CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems

Maysam Yabandeh, Nikola Knežević, Dejan Kostić and Viktor Kuncak
School of Computer and Communication Sciences, EPFL, Switzerland
email: firstname.lastname@epfl.ch

Abstract

We propose a new approach for developing and deploying distributed systems, in which nodes predict distributed consequences of their actions, and use this information to detect and avoid errors. Each node continuously runs a state exploration algorithm on a recent consistent snapshot of its neighborhood and predicts possible future violations of specified safety properties. We describe a new state exploration algorithm, consequence prediction, which explores causally related chains of events that lead to property violation.

This paper describes the design and implementation of this approach, termed CrystalBall. We evaluate CrystalBall on RandTree, BulletPrime, Paxos, and Chord distributed system implementations. We identified new bugs in mature Mace implementations of three systems. Furthermore, we show that if the bug is not corrected during system development, CrystalBall is effective in steering the execution away from inconsistent states at runtime.

1 Introduction

Complex distributed protocols and algorithms are used in enterprise storage systems, distributed databases, large-scale planetary systems, and sensor networks. Errors in these protocols translate to denial of service to some clients, potential loss of data, and monetary losses. The Internet itself is a large-scale distributed system, and there are recent proposals [19] to improve its routing reliability by further treating routing as a distributed consensus problem [26]. Design and implementation problems in these protocols have the potential to deny vital network connectivity to a large fraction of users.

Unfortunately, it is notoriously difficult to develop reliable high-performance distributed systems that run over asynchronous networks. Even if a distributed system is based on a well-understood distributed algorithm, its implementation can contain errors arising from complexities of realistic distributed environments or simply coding errors [27]. Many of these errors can only manifest after the system has been running for a long time, has developed a complex topology, and has experienced a particular sequence of low-probability events such as node resets. Consequently, it is difficult to detect such errors using testing and model checking, and many of such errors remain unfixed after the system is deployed.

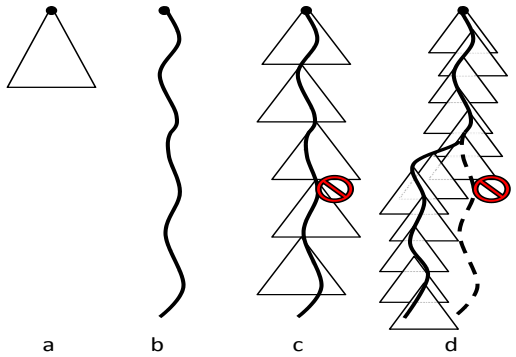


Figure 1: Execution path coverage by a) classic model checking, b) replay-based or live predicate checking, c) CrystalBall in deep online debugging mode, and d) CrystalBall in execution steering mode. A triangle represents the state space searched by the model checker; a full line denotes an execution path of the system; a dashed line denotes an avoided execution path that would lead to an inconsistency.

We propose to leverage increases in computing power and bandwidth to make it easier to find errors in distributed systems, and to increase the resilience of the deployed systems with respect to any remaining errors. In our approach, distributed system nodes predict consequences of their actions while the system is running. Each node runs a state exploration algorithm on a consistent snapshot of its neighborhood and predicts which actions can lead to violations of user-specified consistency properties. As Figure 1 illustrates, the ability to detect future inconsistencies allows us to address the problem

of reliability in distributed systems on two fronts: debugging and resilience.

- Our technique enables deep online debugging because it explores more states than live runs alone or model checking from the initial state. For each state that a running system experiences, our technique checks many additional states that the system did not go through, but that it could reach in similar executions. This approach combines benefits of distributed debugging and model checking.
- Our technique aids resilience because a node can modify its behavior to avoid a predicted inconsistency. We call this approach *execution steering*. Execution steering enables nodes to resolve nondeterminism in ways that aim to minimize future inconsistencies.

To make this approach feasible, we need a fast state exploration algorithm. We describe a new algorithm, termed *consequence prediction*, which is efficient enough to detect future violations of safety properties in a running system. Using this approach we identified bugs in Mace implementations of a random overlay tree, and the Chord distributed hash table. These implementations were previously tested as well as model-checked by exhaustive state exploration starting from the initial system state. Our approach therefore enables the developer to uncover and correct bugs that were not detected using previous techniques. Moreover, we show that, if a bug is not detected during system development, our approach is effective in steering the execution away from erroneous states, without significantly degrading the performance of the distributed service.

1.1 Contributions

We summarize the contributions of this paper as follows:

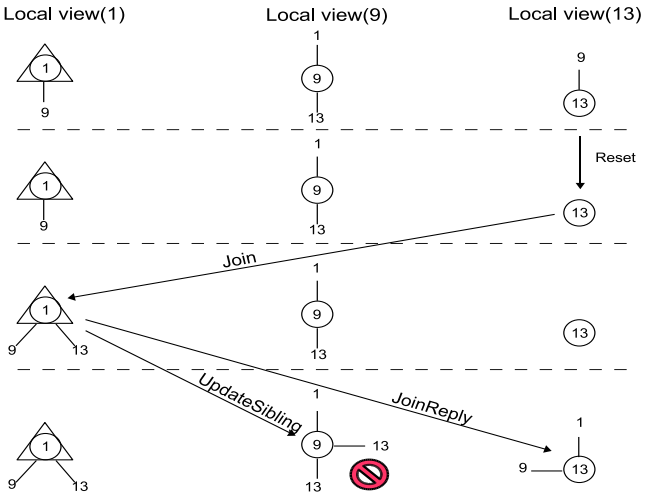
- We introduce the concept of continuously executing a state space exploration algorithm in parallel with a deployed distributed system, and introduce an algorithm that produces useful results even under tight time constraints arising from runtime deployment;
- We describe a mechanism for feeding a consistent snapshot of the neighborhood of a node in a large-scale distributed system into a running model checker; the mechanism enables reliable consequence prediction within limited time and bandwidth constraints;
- We present execution steering, a technique that enables the system to steer execution away from possible inconsistencies;
- We describe CrystalBall, the implementation of our approach on top of the Mace framework [21]. We evaluate CrystalBall on RandTree, Bullet′, Paxos, and Chord distributed system implementations. CrystalBall detected several previously unknown bugs that can cause system nodes to reach inconsistent states. Moreover, if the developer is not in a position to fix these bugs, CrystalBall's execution steering predicts them in a deployed system and steers execution away from them, all with an acceptable impact on the overall system performance.

1.2 Example

We next describe an example of an inconsistency exhibited by a distributed system, then show how CrystalBall predicts and avoids it. The inconsistency appears in the Mace [21] implementation of the RandTree overlay. RandTree implements a random, degree-constrained overlay tree designed to be resilient to node failures and network partitions. Trees built by an earlier version of this protocol serve as a control tree for a number of large-scale distributed services such as Bullet [23] and RanSub [24]. In general, trees are used in a variety of multicast scenarios [3, 7] and data collection/monitoring environments [17]. Inconsistencies in these environments translate to denial of service to users, data loss, inconsistent measurements, and suboptimal control decisions. The RandTree implementation was previously manually debugged both in local- and wide-area settings over a period of three years, as well as debugged using an existing model checking approach [22], but, to our knowledge, this inconsistency has not been discovered before (see Section 4 for some of the additional bugs that CrystalBall discovered).

RandTree Topology. Nodes in a RandTree overlay form a directed tree of bounded degree. Each node maintains a list of its children and the address of the root. The node with the numerically smallest IP address acts as the root of the tree. Each non-root node contains the address of its parent. Children of the root maintain a sibling list. Note that, for a given node, its parent, children, and siblings are all distinct nodes. The seemingly simple task of maintaining a consistent tree topology is complicated by the requirement for groups of nodes to agree on their roles (root, parent, child, sibling) across asynchronous networks, in the face of node failures, and machine slowdowns.

Joining the Overlay. A node n_j joins the overlay by issuing a Join request to one of the designated nodes. If the node receiving the join request is not the root, it forwards the request to the root. If the root already has the maximal number of children, it asks one of its children to incorporate the node into the overlay. Once the



Safety property: children and siblings are disjoint lists

Figure 2: An inconsistency in a run of RandTree

request reaches a node n_p whose number of children is less than maximum allowed, node n_p inserts n_j as one of its children, and notifies n_j about a successful join using a JoinReply message (if n_p is the root, it also notifies its other children about their new sibling n_j using an UpdateSibling message).

Example System State. The first row of Figure 2 shows a state of the system that we encountered by running RandTree in the ModelNet cluster [43] starting from the initial state. We examine the local states of nodes n_1, n_9, and n_{13}. For each node n we display its neighborhood view as a small graph whose central node is n itself, marked with a circle. If a node is root and in a "joined" state, we mark it with a triangle in its own view.

The state in the first row of Figure 2 is formed by n_{13} joining as the only child of n_9 and then n_1 joining and assuming the role of the new root with n_9 as its only child (n_{13} remains as the only child of n_9). Although the final state shown in first row of Figure 2 is simple, it takes 13 steps of the distributed system (such as atomic handler executions, including application events) to reach this state from the initial state.

Scenario Exhibiting Inconsistency. Figure 2 describes a sequence of actions that leads to a state that violates the consistency of the tree. We use arrows to represent the sending and the receiving of some of the relevant messages. A dashed line separates distinct distributed system states (for simplicity we skip certain intermediate states and omit some messages).

The sequence begins by a silent reset of node n_{13} (such reset can be caused by, for example, a power failure). After the reset, n_{13} attempts to join the overlay again. The root n_1 accepts the join request and adds n_{13} as its child. Up to this point node n_9 received no information on actions that followed the reset of n_{13}, so n_9 maintains n_{13} as its own child. When n_1 accepts n_{13} as a child, it sends an UpdateSibling message to n_9. At this point, n_9 simply inserts n_{13} into the set of its sibling. As a result, n_{13} *appears both in the list of children and in the list of siblings of* n_9, which is inconsistent with the notion of a tree.

Challenges in Finding Inconsistencies. We would clearly like to avoid inconsistencies such as the one appearing in Figure 2. Once we have realized the presence of such inconsistency, we can, for example, modify the handler for the UpdateSibling message to remove the new sibling from the children list. Previously, researchers had successfully used explicit-state model checking to identify inconsistencies in distributed systems [22] and reported a number of safety and liveness bugs in Mace implementations. However, due to an exponential explosion of possible states, current techniques capable of model checking distributed system implementations take a prohibitively long time to identify inconsistencies, even for seemingly short sequences such as the ones needed to generate states in Figure 2. For example, when we applied the Mace Model Checker's [22] exhaustive search to the safety properties of RandTree starting from the initial state, it failed to identify the inconsistency in Figure 2 even after running for 17 hours (on a 3.4-GHz Pentium-4 Xeon that we used for all our experiments in Section 4). The reason for this long running time is the large number of states reachable from the initial state up to the depth at which the bug occurs, all of which are examined by an exhaustive search.

1.3 CrystalBall Overview

Instead of running the model checker from the initial state, we propose to execute a model checker concurrently with the running distributed system, and continuously feed current system states into the model checker. When, in our example, the system reaches the state at the beginning of Figure 2, the model checker will predict the state at the end of Figure 2 as a possible future inconsistency. In summary, instead of trying to predict all possible inconsistencies starting from the initial state (which for complex protocols means never exploring states beyond the initialization phase), our model checker predicts inconsistencies that can occur in a system that has been running for a significant amount of time in a realistic environment.

As Figure 1 suggests, compared to the standard model checking approach, this approach identifies inconsistencies that can occur within much longer system executions. Compared to simply running the system for a long time, our approach has two advantages.

1. Our approach systematically covers a large number

of executions that contain low-probability events, such as node resets that ultimately triggered the inconsistency in Figure 2. It can take a very long time for a running system to encounter such a scenario, which makes testing for possible bugs difficult. Our technique therefore improves system debugging by providing a new technique that combines some of the advantages of testing and static analysis.

2. Our approach identifies inconsistencies before they actually occur. This is possible because the model checker can simulate packet transmission in time shorter than propagation latency, and because it can simulate timer events in time shorter than than the actual time delays. This aspect of our approach opens an entirely new possibility: adapt the behavior of the running system on the fly and avoid an inconsistency. We call this technique *execution steering*. Because it does not rely on a history of past inconsistencies, execution steering is applicable even to inconsistencies that were previously never observed in past executions.

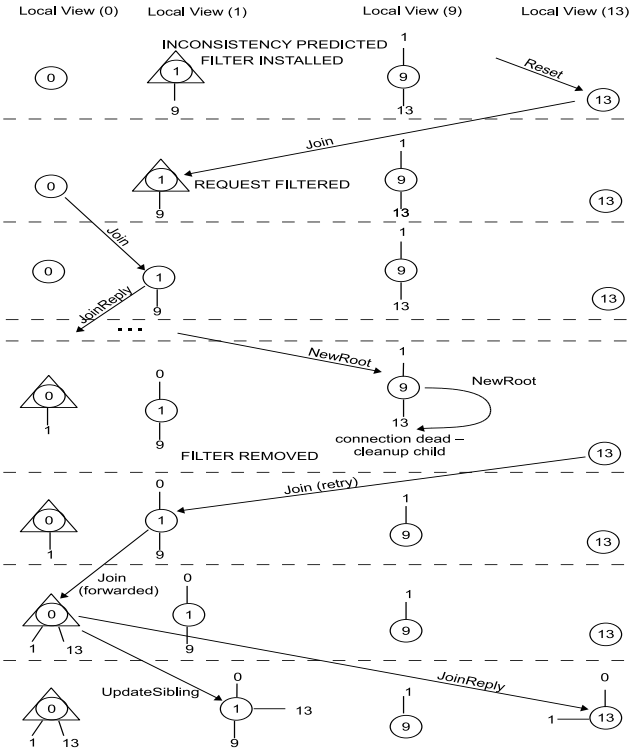


Figure 3: An Example execution sequence that avoids the inconsistency from Figure 2 thanks to execution steering.

Example of Execution Steering. In our example, a model checking algorithm running in n_1 detects the violation at the end of Figure 2. Given this knowledge, execution steering causes node n_1 not to respond to the join request of n_{13} and to break the TCP connection with it. Node n_{13} eventually succeeds joining the random tree (perhaps after some other nodes have joined first). The stale information about n_{13} in n_9 is removed once n_9 discovers that the stale communication channel with n_{13} is closed, which occurs the first time when n_9 attempts to communicate with n_{13}. Figure 3 presents one scenario illustrating this alternate execution sequence. Effectively, execution steering has exploited the non-determinism and robustness of the system to choose an alternative execution path that does not contain the inconsistency.

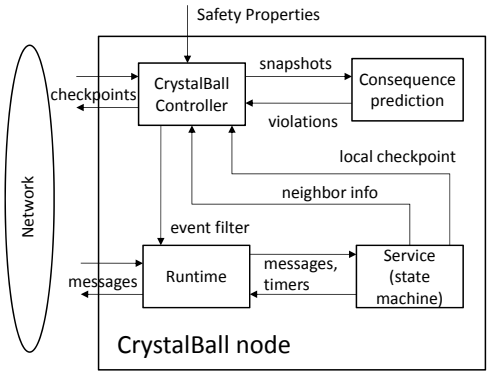


Figure 4: High-level overview of CrystalBall

2 CrystalBall Design

We next sketch the design of CrystalBall (see [44] for details). Figure 4 shows the high-level overview of a CrystalBall-enabled node. We concentrate on distributed systems implemented as state machines, as this is a widely-used approach [21, 25, 26, 37, 39].

The state machine interfaces with the outside world via the runtime module. The runtime receives the messages coming from the network, demultiplexes them, and invokes the appropriate state machine handlers. The runtime also accepts application level messages from the state machines and manages the appropriate network connections to deliver them to the target machines. This module also maintains the timers on behalf of all services that are running.

The CrystalBall controller contains a checkpoint manager that periodically collects consistent snapshots of a node's neighborhood. The controller feeds them to the model checker, along with a checkpoint of the local state. The model checker runs the consequence prediction algorithm which checks user- or developer-defined properties and reports any violation in the form of a sequence of events that leads to an erroneous state.

CrystalBall can operate in two modes. In the *deep online debugging mode* the controller only outputs the information about the property violation. In the *execution*

steering mode the controller examines the report from the model checker, prepares an *event filter* that can avoid the erroneous condition, checks the filter's impact, and installs it into the runtime if it is deemed to be safe.

2.1 Consistent Neighborhood Snapshots

To check system properties, the model checker requires a snapshot of the system-wide state. Ideally, every node would have a consistent, up-to-date checkpoint of every other participant's state. Doing so would give every node high confidence in the reports produced by the model checker. However, given that the nodes could be spread over a high-latency wide-area network, this goal is unattainable. In addition, the sheer amount of bandwidth required to disseminate checkpoints might be excessive.

Given these fundamental limitations, we use a solution that aims for scalability: we apply model checking to a *subset* of all states in a distributed system. We leverage the fact that in scalable systems a node typically communicates with a small subset of other participants ("neighbors") and perform model checking only on this neighborhood. In some distributed hash table implementations, a node keeps track of $O(\log n)$ other nodes; in mesh-based content distribution systems nodes communicate with a constant number of peers; or this number does not explicitly grow with the size of the system. In a random overlay tree, a node is typically aware of the root, its parent, its children, and its siblings. We therefore arrange for a node to distribute its state checkpoints to its neighbors, and we refer to them as *snapshot neighborhood*. The *checkpoint manager* maintains checkpoints and snapshots. Other CrystalBall components can request an on-demand snapshot to be gathered by invoking an appropriate call on the checkpoint manager.

Discovering and Managing Snapshot Neighborhoods. To propagate checkpoints, the checkpoint manager needs to know the set of a node's neighbors. This set is dependent upon a particular distributed service. We use two techniques to provide this list. In the first scheme, we ask the developer to implement a method that will return the list of neighbors. The checkpoint manager then periodically queries the service and updates its snapshot neighborhood.

Since changing the service code might not always be possible, our second technique uses a heuristic to determine the snapshot neighborhood. Specifically, we periodically query the runtime to obtain the list of open connections (for TCP), and recent message recipients (for UDP). We then cluster connection endpoints according to the communication times, and selects a sufficiently large cluster of recent connections.

Enforcing Snapshot Consistency. To avoid false positives, we ensure that the neighborhood snapshot corresponds to a consistent view of a distributed system at some point of logical time. There has been a large body of work in this area, starting with the seminal paper by Chandy and Lamport [5]. We use one of the recent algorithms for obtaining consistent snapshots [29], in which the general idea is to collect a set of checkpoints that do not violate the happens-before relationship [25] established by messages sent by the distributed service.

Instead of gathering a global snapshot, a node periodically sends a checkpoint request to the members of its snapshot neighborhood. Even though nodes receive checkpoints only from a subset of nodes, all distributed service and checkpointing messages are instrumented to carry the checkpoint number (logical clock) and each neighborhood snapshot is a fragment of a globally consistent snapshot. In particular, a node that receives a message with a logical timestamp greater than its own logical clock takes a forced checkpoint. The node then uses the forced checkpoint to contribute to the consistent snapshot when asked for it.

Node failures are commonplace in distributed systems, and our algorithm has to deal with them. The checkpoint manager proclaims a node to be dead if it experiences a communication error (e.g., a broken TCP connection) with it while collecting a snapshot. An additional cause for an apparent node failure is a change of a node's snapshot neighborhood in the normal course of operation (e.g., when a node changes parents in the random tree). In this case, the node triggers a new snapshot gather operation.

Checkpoint Content. Although the total footprint of some services might be very large, this might not necessarily be reflected in checkpoint size. For example, the Bullet′ [23] file distribution application has non-negligible total footprint, but the actual file content transferred in Bullet′ does not play any role in consistency detection. In general, the checkpoint content is given by a serialization routine. The developer can choose to omit certain parts of the state from serialized content and reconstruct them if needed at de-serialization time. As a result, checkpoints are smaller, and the code compensates the lack of serialized state when a local state machine is being created from a remote node's checkpoint in the model checker. We use a set of well-known techniques for managing checkpoint storage (quotas) and controlling the bandwidth used by checkpoints (bandwidth limits, compression).

2.2 Consequence Prediction Algorithm

The key to enabling fast prediction of future inconsistencies in CrystalBall is our consequence prediction algorithm, presented in Figure 5. For readability, we present

```
1  proc findConseq(currentState : G, property : (G → boolean))
2    explored = emptySet(); errors = emptySet();
3    localExplored = emptySet();
4    frontier = emptyQueue();
5    frontier.addLast(currentState);
6    while (!STOP_CRITERION)
7      state = frontier.popFirst();
8      if (!property(state))
9        errors.add(state); // predicted inconsistency found
10     explored.add(hash(state));
11     foreach ((n,s) ∈ state.L) // node n in local state s
12       // process all network handlers
13       foreach (((s,m),(s',c)) ∈ H_M where (n,m) ∈ state.I)
14         // node n handles message m according to st. machine
15         addNextState(state,n,s,s',{m},c);
16       // process local actions only for fresh local states
17       if (!localExplored.contains(hash(n,s)))
18         foreach (((s,a),(s',c)) ∈ H_A)
19           addNextState(state,n,s,s',{},c);
20       localExplored.add(hash(n,s));
21
22 proc addNextState(state,n,s,s',c0,c)
23   nextState.L = (state.L \ {(n,s)}) ∪ {(n,s')};
24   nextState.I = (state.I \ c0) ∪ c;
25   if (!explored.contains(hash(nextState)))
26     frontier.addLast(nextState);
```

Figure 5: Consequence Prediction Algorithm

the algorithm as a refinement of a generic state-space search. The notation is based on a high-level semantics of a distributed system, shown in Figure 6. (Our concrete model checker implementation uses an iterative deepening algorithm which combines memory efficiency of depth-first search, while favoring the states in the near future, as in breadth-first search.) The STOP_CRITERION in Figure 5 in our case is given by time constraints and external commands to restart the model checker upon the arrival of a new snapshot.

In Line 8 of Figure 5 the algorithm checks whether the explored state satisfies the desired safety properties. The developer can use a simple language [22] that involves loops, existential and comparison operators, state variables, and function invocations to specify the properties.

Exploring Independent Chains. We can divide the actions in a distributed system into *event chains*, where each chain starts with an application or scheduler event and continues by triggering network events. We call two chains *independent* if no event of the first chain changes state of a node involved in the second chain. Consequence Prediction avoids exploring the interleavings of independent chains. Therefore, the test in Line 17 of Figure 5 makes the algorithm re-explore the scheduler and application events of a node if and only if the previous events changed the local state of the node. For dependent chains, if a chain event changes local state of a node, Consequence Prediction therefore explores all other active chains which have been initiated from this node.

N – node identifiers
S – node states
M – message contents
$N \times M$ – (destination process, message)-pair
$C = 2^{N \times M}$ – set of messages with destination
A – local node actions (timers, application calls)

system state : $(L, I) \in G, \;\; G = 2^{N \times S} \times 2^{N \times M}$
local node states : $L \subseteq N \times S$ (function from N to S)
in-flight messages (network) : $I \subseteq N \times M$

behavior functions for each node :
message handler : $H_M \subseteq (S \times M) \times (S \times C)$
internal action handler : $H_A \subseteq (S \times A) \times (S \times C)$

transition function for distributed system :

node message handler execution :

$$\frac{((s_1, m), (s_2, c)) \in H_M}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I_0 \uplus \{(n, m)\}) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I_0 \cup c)\end{array}}$$

internal node action (timer, application calls) :

$$\frac{((s_1, a), (s_2, c)) \in H_A}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I \cup c)\end{array}}$$

Figure 6: A Simple Model of a Distributed System

Note that $hash(n, s)$ in Figure 5 implies that we have separate tables corresponding to each node for keeping hashed local states. If a state variable is not necessary to distinguish two separate states, the user can annotate the state variable that he or she does not want include in the hash function, improving the performance of Consequence Prediction. Instead of holding all encountered hashes, the hash table could be designed as a bounded cache to fit into the L2 cache or main memory, favoring access speed while admitting the possibility of re-exploring previously seen states.

Although simple, the idea of removing from the search actions of nodes with previously seen states eliminates many (uninteresting) interleavings from search and has a profound impact on the search depth that the model checker can reach with a limited time budget. This change was therefore key to enabling the use of the model checker at runtime. Knowing that consequence prediction avoids considering certain states, the question remains whether the remaining states are sufficient to make the search useful. Ultimately, the answer to this question comes from our evaluation (Section 4).

2.3 Execution Steering

CrystalBall's execution steering mode enables the system to avoid entering an erroneous state by steering its

execution path away from predicted inconsistencies. If a protocol was designed with execution steering in mind, the runtime system could report a predicted inconsistency as a special programming language exception, and allow the service to react to the problem using a service-specific policy. However, to measure the impact on existing implementations, this paper focuses on generic runtime mechanisms that do not require the developer to insert exception-handling code.

Event Filters. Recall that a node in our framework operates as a state machine and processes messages, timer events, and application calls via handlers. Upon noticing that running a certain handler can lead to an erroneous state, CrystalBall installs an *event filter*, which temporarily blocks the invocation of the state machine handler for messages from the relevant sender.

The rationale is that a distributed system often contains a large amount of non-determinism that allows it to proceed even if certain transitions are disabled. For example, if the offending message is a Join request in a random tree, ignoring the message can prevent violating a local state property. The joining nodes can later retry the procedure with an alternative potential parent and successfully join the tree. Similarly, if handling a message causes an equivalent of a race condition manifested as an inconsistency, delaying message handling allows the system to proceed to the point where handling the message becomes safe again. Note that state machine handlers are atomic, so CrystalBall is unlikely to interfere with any existing recovery code.

Point of Intervention. In general, execution steering can intervene at several points in the execution path. Our current policy is to steer the execution as early as possible. For example, if the erroneous execution path involves a node issuing a Join request after resetting, the system's first interaction with that node occurs at the node which receives its join request. If this node discovers the erroneous path, it can install the event filter.

Non-Disruptiveness of Execution Steering. Ideally, execution steering would always prevent inconsistencies from occurring, without introducing new inconsistencies due to a change in behavior. In general, however, guaranteeing the absence of inconsistencies is as difficult as guaranteeing that the entire program is error-free. CrystalBall therefore makes execution steering safe in practice through two mechanisms:

1. **Sound Choice of Filters.** It is important that the chosen corrective action does not sacrifice the soundness of the state machine. A *sound filtering* is the one in which the observed sequence of events after filtering is a subset of possible sequence of events without filtering. The breaking of a TCP connection is common in a distributed system using TCP. Therefore, such distributed systems include failure-handling code that deals with broken TCP connections. This makes sending a TCP RST signal a good candidate for a sound event filter, and is the filter we choose to use in CrystalBall. In the case of communication over UDP, the filter simply drops the UDP packet, which could similarly happen in normal operation of the network.

2. **Exploration of Corrected Executions.** Before allowing the event filter to perform an execution steering action, CrystalBall runs the consequence prediction algorithm to check the effect of the event filter action on the system. If the consequence prediction algorithm does not suggest that the filter actions are safe, CrystalBall does not attempt execution steering and leaves the system to proceed as usual.

Rechecking Previously Discovered Violations. An event filter reflects possible future inconsistencies reachable from the current state, and leaving an event filter in place indefinitely could deny service to some distributed system participants. CrystalBall therefore removes the filters from the runtime after every model checking run. However, it is useful to quickly check whether the previously identified error path can still lead to an erroneous condition in a new model checking run. This is especially important given the asynchronous nature of the model checker relative to the system messages, which can prevent the model checker from running long enough to rediscover the problem. To prevent this from happening, the first step executed by the model checker is to replay the previously discovered error paths. If the problem reappears, CrystalBall immediately reinstalls the appropriate filter.

Immediate Safety Check. CrystalBall also supports *immediate safety check*, a mechanism that avoids inconsistencies that would be caused by executing the current handler. Such imminent inconsistencies can happen even in the presence of execution steering because 1) consequence prediction explores states given by only a subset of all distributed system nodes, and 2) the model checker runs asynchronously and may not always detect inconsistencies in time. The immediate safety check speculatively runs the handler, checks the consistency properties in the resulting state, and prevents actual handler execution if the resulting state is inconsistent.

We have found that exclusively using immediate safety check would not be sufficient for avoiding inconsistencies. The advantages of installing event filters are: i) performance benefits of avoiding the bug sooner, e.g., reducing unnecessary message transmission, ii) faster reaction to an error, which implies greater chance of avoiding a "point of no return" after which error avoidance is impossible, and iii) the node that is supposed to ultimately avoid the inconsistency by immediate safety

check might not have all the checkpoints needed to notice the violation; this can result in false negatives (as shown in Figure 9).

Liveness Issues. It is possible that by applying an event filter would affect liveness properties of a distributed system. In our experience, due to a large amount of nondeterminism (e.g., the node is bootstrapped with a list of multiple nodes it can join), the system usually finds a way to make progress. We focus on enforcing safety properties, and we believe that occasionally sacrificing liveness is a valid approach. According to a negative result by Fischer, Lynch, and Paterson [12], it is impossible to have both in an asynchronous system anyway. (For example, the Paxos [26] protocol guarantees safety but not liveness.)

2.4 Scope of Applicability

CrystalBall does not aim to find all errors; it is rather designed to find and avoid important errors that can manifest in real runs of the system. Results in Section 4 demonstrate that CrystalBall works well in practice. Nonetheless, we next discuss the limitations of our approach and characterize the scenarios in which we believe CrystalBall to be effective.

Up-to-Date Snapshots. For Consequence Prediction to produce results relevant for execution steering and immediate safety check, it needs to receive sufficiently many node checkpoints sufficiently often. (Thanks to snapshot consistency, this is not a problem for deep online debugging.) We expect the stale snapshots to be less of an issue with *stable properties*, e.g., those describing a deadlock condition [5]. Since the node's own checkpoint might be stale (because of enforcing consistent neighborhood snapshots for checking multi-node properties), immediate safety check is perhaps more applicable to node-local properties.

Higher frequency of changes in state variables requires higher frequency of snapshot exchanges. High-frequency snapshot exchanges in principle lead to: 1) more frequent model checker restarts (given the difficulty in building incremental model checking algorithms), and 2) high bandwidth consumption. Among the examples for which our techniques is appropriate are overlays in which state changes are infrequent.

Consequence Prediction as a Heuristic. Consequence Prediction is a heuristic that explores a subset of the search space. This is an expected limitation of explicit-state model checking approaches applied to concrete implementations of large software systems. The key question in these approaches is directing the search towards most interesting states. Consequence Prediction uses information about the nature of the distributed system to guide the search; the experimental results in Section 4 show that it works well in practice, but we expect that further enhancements are possible.

3 Implementation Highlights

We built CrystalBall on top of the Mace [21] framework. Mace allows distributed systems to be specified succinctly and outputs high-performance C++ code. We implemented our consequence prediction within the Mace model checker, and run the model checker as a separate thread that communicates future inconsistencies to the runtime. Our current implementation of the immediate safety check executes the handler in a copy of the state machine's virtual memory (using fork()), and holds the transmission of messages until the successful completion of the consistency check. Upon encountering an inconsistency in the copy, the runtime does not execute the handler in the primary state machine. In case of applications with high messaging/state change rates in which the performance of immediate safety check is critical, we could obtain a state checkpoint [41] before running the handler and rollback to it in case of an encountered inconsistency. Another option would be to employ operating system-level speculation [32].

4 Evaluation

Our experimental evaluation addresses the following questions: **1)** Is CrystalBall effective in finding bugs in live runs? **2)** Can any of the bugs found by CrystalBall also be identified by the MaceMC model checker alone? **3)** Is execution steering capable of avoiding inconsistencies in deployed distributed systems? **4)** Are the CrystalBall-induced overheads within acceptable levels?

4.1 Experimental Setup

We conducted our live experiments using ModelNet [43]. ModelNet allows us to run live code in a cluster of machines, while application packets are subjected to packet delay, loss, and congestion typical of the Internet. Our cluster consists of 17 older machines with dual 3.4 GHz Pentium-4 Xeons with hyper-threading, 8 machines with dual 2.33 GHz dual-core Xeon 5140s, and 3 machines with 2.83 GHz Xeon X3360s (for Paxos experiments). Older machines have 2 GB of RAM, while the newer ones have 4 GB and 8 GB. These machines run GNU/Linux 2.6.17. One 3.4 GHz Pentium-4 machine running FreeBSD 4.9 served as the ModelNet packet forwarder for these experiments. All machines are interconnected with a full-rate 1-Gbps Ethernet switch.

We consider two deployment scenarios. For our large-scale experiments with deep online debugging, we multiplex 100 logical end hosts running the distributed ser-

vice across the 20 Linux machines, with 2 participants running the model checker on 2 different machines. We run with 6 participants for small-scale debugging experiments, one per machine.

We use a 5,000-node INET [6] topology that we further annotate with bandwidth capacities for each link. The INET topology preserves the power law distribution of node degrees in the Internet. We keep the latencies generated by the topology generator; the average network RTT is 130ms. We randomly assign participants to act as clients connected to one-degree stub nodes in the topology. We set transit-transit links to be 100 Mbps, while we set access links to 5 Mbps/1 Mbps inbound-/outbound bandwidth. To emulate the effects of cross traffic, we instruct ModelNet to drop packets at random with a probability chosen uniformly at random between [0.001,0.005] separately for each link.

4.2 Deep Online Debugging Experience

We have used CrystalBall to find inconsistencies (violations of safety properties) in two mature implemented protocols in Mace, namely an overlay tree (RandTree) and a distributed hash table (Chord [42]). These implementation were not only manually debugged both in local- and wide-area settings, but were also model checked using MaceMC [22]. We have also used our tool to find inconsistencies in Bullet′, a file distribution system that was originally implemented in MACEDON [37], and then ported to Mace. We found 13 new subtle bugs in these three systems that caused violation of safety properties.

System	Bugs found	LOC Mace/C++
RandTree	7	309 / 2000
Chord	3	254 / 2200
Bullet′	3	2870 / 19628

Table 1: Summary of inconsistencies found for each system using CrystalBall. LOC stands for lines of code and reflects both the MACE code size and the generated C++ code size. The low LOC counts for Mace service implementations are a result of Mace's ability to express these services succinctly. This number does not include the line counts for libraries and low-level services that services use from the Mace framework.

Table 1 summarizes the inconsistencies that CrystalBall found in RandTree, Chord and Bullet′. Typical elapsed times (wall clock time) until finding an inconsistency in our runs have been from less than an hour up to a day. This time allowed the system being debugged to go through complex realistic scenarios.[1] CrystalBall identified inconsistencies by running consequence prediction from the current state of the system for up to several hundred seconds. To demonstrate their depth and complexity, we detail four out of 13 inconsistencies we found in the three services we examined.

[1]During this time, the model checker ran concurrently with a normally executing system. We therefore do not consider this time to be wasted by the model checker before deployment; rather, it is the time consumed by a running system.

4.2.1 Example RandTree Bugs Found

We next discuss bugs we identified in the RandTree overlay protocol presented in Section 1.2. We name bugs according to the consistency properties that they violate.

Children and Siblings Disjoint. The first safety property we considered is that the children and sibling lists should be disjoint. CrystalBall identified the scenario from Figure 2 in Section 1.2 that violates this property. The problem can be corrected by removing the stale information about children in the handler for the UpdateSibling message. CrystalBall also identified variations of this bug that requires changes in other handlers.

Recovery Timer Should Always Run. An important safety property for RandTree is that the recovery timer should always be scheduled. This timer periodically causes the nodes to send Probe messages to the peer list members with which it does not have direct connection. It is vital for the tree's consistency to keep nodes up-to-date about the global structure of the tree. The property was written by the authors of [22] but the authors did not report any violations of it. We believe that our approach discovered it in part because our experiments considered more complex join scenarios.

Scenario exhibiting inconsistency. CrystalBall found a violation of the property in a state where node A joins itself, and changes its state to "joined" but does not schedule any timers. Although this does not cause problems immediately, the inconsistency happens when another node B with smaller identifier tries to join, at which point A gives up the root position, selects B as the root, and adds B it to its peer list. At this point A has a non-empty peer list but no running timer.

Possible correction. Keep the timer scheduled even when a node has an empty peer list.

4.2.2 Example Chord Bug Found

We next describe a violation of a consistency property in Chord [42], a distributed hash table that provides key-based routing functionality. Chord and other related distributed hash tables form a backbone of a large number of proposed and deployed distributed systems [17, 35, 38].

Chord Topology. Each Chord node is assigned a Chord id (effectively, a key). Nodes arrange themselves in an overlay ring where each node keeps pointers to its predecessor and successor. Even in the face of asynchronous message delivery and node failures, Chord has to maintain a ring in which the nodes are ordered according to

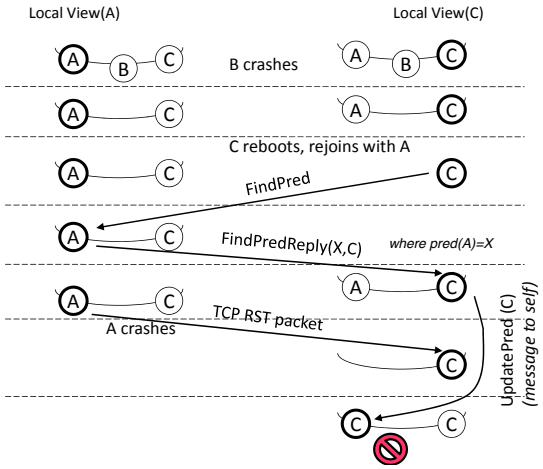


Figure 7: An inconsistency in a run of Chord. Node C has its predecessor pointing to itself while its successor list includes other nodes.

their ids, and each node has a set of "fingers" that enables it to reach exponentially larger distances on the ring.

Joining the System. To join the Chord ring, a node A first identifies its potential predecessor by querying with its id. This request is routed to the appropriate node P, which in turn replies to A. Upon receiving the reply, A inserts itself between P and P's successor, and sends the appropriate messages to its predecessor and successor nodes to update their pointers. A "stabilize" timer periodically updates these pointers.

Property: If Successor is Self, So Is Predecessor. If a predecessor of a node A equals A, then its successor must also be A (because then A is the only node in the ring). This is a safety property of Chord that had been extensively checked using MaceMC, presumably using both exhaustive search and random walks.

Scenario exhibiting inconsistency: CrystalBall found a state where node A has A as a predecessor but has another node B as its successor. This violation happens at depths that are beyond those reachable by exhaustive search from the initial state. Figure 7 shows the scenario. During live execution, several nodes join the ring and all have a consistent view of the ring. Three nodes A, B, and C are placed consecutively on the ring, i.e., A is predecessor of B and B is predecessor of C. Then B experiences a node reset and other nodes which have established TCP connection with B receive a TCP RST. Upon receiving this error, node A removes B from its internal data structures. As a consequence, Node A considers C as its immediate successor.

Starting from this state, consequence prediction detects the following scenario that leads to violation. C experiences a node reset, losing all its state. C then tries to rejoin the ring and sends a FindPred message to A. Because nodes A and C did not have an established TCP connection, A does not observe the reset of C. Node A replies to C by a FindPredReply message that shows A's successor to be C. Upon receiving this message, node C i) sets its predecessor to A; ii) stores the successor list included in the message as its successor list; and iii) sends an UpdatePred message to A's successor which, in this case, is C itself. After sending this message, C receives a transport error from A and removes A from all of its internal structures including the predecessor pointer. In other words, C's predecessor would be unset. Upon receiving the (loopback) message to itself, C observes that the predecessor is unset and then sets it to the sender of the UpdatePred message which is C. Consequently, C has its predecessor pointing to itself while its successor list includes other nodes.

Possible corrections. One possibility is for nodes to avoid sending UpdatePred messages to themselves (this appears to be a deliberate coding style in Mace Chord). If we wish to preserve such coding style, we can alternatively place a check after updating a node's predecessor: if the successor list includes nodes in addition to itself, avoid assigning the predecessor pointer to itself.

4.2.3 Example Bullet′ Bug Found

Next, we describe our experience of applying CrystalBall to the Bullet′ [23] file distribution system. The Bullet′ source sends the blocks of the file to a subset of nodes in the system; other nodes discover and retrieve these blocks by explicitly requesting them. Every node keeps a file map that describes blocks that it currently has. A node participates in the discovery protocol driven by RandTree, and peers with other nodes that have the most disjoint data to offer to it. These peering relationships form the overlay mesh.

Bullet′ is more complex than RandTree, Chord (and tree-based overlay multicast protocols) because of 1) the need for senders to keep their receivers up-to-date with file map information, 2) the block request logic at the receiver, and 3) the finely-tuned mechanisms for achieving high throughput under dynamic conditions. The starting point for our exploration was property 1):

Sender's File Map and Receivers View of it Should Be Identical. Every sender keeps a "shadow" file map for each receiver informing it which are the blocks it has not told the receiver about. Similarly, a receiver keeps a file map that describes the blocks available at the sender. Senders use the shadow file map to compute "diffs" on-demand for receivers containing information about blocks that are "new" relative to the last diff.

Senders and receivers communicate over non-blocking TCP sockets that are under control of MaceTcpTransport. This transport queues data on top of the TCP socket buffer, and refuses new data when its buffer is full.

Scenario exhibiting inconsistency: In a live run lasting less than three minutes, CrystalBall quickly identified a mismatch between a sender's file map and the receiver's view of it. The problem occurs when the diff cannot be accepted by the underlying transport. The code then clears the receiver's shadow file map, which means that the sender will never try again to inform the receiver about the blocks containing that diff. Interestingly enough, this bug existed in the original MACEDON implementation, but there was an attempt to fix it by the UCSD researchers working on Mace. The attempted fix consisted of retrying later on to send a diff to the receiver. Unfortunately, since the programmer left the code for clearing the shadow file map after a failed send, all subsequent diff computations will miss the affected blocks.

Possible corrections. Once the inconsistency is identified, the fix for the bug is easy and involves not clearing the sender's file map for the given receiver when a message cannot be queued in the underlying transport. The next successful enqueuing of the diff will then correctly include the block info.

4.3 Comparison with MaceMC

To establish the baseline for model checking performance and effectiveness, we installed our safety properties in the original version of MaceMC [22]. We then ran it for the three distributed services for which we identified safety violations. After 17 hours, exhaustive search did not identify any of the violations caught by CrystalBall, and reached the depth of only Some of the specific depths reached by the model checker are as follows 1) RandTree with 5 nodes: 12 levels, 2) RandTree with 100 nodes: 1 level, 3) Chord with 5 nodes: 14 levels, and Chord with 100 nodes: 2 levels. This illustrates the limitations of exhaustive search from the initial state.

In another experiment, we additionally employed random walk feature of MaceMC. Using this setup, MaceMC identified some of the bugs found by CrystalBall, but it still failed to identify 2 Randtree, 2 Chord, and 3 Bullet′ bugs found by CrystalBall. In Bullet′, MaceMC found no bugs despite the fact that the search lasted 32 hours. Moreover, even for the bugs found, the long list of events that lead to a violation (on the order of hundreds) made it difficult for the programmer to identify the error (we spent five hours tracing one of the violations involving 30 steps). Such a long event list is unsuitable for execution steering, because it describes a low probability way of reaching the final erroneous state. In contrast, CrystalBall identified violations that are close to live executions and therefore more likely to occur in the immediate future.

4.4 Execution Steering Experience

We next evaluate the capability of CrystalBall as a runtime mechanism for steering execution away from previously unknown bugs.

4.4.1 RandTree Execution Steering

To estimate the impact of execution steering on deployed systems, we instructed the CrystalBall controller to check for violations of RandTree safety properties (including the one described in Section 4.2.1). We ran a live churn scenario in which one participant (process in a cluster) per minute leaves and enters the system on average, with 25 tree nodes mapped onto 25 physical cluster machines. Every node was configured to run the model checker. The experiment ran for 1.4 hours and resulted in the following data points, which suggest that in practice the execution steering mechanism is not disruptive for the behavior of the system.

When CrystalBall is not active, the system goes through a total of 121 states that contain inconsistencies. When only the immediate safety check but not the consequence prediction is active, the immediate safety check engages 325 times, a number that is higher because blocking a problematic action causes further problematic actions to appear and be blocked successfully. Finally, we consider the run in which both execution steering and the immediate safety check (as a fallback) are active. Execution steering detects a future inconsistency 480 times, with 65 times concluding that changing the behavior is unhelpful and 415 times modifying the behavior of the system. The immediate safety check fallback engages 160 times. Through a combined action of execution steering and immediate safety check, CrystalBall avoided all inconsistencies, so there were no uncaught violations (false negatives) in this experiment.

To understand the impact of CrystalBall actions on the overall system behavior, we measured the time needed for nodes to join the tree. This allowed us to empirically address the concern that TCP reset and message blocking actions can in principle cause violations of liveness properties (in this case extending the time nodes need to join the tree). Our measurements indicated an average node join times between 0.8 and 0.9 seconds across different experiments, with variance exceeding any difference between the runs with and without CrystalBall. In summary, CrystalBall changed system actions 415 times (2.77% of the total of 14956 actions executed), avoided all specified inconsistencies, and did not degrade system performance.

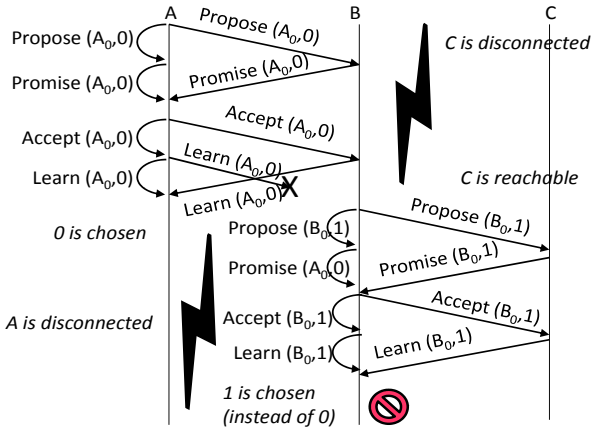


Figure 8: Scenario that exposes a previously reported Paxos violation of a safety property (two different values are chosen in the same instance).

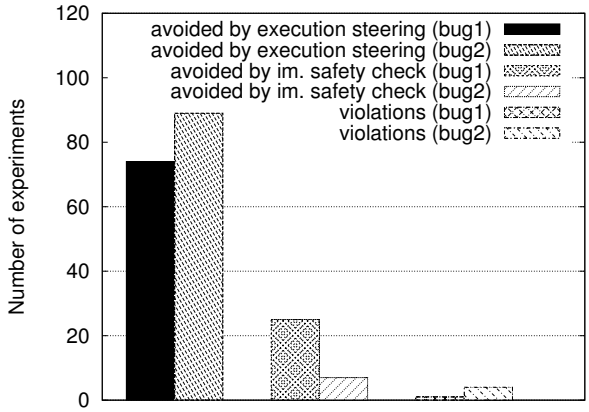


Figure 9: In 200 runs that expose Paxos safety violations due to two injected errors, CrystalBall successfully avoided the inconsistencies in all but 1 and 4 cases, respectively.

4.4.2 Paxos Execution Steering

Paxos [26] is a well known fault-tolerant protocol for achieving consensus in distributed systems. Recently, it has been successfully integrated in a number of deployed [4, 28] and proposed [19] distributed systems. In this section, we show how execution steering can be applied to Paxos to steer away from realistic bugs that have occurred in previously deployed systems [4, 28]. The Paxos protocol includes five steps:

1. A leader tries to take the leadership position by sending Prepare messages to acceptors, and it includes a unique round number in the message.
2. Upon receiving a Prepare message, each acceptor consults the last promised round number. If the message's round number is greater than that number, the acceptor responds with a Promise message that contains the last accepted value if there is any.
3. Once the leader receives a Promise message from the majority of acceptors, it broadcasts an Accept request to all acceptors. This message contains the value of the Promise message with the highest round number, or is any value if the responses reported no proposals.
4. Upon the receipt of the Accept request, each acceptor accepts it by broadcasting a Learn message containing the Accepted value to the learners, unless it had made a promise to another leader in the meanwhile.
5. By receiving Learn messages from the majority of the nodes, a learner considers the reported value as chosen.

The implementation we used was a baseline Mace Paxos implementation that includes a minimal set of features. In general, a physical node can implement one or more of the roles (leader, acceptor, learner) in the Paxos algorithm; each node plays all the roles in our experiments. The safety property we installed is the original Paxos safety property: at most one value can be chosen, across all nodes. The first bug we injected [28] is related to an implementation error in step 3, and we refer to it as *bug1*: once the leader receives the Promise message from the majority of nodes, it creates the Accept request by using the submitted value from the last Promise message instead of the Promise message with highest round number. Because the rate at which the violation (due to the injected error) occurs was low, we had to schedule some events to lead the live run toward the violation in a repeatable way. The setup we use comprises 3 nodes and two rounds, without any artificial packet delays. As illustrated in Figure 8, in the first round the communication between node C and the other nodes is broken. Also, a Learn packet is dropped from A to B. At the end of this round, A chooses the value proposed by itself (0). In the second round, the communication between A and other nodes is broken. At the end of this round, the value proposed by B (1) is chosen by B itself.

The second bug we injected (inspired by [4]) involves keeping a promise made by an Acceptor, even after crashes and reboots. As pointed in [4], it is often difficult to implement this aspect correctly, especially under various hardware failures. Hence, we inject an error in the way a promise is kept by not writing it to disk (we refer to it as *bug2*). To expose this bug we use a scenario similar to the one used for *bug1*, with the addition of a reset of node B.

To stress test CrystalBall's ability to avoid inconsistencies at runtime, we repeat the live scenarios in the cluster 200 times (100 times for each bug) while vary-

ing the time between rounds uniformly at random between 0 and 20 seconds. As we can see in Figure 9, CrystalBall's execution steering is successful in avoiding the inconsistency at runtime 74% and 89% of the time for *bug1* and *bug2*, respectively. In these cases, CrystalBall starts model checking after node C reconnects and receives checkpoints from other participants. After running the model checker for 3.3 seconds, C successfully predicts that the scenario in the second round would result in violation of the safety property, and it then installs the event filter. The avoidance by execution steering happens when C rejects the Propose message sent by B. Execution steering is more effective for *bug2* than for *bug1*, as the former involves resetting B. This in turn leaves more time for the model checker to rediscover the problem by: i) consequence prediction, or ii) replaying a previously identified erroneous scenario. Immediate safety check engages 25% and 7% of the time, respectively (in cases when model checking did not have enough time to uncover the inconsistency), and prevents the inconsistency from occurring later, by dropping the Learn message from C at node B. CrystalBall could not prevent the violation for only 1% and 4% of the runs, respectively. The cause for these false negatives was the incompleteness of the set of checkpoints.

4.5 Performance Impact of CrystalBall

Memory, CPU, and bandwidth consumption. Because consequence prediction runs in a separate process that is most likely mapped to a different CPU core on modern processors, we expect little impact on the service performance. In addition, since the model checker does not cache previously visited states (it only stores their hashes) the memory is unlikely to become a bottleneck between the model-checking CPU core and the rest of the system.

One concern with state exploration such as model-checking is the memory consumption. Figure 10 shows the consequence prediction memory footprint as a function of search depth for our RandTree experiments. As expected, the consumed memory increases exponentially with search depth. However, since the effective CrystalBall's search depth in is less than 7 or 8, the consumed memory by the search tree is less than 1MB and can thus easily fit in the L2 or L3 (most recently) cache of the state of the art processors. Having the entire search tree in-cache reduces the access rate to main memory and improves performance.

In the deep online debugging mode, the model checker was running for 950 seconds on average in the 100-node case, and 253 seconds in the 6-node case. When running in the execution steering mode (25 nodes), the model checker ran for an average of about 10 seconds. The checkpointing interval was 10 seconds.

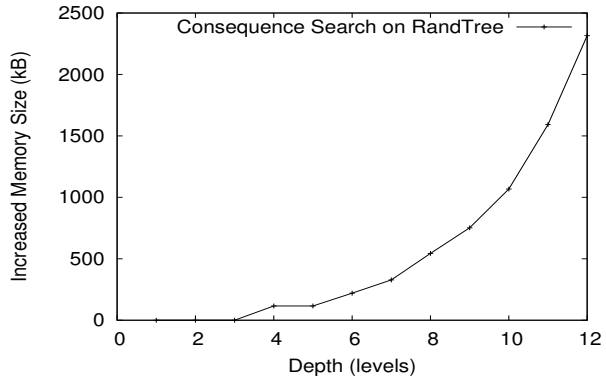


Figure 10: The memory consumed by consequence prediction (RandTree, depths 7 to 8) fits in an L2 CPU cache.

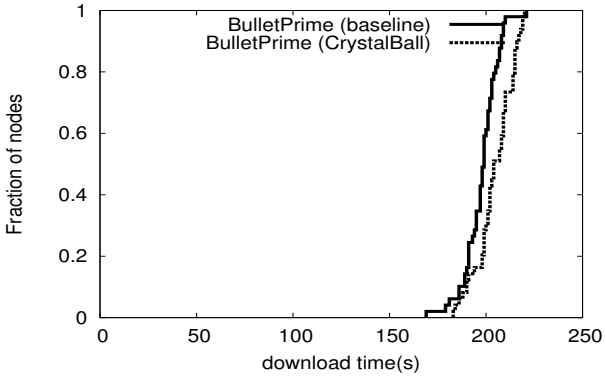


Figure 11: CrystalBall slows down Bullet′ by less than 10% for a 20 MB file download.

The average size of a RandTree node checkpoint is 176 bytes, while a Chord checkpoint requires 1028 bytes. Average per-node bandwidth consumed by checkpoints for RandTree and Chord (100-nodes) was 803 bps and 8224 bps, respectively. These figures show that overheads introduced by CrystalBall are low. Hence, we did not need to enforce any bandwidth limits in these cases.

Overhead from Checking Safety Properties. In practice we did not find the overhead of checking safety properties to be a problem because: i) the number of nodes in a snapshot is small, ii) the most complex of our properties have $O(n^2)$ complexity, where n is the number of nodes, and iii) the state variables fit into L2 cache.

Overall Impact. Finally, we demonstrate that having CrystalBall monitor a bandwidth-intensive application featuring a non-negligible amount of state such as Bullet′ does not significantly impact the application's performance. In this experiment, we instructed 49 Bullet′ instances to download a 20 MB file. Bullet′ is not a CPU intensive application, although computing the next block to request from a sender has to be done quickly. It is therefore interesting to note that in 34 cases during

this experiment the Bullet′ code was competing with the model checker for the Xeon CPU with hyper-threading. Figure 11 shows that in this case using CrystalBall reduced performance by less than 5%. Compressed Bullet′ checkpoints were about 3 kB in size, and the bandwidth that was used for checkpoints was about 30 Kbps per node (3% of a node's outbound bandwidth of 1 Mbps). The reduction in performance is therefore primarily due to the bandwidth consumed by checkpoints.

5 Related Work

Debugging distributed systems is a notoriously difficult and tedious process. Developers typically start by using an ad-hoc logging technique, coupled with strenuous rounds of writing custom scripts to identify problems. Several categories of approaches have gone further than the naive method, and we explain them in more detail in the remainder of this section.

Collecting and Analyzing Logs. Several approaches (Magpie [2], Pip [34]) have successfully used extensive logging and off-line analysis to identify performance problems and correctness issues in distributed systems. Unlike these approaches, CrystalBall works on deployed systems, and performs an online analysis of the system state.

Deterministic Replay with Predicate Checking. Friday [14] goes one step further than logging to enable a gdb-like replay of distributed systems, including watch points and checking for global predicates. WiDS-checker [28] is a similar system that relies on a combination of logging/checkpointing to replay recorded runs and check for user predicate violations. WiDS-checker can also work as a simulator. In contrast to replay- and simulation-based systems, CrystalBall explores additional states and can steer execution away from erroneous states.

Online Predicate Checking. Singh *et al.* [40] have advocated debugging by online checking of distributed system state. Their approach involves launching queries across the distributed system that is described and deployed using the OverLog/P2 [40] declarative language/runtime combination. D^3S [27] enables developers to specify global predicates which are then automatically checked in a deployed distributed system. By using binary instrumentation, D3S can work with legacy systems. Specialized *checkers* perform predicate-checking topology on snapshots of the nodes' states. To make the snapshot collection scalable, the checker's *snapshot neighborhood* can be manually configured by the developer. This work has shown that it is feasible to collect snapshots at runtime and check them against a set of user-specified properties. CrystalBall advances the state-of-the-art in online debugging in two main directions: 1) it employs an efficient algorithm for model checking from a live state to search for bugs "deeper" and "wider" than in the live run, and it 2) enables execution steering to automatically prevent previously unidentified bugs from manifesting themselves in a deployed system.

Model Checking. Model checking techniques for finite state systems [16, 20] have proved successful in analysis of concurrent finite state systems, but require the developer to manually abstract the system into a finite-state model which is accepted as the input to the system. Early efforts on explicit-state model checking of C and C++ implementations [31, 30, 46] have primarily concentrated on a single-node view of the system.

MODIST [45] and MaceMC [22] represent the state-of-the-art in model checking distributed system implementations. MODIST [45] is capable of model checking unmodified distributed systems; it orchestrates state space exploration across a cluster of machines. MaceMC runs state machines for multiple nodes within the same process, and can determine safety and liveness violations spanning multiple nodes. MaceMC's exhaustive state exploration algorithm limits in practice the search depth and the number of nodes that can be checked. In contrast, CrystalBall's consequence prediction allows it to achieve significantly shorter running times for similar depths, thus enabling it to be deployed at runtime. In [22] the authors acknowledge the usefulness of prefix-based search, where the execution starts from a given supplied state. Our work addresses the question of obtaining prefixes for prefix-based search: we propose to directly feed into the model checker states as they are encountered in live system execution. Using CrystalBall we found bugs in code that was previously debugged in MaceMC and that we were not able to reproduce using MaceMC's search. In summary, CrystalBall differs from MODIST and MaceMC by being able to run state space exploration from live state. Further, CrystalBall supports execution steering that enables it to automatically prevent the system from entering an erroneous state.

Cartesian abstraction [1] is a technique for over-approximating state space that treats different state components independently. The independence idea is also present in our consequence prediction, but, unlike over-approximating analyses, bugs identified by consequence search are guaranteed to be real with respect to the model explored. The idea of disabling certain transitions in state-space exploration appears in partial-order reduction (POR) [15],[13]. Our initial investigation suggests that a POR algorithm takes considerably longer than the consequence prediction algorithm. The advantage of POR is its completeness, but completeness is of second-order importance in our case because no complete search can terminate in a reasonable amount of time for state spaces of distributed system implementations.

Runtime Mechanisms. In the context of operating systems, researchers have proposed mechanisms that safely re-execute code in a changed environment to avoid errors [33]. Such mechanisms become difficult to deploy in the context of distributed systems. Distributed transactions are a possible alternative to execution steering, but involve several rounds of communication and are inapplicable in environments such as wide-area networks. A more lightweight solution involves forming a FUSE [11] failure group among all nodes involved in a join process. Making such approaches feasible would require collecting snapshots of the system state, as in CrystalBall. Our execution steering approach reduces the amount of work for the developer because it does not require code modifications. Moreover, our experimental results show an acceptable computation and communication overhead.

In Vigilante [9] and Bouncer [8], end hosts cooperate to detect and inform each other about worms that exploit even previously unknown security holes. Hosts protect themselves by generating filters that block bad inputs. Relative to these systems, CrystalBall deals with distributed system properties, and predicts inconsistencies before they occur.

Researchers have explored modifying actions of concurrent programs to reduce data races [18] by inserting locks in an approach that does not employ running static analysis at runtime. Approaches that modify state of a program at runtime include [10, 36]; these approaches enforce program invariants or memory consistency without computing consequences of changes to the state.

6 Conclusions

We presented a new approach for improving the reliability of distributed systems, where nodes predict and avoid inconsistencies before they occur, even if they have not manifested in any previous run. We believe that our approach is the first to give running distributed system nodes access to such information about their future. To make our approach feasible, we designed and implemented consequence prediction, an algorithm for selectively exploring future states of the system, and developed a technique for obtaining consistent information about the neighborhood of distributed system nodes. Our experiments suggest that the resulting system, CrystalBall, is effective in finding bugs that are difficult to detect by other means, and can steer execution away from inconsistencies at runtime.

Acknowledgments

We thank our shepherd Arvind Krishnamurthy and the anonymous reviewers who provided excellent feedback. We also thank Barbara Jobstmann for useful discussions, James Anderson for his help with the Mace Paxos implementation, and Charles Killian for answering questions about MaceMC. Nikola Knežević was funded in part by a grant from the Hasler foundation (grant 2103).

References

[1] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In *TACAS*, 2001.

[2] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[3] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth Content Distribution in Cooperative Environments. In *SOSP*, October 2003.

[4] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: an Engineering Perspective. In *PODC*, 2007.

[5] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[6] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *SIGMETRICS*, June 2002.

[7] Yang Chu, S. G. Rao, S. Seshan, and Hui Zhang. A Case for End System Multicast. *Selected Areas in Communications, IEEE Journal on*, 20(8):1456–1471, Oct 2002.

[8] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. In *SOSP*, 2007.

[9] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*, October 2005.

[10] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *OOPSLA*, 2003.

[11] John Dunagan, Nicholas J. A. Harvey, Michael B. Jones, Dejan Kostić, Marvin Theimer, and Alec Wolman. FUSE: Lightweight Guaranteed Distributed Failure Notification. In *OSDI*, 2004.

[12] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[13] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[14] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *NSDI*, 2007.

[15] Patrice Godefroid and Pierre Wolper. A Partial Approach to Model Checking. *Inf. Comput.*, 110(2):305–326, 1994.

[16] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[17] Navendu Jain, Prince Mahajan, Dmitry Kit, Praveen Yalagandula, Mike Dahlin, and Yin Zhang. Network Imprecision: A New Consistency Metric for Scalable Monitoring. In *OSDI*, December 2008.

[18] Muhammad Umar Janjua and Alan Mycroft. Automatic Correction to Safety Violations. In *Thread Verification (TV06)*, 2006.

[19] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus Routing: The Internet as a Distributed System. In *NSDI*, San Francisco, April 2008.

[20] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS*, 1990.

[21] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[22] Charles E. Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[23] Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat. Maintaining High Bandwidth under Dynamic Network Conditions. In *USENIX ATC*, 2005.

[24] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *USITS*, 2003.

[25] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Com. of the ACM*, 21(7):558–565, 1978.

[26] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[27] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^3S: Debugging Deployed Distributed Systems. In *NSDI*, 2008.

[28] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[29] D. Manivannan and Mukesh Singhal. Asynchronous Recovery Without Using Vector Timestamps. *J. Parallel Distrib. Comput.*, 62(12):1695–1728, 2002.

[30] Madanlal Musuvathi and Dawson R. Engler. Model Checking Large Network Protocol Implementations. In *NSDI*, 2004.

[31] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[32] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative Execution in a Distributed File System. In *SOSP*, 2005.

[33] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies—a Safe Method to Survive Software Failures. In *SOSP*, 2005.

[34] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[35] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A Public DHT Service and Its Uses. In *SIGCOMM*, 2005.

[36] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, 2004.

[37] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*, 2004.

[38] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *SOSP*, 2001.

[39] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[40] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using Queries for Distributed Monitoring and Forensics. *SIGOPS Oper. Syst. Rev.*, 40(4):389–402, 2006.

[41] Sudarshan M. Srinivasan, Srikanth K, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX ATC*, 2004.

[42] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[43] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*, December 2002.

[44] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. Technical report, EPFL, `http://infoscience.epfl.ch/record/124918`, 2008.

[45] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, April 2009.

[46] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.