAMC might miss some potential reductions. Furthermore, it mainly focuses on reducing unnecessary crash

**Performance metrics:** The primary metrics of our evaluation are the numbers of (1) explored paths to hit a bug, (2) total wall-clock time to hit a bug, (3) explored paths to exhaust the entire state space of a given workload, and lastly, (4) unique global states covered over time.

**Evaluation scale:** Our extensive evaluation exercised over 200,000 paths (across all compared techniques) and used more than 130 machine days. We use Emulab “d430” machines [18] and Chameleon “compute_haswell” machines [16].
timings, hence does not scale for workloads with many concurrent messages.

(c) Random is random. Rand is the slowest method for 3 of the bugs, but it is faster than m-DP and SAMC in 4 and 1 other cases, respectively. In the latter cases, the degree of concurrency is low (e.g., to enable an event, random only needs to pick 1 out of 3 outstanding events), hence the probability that the “interesting” event is picked is high. However, in the former cases (more complex concurrency), random is not effective as there are too many choices and it blindly reorders non-interesting interleavings. For example, for CASS-1, Figure 9 shows how many inflight messages (y-axis) that can be chosen for every to-enable event (x-axis) within a path execution. For example, for pick #10 (x=10), there are 9 events to choose from (y=9). The figure shows that there are up to 10 choices when making a pick, hence random techniques are not effective for finding bugs in “deep” complex protocols and workloads.

(d) Bounded DPOR (b-DP) approximately has the same speed as random-hybrid ones, but interestingly for bug MAPR-1, the exploration completes, but the bug was not found (“X” in the figure). This shows a weakness of bounding the number of events to flip. Note that with bounded+random (br-DP) the randomness might shuffle the critical events first.

(e) Finally, FLYMC is the fastest among all methods. In our bug benchmark, we have not found any other checker that wins over FLYMC. For the most complex bug, CASS-1, FLYMC can find the buggy path in less than 2500 paths. In overall, FLYMC is faster at least by 16× on average and up to 78× (“at least” because of the non-finished explorations, labeled with “!” in Figure 8a). Sometimes “significant state-space reduction does not automatically translate to proportional increases in bug-finding effectiveness” [49, §5.3], however, we believe our results show that it is possible to stay systematic and increase bug-finding effectiveness with more advanced reduction and prioritization strategies.

Now we discuss the wall-clock speed in Figure 8b. As discussed in Section 5, for example in CASS-1, the per-path execution time is around 40 seconds in SAMC, 6 seconds in total in FLYMC, and 2 seconds (plus initialization time) in MODIST. Overall, per our design optimizations (§5), FLYMC is now 28× faster on average (up to 158×) compared to all methods. We do not show Rand in Figure 8b because we are comparing specific design implementations.

7.2 Scalability

To analyze why non-FLYMC algorithms cannot or are slow to hit some of the deep bugs above, we plot a different type of graph in Figure 10. Here, the x-axis represents the number of remaining events to hit the bug. For this, we control the “path prefix,” i.e., an initial subset of the buggy path. The maximum value in the x-axis represents the total number of events to hit the bug without any prefix (as in the “#E” column in Table 1). For example, for reproducing CASS-1 (Figure 10a), the workload generates a total of 54 events. Controlling path prefix means that the checker executes in deterministic order some of the earlier events (the prefix) and let the rest be reordered. For example, in Figure 10a, with x=30, we first enable the first 24 initial events and then let the checker reorder the remaining 30 events.

The y-axis shows the number of paths explored until the bug is reached given the remaining events. For instance, in Figure 10a, at x=26, MODIST’s m-DP must explore 163 paths to hit the bug, but SAMC and FLYMC are able to hit the bug in 55 and 27 paths respectively. With more remaining events to reorder (higher x), then more paths need to be explored (higher y), i.e., the larger the path explosion problem will be.

Essentially, the graphs in the figure show how FLYMC is more scalable than other approaches. For example, in CASS-1 (Figure 10a), at x=32, SAMC already explodes to more than 10,000 paths. On the other hand, FLYMC can find the buggy path in 2318 paths without any prefix (at x=54).
For ZOOK-1, our SAMC exploration results are different compared to the one reported in the SAMC’s paper [58]. Upon our conversation with the SAMC developers, there are two reasons for this difference. First, SAMC includes all the initialization events to reach the initial state \( S_0 \) (e.g., initial leader election to reach stable cluster), which FlyMC ignored. Second, in our checker, we implemented a proper vector clock, while SAMC did not, which causes our checker to detect more concurrent chain of events. Therefore, in our experiment, SAMC is no longer able to hit the bug under 10,000 paths. However, it does not take away SAMC’s conclusion that SAMC is still an order magnitude faster than MoDIST’s DPOR algorithm.

7.3 Coverage

For this evaluation, parallel flips algorithm is disabled because it is a prioritization algorithm. Hence, it does not affect all of the coverage evaluation.

(a) State coverage: The prior sections mainly measured FlyMC’s speed in finding buggy paths. Another form of evaluation is the speed to cover unique protocol states over the explored paths. Figure 11 (similar to the format of Figure 10 in [81]) shows the Cassandra Paxos protocol states covered (in y-axis) under a 3-update Paxos workload in CASS-1 within the first 10,000 explored paths (in x-axis). We make the following observations.

First, DFS is the worst among all (flat line). SAMC and mDPOR are faster but the growth rate is small. Random is the fastest, and for this reason, checkers sometimes mix their algorithms with random walks (see [81, Figure 10]), but unfortunately reduce their systematicity. So, random is fast in state coverage, but its non-systematic nature does not guarantee a buggy path to be found (e.g., random fails to reach three bugs in Figure 8a).

Second, on the other hand, FlyMC does not sacrifice systematicity and is only \( 3 \times \) slower than random. Being both fast and systematic is feasible. The figure also shows that coverage growth rate reduces over time (i.e., more paths to explore but they do not always lead to new unseen states).

(b) Complete coverage: Another question is whether the entire state space in a given workload can be covered, i.e., there are no more new unique states to explore. We performed this experiment for Cassandra Paxos and Kudu Raft workloads as shown earlier in Figure 1 on page, which we now elaborate.

FlyMC successfully exhausts the state space for all the workloads, with one to three concurrent key-value updates in Kudu Raft and Cassandra Paxos (Raft-1 to -3 and Paxos-1 to -3) within a reasonable time budget, as shown in Figure 1. The most complex one, Paxos-3, requires FlyMC to exhaust around 50,000 paths (1 machine week). Raft-3 is a much simpler case than Paxos as Raft only allows one leader node (per table/partition) to coordinate concurrent updates; for example, three coordinators A, B, C in Figure 2 (on page) updating the same key/partition is not allowed in Kudu Raft. Given this simplicity, Raft only needs two rounds, unlike Paxos’ three rounds. For this reason, Raft search space is much smaller than Paxos.

MoDIST’s DPOR and SAMC cannot finish the exploration under 10,000 paths. They do not scale well under these workloads for the following reasons.

MoDIST’s inter-node-independence DPOR algorithm focuses on taming the \( N \)-induced explosion (e.g., checking 3 nodes will not explode much compared to 2 nodes). However, under a more complex workload where the number of concurrent messages to each node increases, this inter-node-independence DPOR algorithm does not scale (e.g., Raft/Paxos-2/3 in Figure 1). What is needed is the intra-node message independence.

SAMC implements such an intra-node message independence. For example, in a single Paxos update (Paxos-1), the \( \text{ack++} \) received by the coordinator in each round of the Paxos three stages are considered commutative/independent. Hence, SAMC is more scalable than MoDIST. However, SAMC’s other algorithms such as crash independence and symmetry do not work in no-crash workloads.

Under two concurrent updates (“Paxos-2” in Figure 1), the path exploration explodes significantly in all the checkers. This is because in a single update, the three Paxos rounds (prepare, propose, commit) are serialized, but under two updates, each round of the first update can interleave with any round of the second update.

(c) Systematic coverage: We use the same sense of “systematic” that is used with concurrent programs [42, 77], where it refers to exploring the state spaces of concurrent processes. Our FlyMC reduction algorithms are systematic in that they cover all states relevant to observable events, i.e., the intercepted messages in distributed systems. These algorithms do not skip any interleavings that would lead to new unique states (more correctness sketches in [21, §4.1]). We would like to emphasize that this systematic property follows in principle from correct identification of communication and state symmetry and event independence, which is supported.
Next, the middle region shows that SAMC slightly reduces the explosion (SAMC is not highly effective for this bug, as explained in §7.1-7.2).

Figures 12b-d depict the individual reductions by state symmetry, event independence, and parallel flips, each reduces the explosion by almost an order of magnitude. Ultimately, Figure 12e shows that all FLYMC algorithms collectively provide two orders of magnitude of reduction in the CASS-1 Paxos workload.

(b) Ratio of reduced paths per algorithm: We plot Figure 13 to show the effectiveness of the individual FLYMC algorithms. Here, the x-axis represents the ratio of paths removed (by symmetry and event independence) and deprioritized (by parallel flips) from all the paths. This figure focuses on displaying two bugs from our benchmark with the most complex workloads. The graph essentially shows how all FLYMC algorithms successfully complement each other. We can also see that for different workloads, certain algorithms are more effective than the others.

For example, parallel flips are effective for CASS-1 (45%) because this workload (three Paxos updates) generates a high degree of concurrency (e.g., up to 9 outstanding events at a given time) and the important flips are far from the end of the queue, which parallel flips address (as illustrated earlier in Figure 7). Symmetry also works best in CASS-1 (33%) as the workload exercises replication-based protocols involving multiple worker/follower nodes, which are automatically considered symmetrical in FLYMC. Our event independence algorithm is effective in ZOOK-1 (43%) as the messages in this workload update different sets of variables (e.g., leader election messages and snapshot messages that touch different sets of variables) and, as this bug requires three crashes to surface, reducing unnecessary crashes is effective.

7.5 New Bugs

Finally, our last evaluation tests whether FLYMC can find new bugs. For this, we integrated FLYMC with (1) a recent stable version of Cassandra and (2) ZooKeeper. (3) a 2-year old proprietary system; the proprietary system is a production system that supports five other cloud services within the company (akin to how ZooKeeper supports HBase, Yarn, and other cloud systems). We found 10 new bugs in total, all con-
A client submits Paxos Write-1 (W1) to node A with a column in key K’s row. (2) Node A sends W1’s prepare messages and propose messages, accepted by all nodes A, B, and C. (3) Node A sends W1’s commit messages. (4) Node C crashes before accepting the commit message. (5) Nodes A and B accept the W1’s commit messages. At this point, A and B have stored W1 locally. (6) Node C reboots. (7) Another client submits Paxos Write-2 (W2) to A, updating another column in key K’s row. (8) Node A sends W2’s prepare messages (then propose and commit messages), accepted by all the nodes. At this point, Paxos nodes incorrectly have inconsistent data; A and B store W1-2, but C only stores W2 locally. The read repair did not happen during W2’s preparation. Thus, if a client reads K from C, she would get an inconsistent data (missing W1’s update).

8 Related Work

We now discuss the many efforts by the systems community in making distributed systems more reliable. **Model checking:** The main issue that these checkers try to overcome is the state-space explosion challenge. Table 3 summarizes many state-of-the-art checkers. First, MacEmC [55] combines DFS and random walk biased with weighted (prioritized) events manually labeled by testers (“Ran” ✓ in Table 3).

<table>
<thead>
<tr>
<th></th>
<th>Ran</th>
<th>Ind</th>
<th>Sym</th>
<th>Prio</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaceMC [55]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CrystalBall [80]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dBug [73]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MoDIST [81]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SAMC [58]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FLYMC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Subsequent works, CrystalBall [80], MoDIST [81], and dBug [73], began to adopt DPOR independence [38] in a black-box manner without domain-specific knowledge (“✓” ✓ and “Ind” ✓), hence do not scale well for complex workloads. The ✓ column in Table 3 shows that prior checkers did not interleave crash timings.

SAMC [58] enhances DPOR’s independence by exploiting white-box information and employs symmetry-based reduction (“✓” ✓ and “Sym” ✓). However, SAMC has three weaknesses. First, the domain-specific algorithms are written manually without any static analysis support, hence SAMC only introduces cautious reduction so that it does not accidentally skip interleavings that would lead to unexplored states. Second, SAMC primarily reduces crash interleavings, hence do not scale for complex messages (e.g., 3 concurrent Paxos updates). Third, SAMC does not consider any prioritization strategy.

FLYMC employs more advanced and powerful (“✓” ✓) independence- and symmetry-based reductions and a well-grounded prioritization strategy (“✓” ✓), backed by static analyses support (“✓” ✓). Hence, FLYMC scales for complex workloads including multiple crashes.

There are other checkers such as DIR [49] and LMC [46] but they mainly address the decoupling of local and global explorations (orthogonal to FLYMC). Hence, they are not shown in Table 3 because none of the bugs in our benchmarks require local thread interleavings. Other practical checkers, such as Jepsen [23] and Namazu [26], do not introduce a new reduction algorithm. Instead, they mainly depend on random-walk and randomly injecting network partition or failures to detect bugs. Finally, others suggested parallelizing DPOR by distributing the path executions across many worker nodes [74, 82]. This can be engineered into FLYMC.

Techniques in distributed checkers above are similar to those in “local” concurrency checking, e.g., symmetry [32, 75], disjoint-update independence [34, 42], property-driven pruning [76], and multiple branch flips [44], but the details are vastly different.

Lastly, one limitation that stateless distributed model checking has is that this technique only provides correctness
Acknowledgments

We thank Peter Druschel, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. This material was supported by funding from NSF (grant Nos. CNS-1350499, CNS-1526304, CNS-1405959, and CNS-1563956) as well as generous donations from Huawei, Dell EMC, Google Faculty Research Award, NetApp Faculty Fellowship, and CERES Center for Unstoppable Computing. The experiments in this paper were performed in the Utah Emulab¹ [78], the University of Chicago River [28] and Chameleon [15] testbeds.

Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References


¹http://www.emulab.net


[27] Personal Communication with ZooKeeper Developers (Michael Han, Patrick Hunt, and Alex Shraer).


