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

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 article 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.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; H.4.3 [**Information Systems Applications**]: Communications Applications

General Terms: Experimentation, Reliability

Additional Key Words and Phrases: distributed systems, consequence prediction, reliability, execution steering, enforcing safety properties

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 [John et al. 2008] to improve its routing reliability by further treating routing as a distributed consensus problem [Lamport 1998]. 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 con-

...

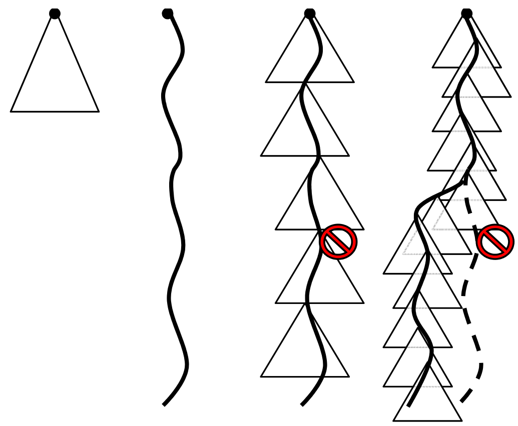

Fig. 1. Execution path coverage by a) classic model checking, b) replay-based or live predicate checking, c) CrystalBall in deep online debugging mode, and d) CrystalBall in execution steering mode. A triangle represents the state space searched by the model checker; a full line denotes an execution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

tain errors arising from complexities of realistic distributed environments or simply coding errors [Liu et al. 2008]. 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.

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 more relevant states than 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 non-determinism 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 article 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 the predicted inconsistencies;

—We describe CrystalBall [Yabandeh et al. 2009], the implementation of our approach on top of the Mace framework [Killian et al. 2007]. 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, in the case of remaining 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 later we show how CrystalBall predicts and avoids it. The inconsistency appears in the Mace [Killian et al. 2007] implementation of the RandTree overlay. RandTree implements a random, degree-constrained overlay tree designed to be resilient to node failures and network partitions. The 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 [Kostić et al. 2005] and RanSub [Kostić et al. 2003]. In general, trees are used in a variety of multicast scenarios [Castro et al. 2003; Chu et al. 2002] and data collection/monitoring environments [Jain et al. 2008]. Inconsistencies in these environments translate into denial of service to the 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

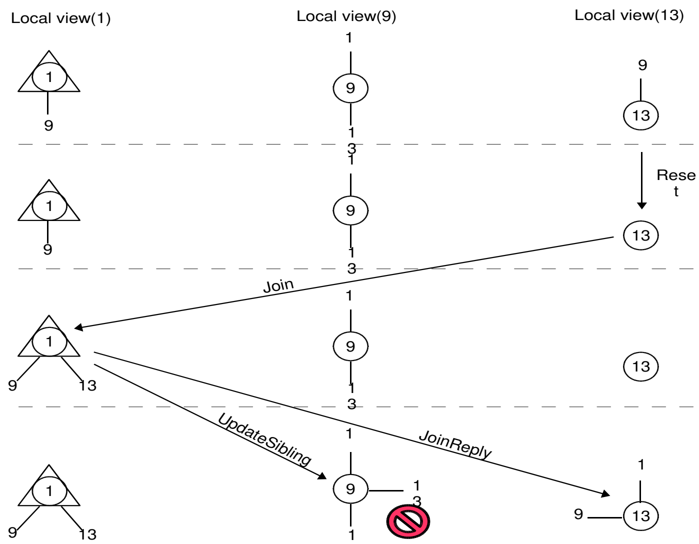

Fig. 2. An inconsistency in a run of RandTree; Safety property: children and siblings are disjoint lists

period of three years, as well as debugged using an existing model checking approach [Killian et al. 2007] but, to our knowledge, this inconsistency has not been discovered before (see Section 5 for some of the additional inconsistencies 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 turns out to be complicated 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 message has not been forwarded by the root, then the receiver forwards the request back 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 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 [Vahdat et al. 2002] 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 the 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 [Killian et al. 2007] and reported a number of safety and liveness bugs in Mace implementations. However, due to an exponential explosion of state space, 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 [Killian et al. 2007] 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 some of our experiments in Section 5). 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, this state is fed to the model checker as initial state and the model checker will predict the state at the end of Figure 2 as a possible future inconsistency. In summary, instead of focusing only on 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 checking the live state of the running system, 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 a predicted 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.

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[1].

Consequence Prediction. We believe that inconsistency detection and execution steering are compelling reasons to use an approach where a model checker is deployed online to find future inconsistencies. To make this approach feasible, it is essential to have a model checking technique capable of quickly discovering potential inconsistencies till significant depths in a very short amount of time. The previous model checking technique is not sufficient for this purpose: when we tried deploying it online, by the time a future inconsistency was identified, the system had already passed the execution depth at which the inconsistency occurs. We need an exploration technique that is sufficiently fast and focused to be able to discover a future inconsistency faster than the time that it takes node interaction to cause the inconsistency in a distributed system. We present such an exploration technique,

[1]Besides external events (such as additional node joining), a node typically has sufficient amount of choice to allow it to make progress (e.g., multiple bootstrap nodes). In general however, it is possible that enforcing safety can reduce liveness.

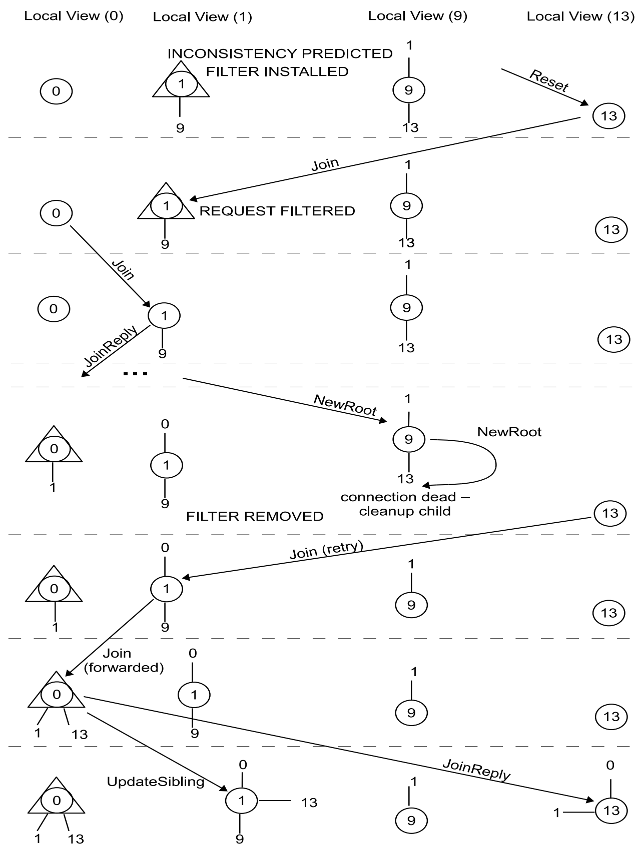

Fig. 3. An example execution sequence that avoids the inconsistency from Figure 2; thanks to execution steering.

termed *consequence prediction*.

Consequence prediction focuses on exploring causally related chains of events. Our system identifies the scenario in Figure 2 by running consequence prediction on node n_1. Consequence prediction considers, among others, the Reset action on node n_{13}. It then uses the fact that the Reset action brings the node into a state where it can issue a Join request. Even though there are many transitions that a distributed system could take at this point, consequence prediction focuses on the transitions that were enabled by the recent state change. It will therefore examine the consequences of the response of n_1 to the Join request and, using the knowledge of the state of its neighborhood, discover a possible inconsistency that could occur in n_9. Consequence prediction also explores other possible sequences of events, but, as we explain in Section 3.2, it avoids certain sequences, which makes it faster than applying the standard search to the same search depth.

2. BACKGROUND

We next present a simple model of distributed systems and describe a basic model checking algorithm based on breadth-first search and state caching.

2.1 System Model

Figure 4 describes a simple model of a distributed system. We use this model to describe system execution at a high level of abstraction, describe an existing model checking algorithm, and present our new algorithm, consequence prediction. (The model does not attempt to describe our technique for obtaining consistent snapshots.)

System state. The state of the entire distributed system is given by 1) local state of each node, and 2) in-flight network messages. We assume a finite set of node identifiers N (corresponding to, for example, IP addresses). Each node $n \in N$ has a local state $L(n) \in S$. Local state models all node-local information such as explicit state variables of the distributed node implementation, the status of timers, and the state that determines application calls. Network state is given by in-flight messages, I. We represent each in-flight message by a pair (N, M) where N is the destination node of the message and M is the remaining message content (including sender node information and message body).

Node behavior. Each node in our system runs the same state-machine implementation. The state machine is given by two kinds of handlers: a message handler executes in response to a network message; an internal handler executes in response to a node-local event such as a timer and an application call.

We represent message handlers by a set of tuples H_M. The condition $((s_1, m), (s_2, c)) \in H_M$ means that, if a node is in state s_1 and it receives a message m, then it transitions into state s_2 and sends the set c of messages. Each element $(n', m') \in c$ is a message with target destination node n' and content m'. Internal node action handler is analogous to a message handler, but it does not consume a network message. Instead, $((s_1, a), (s_2, c)) \in H_A$ represents handling of an internal node action $a \in A$. (In both handlers, if c is the empty set, it means that the handler did not generate any messages.)

System behavior. The behavior of the system specifies one step of a transition from one global distributed system state (L, I) to another global state (L', I'). We denote this transition by $(L, I) \leadsto (L', I')$ and describe it in Figure 4 in terms of handlers H_M and H_A. The handler that sends the message directly inserts the message into the network state I, whereas the handler receiving the message simply removes it from I. To keep the model simple, we assume that transport errors are particular messages, generated and processed by message handlers.

2.2 Model-Checking Distributed Systems

Figure 5 presents a standard breadth-first search (BFS) for finding safety violations in a transition system given by relation $\leadsto$. We describe BFS here, because it is easier to understand. In practice, model checkers usually use bounded depth-first search (DFS), which is more memory efficient. The search starts from a given global state firstState, which in the standard approach, is the initial state of the system. The search systematically explores reachable global states at larger and

basic notions:
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 – internal node actions (timers, application calls)

system state : $(L, I) \in G, \quad G = 2^{N \times S} \times 2^{N \times M}$
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 :

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

internal node action (timer, application calls) :

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

Fig. 4. A Simple Model of a Distributed System

larger depths and checks whether the states satisfy the given property condition. In practice, the number of reachable states is very large and the search needs to be terminated upon exceeding some bound such as running time or search depth. The condition of exceeding some bound is denoted StopCriterion in Figure 5.

```
proc findErrors(firstState : G, property : (G → boolean)) {
  explored = emptySet(); errors = emptySet();
  frontier = emptyQueue();
  frontier.addLast(firstState);
  while (!StopCriterion) {
    state = frontier.popFirst();
    if (!property(state))
      errors.add(state);
    explored.add(hash(state));
    foreach (nextState where (state ⇝ nextState))
      if (!explored.contains(hash(nextState)))
        frontier.addLast(nextState);
  }
}
```

Fig. 5. The algorithm for finding errors in model checkers using state space exploration. The presented algorithm is based on BFS (Breadth First Search) approach.

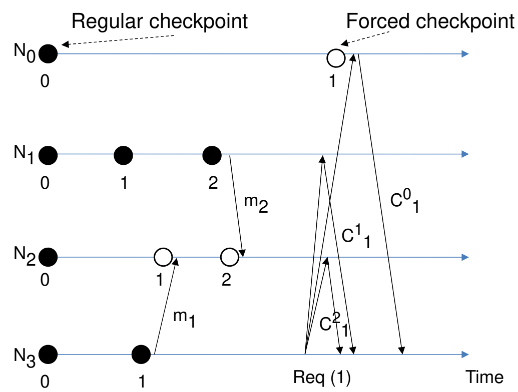

Fig. 6. Example illustrating the consistent snapshot collection algorithm. Black ovals represent regular checkpoints. Messages m_1 and m_2 force checkpoints (white ovals) to be taken before messages are processed at nodes 2 and 1, respectively, and so does the checkpoint request from node 3 when it arrives at node 0.

2.3 Consistent Global Snapshots

Examining global state of a distributed system is useful in a variety of scenarios, such as checkpointing/recovery, debugging, and in our case, running a model checking algorithm in parallel with the system. A *snapshot* consists of *checkpoints* of nodes' states. For the snapshot to be useful, it needs to be consistent. There has been a large body of work in this area, starting with the seminal paper by Chandy and Lamport [Chandy and Lamport 1985]. We next describe one of the recent algorithms for obtaining consistent snapshots [Manivannan and Singhal 2002]. The general idea is to collect a set of checkpoints which do not violate the happens-before relationship [Lamport 1978] established by messages sent by the distributed service.

In this algorithm, the runtime of each node n_i keeps track of the checkpoint number cn_i (the role of checkpoint number is similar to the Lamport's logical clock [Lamport 1978]). Whenever n_i sends a message M, it stores cn_i in it (denote this value $M.cn$). When node n_j receives a message, it compares cn_j with $M.cn$. If $M.cn > cn_j$, then n_j takes a checkpoint C, assigns $C.cn = M.cn$, and sets $cn_j = M.cn$. This is the key step of the algorithm that avoids violating the happens-before relationship. A node n_i can take snapshots on its own, and this is done whenever the cn_i is locally incremented, which happens periodically.

To collect the required checkpoints, a node n_i sends a checkpoint request message containing a checkpoint request number cr_i. Upon receiving the request, a node n_j responds with the appropriate checkpoint. There are two cases: 1) if $cr_i > cn_j$ (the request number is greater than any number n_j has seen), then n_j takes a checkpoint, stamps it with $C.cn = cr_i$, sets $cn_j = cr_i$, and sends that checkpoint; 2) if $cr_i \leq cn_j$, the request is for a checkpoint taken in the past, and n_j responds with the earliest checkpoint C for which $C.cn \geq cr_i$. The example depicted in Figure 6, includes different scenarios that the algorithm forces taking checkpoints in order to maintain the checkpoints globally consistent.

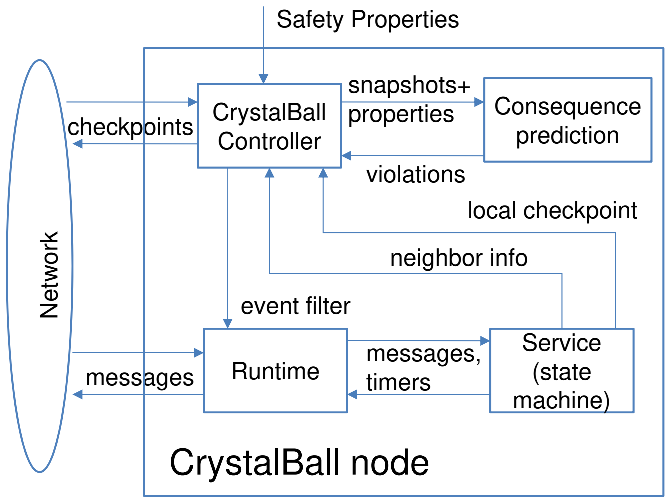

Fig. 7. High-level overview of CrystalBall. This figure depicts the high-level architecture of CrystalBall and its main components.

3. CRYSTALBALL DESIGN

We next sketch the design of CrystalBall. Figure 7 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 [Killian et al. 2007; Lamport 1978; 1998; Rodriguez et al. 2004; Schneider 1990].

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, including the local node. 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.

3.1 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 nodes 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 thus needs to 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. Also 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 received checkpoints 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.

3.1.1 *Discovering and Managing Snapshot Neighborhoods.* To propagate checkpoints, the checkpoint manager needs to know the set of a node's neighbors. This set depends on the 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.

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

3.1.2 *Enforcing Snapshot Consistency.* CrystalBall ensures that the neighborhood snapshot corresponds to a consistent view of the distributed system at some point of logical time. Our starting point is a technique similar to the one described in Section 2.3.

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 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.

3.1.3 *Checkpoint Content.* Although the total footprint of some services might be very large, this need not necessarily be reflected in checkpoint size. For example, the Bullet′ [Kostić et al. 2005] 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 recreated from a remote node's checkpoint in the model checker.

3.1.4 *Managing Checkpoint Storage.* The checkpoint manager keeps track of checkpoints via their checkpoint numbers. Over the course of its operation, a node can collect a large number of checkpoints, and a long-running system might demand an excessive amount of memory and storage for this task. It is therefore important to prune old checkpoints in a way that nevertheless leaves the ability to gather consistent snapshots.

Our approach to managing checkpoint storage is to enforce a per-node storage quota for checkpoints. Older checkpoints are removed first to make room. Removing older checkpoints might cause a checkpoint request to fail when the request is asking for a checkpoint that is outside the range of remaining checkpoints at the node. In this case, the node responds negatively to the checkpoint requester and inserts its current checkpoint number in the response ($R.cn = cn_i$). Then, upon receiving the responses from all nodes in the snapshot neighborhood, the requestor chooses the greatest among the $R.cn$ received, and initiates another snapshot round. Provided that the rate at which the snapshots are removed is not greater than the rate at which the nodes are communicating, this second snapshot collection will likely succeed.

3.1.5 *Managing Bandwidth Consumption.* For a large class of services, the relevant per-node state is relatively small, e.g., a few KB. It is nevertheless important to limit bandwidth consumed by state checkpoints for a number of reasons: 1) sending large amounts of data might congest the node's outbound link, and 2) consuming bandwidth for checkpoints might adversely affect the performance and the reaction time of the system.

To reduce the amount of checkpoint data transmitted, CrystalBall can use a number of techniques. First, it can employ "diffs" that enable a node to transmit only parts of state that are different from the last sent checkpoint. Second, the checkpoints can be compressed on-the-fly. Finally, CrystalBall can enforce a bandwidth limit by: 1) making the checkpoint data be a fraction of all data sent by a node, or 2) enforcing an absolute bandwidth limit (e.g., 10 Kbps). If the checkpoint manager is above the bandwidth limit, it responds with a negative response to a checkpoint request and the requester temporarily removes the node from the cur-

```
proc findConseq(currentState : G, property : (G → boolean))
  explored = emptySet(); errors = emptySet();
  localExplored = emptySet();
  frontier = emptyQueue();
  frontier.addLast(currentState);
  while (!STOP_CRITERION)
    state = frontier.popFirst();
    if (!property(state))
      errors.add(state); // predicted inconsistency found
    explored.add(hash(state));
    foreach ((n,s) ∈ state.L) // node n in local state s
      // process all network handlers
      foreach (((s,m),(s',c)) ∈ H_M where (n,m) ∈ state.I)
         // node n handles message m according to st. machine
         addNextState(state,n,s,s',{m},c);
      // process local actions only for fresh local states
      if (!localExplored.contains(hash(n,s)))
        foreach (((s,a),(s',c)) ∈ H_A)
          addNextState(state,n,s,s',{},c);
      localExplored.add(hash(n,s));

proc addNextState(state,n,s,s',c0,c)
  nextState.L = (state.L \ {(n,s)}) ∪ {(n,s')};
  nextState.I = (state.I \ c0) ∪ c;
  if (!explored.contains(hash(nextState)))
    frontier.addLast(nextState);
```

Fig. 8. Consequence Prediction Algorithm

rent snapshot. A node that wishes to reduce its inbound bandwidth consumption can reduce the rate at which it requests checkpoints from other nodes.

3.2 Consequence Prediction Algorithm

The key to enabling fast prediction of future inconsistencies in CrystalBall is our consequence prediction algorithm. The idea of the algorithm is to avoid exploring internal (local) actions of nodes whose local state was encountered previously at a smaller depth in the search tree. Recall that the state of the entire system contains the local states of each node in the neighborhood. A standard search avoids re-exploring actions of states whose particular combination of local states was encountered previously; it achieves this by storing hashes of the entire *global state*. We found this standard approach to be too slow for our purpose because of the high branching in the search tree (Figure 13 shows an example for RandTree). Consequence prediction therefore additionally avoids exploring internal actions of a node if this node was already explored with the same local state (regardless of the states of other nodes). Consequence prediction implements this policy by maintaining an additional hash table that stores hashes of visited *local states* for each node.

Figure 8 shows the consequence prediction pseudo code. In its overall structure, the algorithm is similar to the standard state-space search in Figure 5. (We present the algorithm at a more concrete level, where the relation $\leadsto$ is expressed in terms of action handlers H_A and H_M introduced in Figure 4.) In fact, if we omitted the test in Line 17,

if (!localExplored.contains(hash(n,s)))

the algorithm would reduce precisely to Figure 5.

In Line 8 of Figure 8, the algorithm checks whether the explored state satisfies the desired safety properties. To specify the properties, the developer can use a simple language [Killian et al. 2007] that supports universal and existential quantifiers, comparison operators, state variables, and function invocations.

3.2.1 *Exploring Chains.* To understand the intuition behind consequence search, we identify certain executions that consequence prediction does and does not explore. Consider the model of Figure 4 and the algorithm in Figure 8. We view local actions as triggering a sequence of message exchanges. Define an *event chain* as a sequence of steps $e_A e_{M1} \dots e_{Mp}$ (for $p \geq 0$) where the first element e_A is an internal node action (element of H_A in Figure 4, representing application or scheduler event), and where the subsequent elements e_{Mi} are network events (elements of H_M in Figure 4). Clearly, every finite execution sequence can be written as a concatenation of chains.

Executions explored: Consequence prediction explores all chains that start from the current state (i.e. the live state of the system). This is because 1) consequence search prunes only local actions, not network events, 2) a chain has only one local action, namely the first one, and 3) at the beginning of the search the `localExplored` hash tables are empty, so no pruning occurs.

Executions not necessarily explored: Consider a chain C of the form $e_A e_{M1} \dots e_{Mp}$ and another chain C' of the form $e'_A e'_{M1} \dots e'_{Mq}$, where e'_A is a local action applicable to a node n. Then consequence prediction will explore the concatenation CC' of these chains if there is no previously explored reachable state (at the depth less than p) at which the node n has the same state as after C. As a special case, suppose chains C and C' involve disjoint sets of nodes. Then consequence prediction will explore both C and C' in isolation. However, it will not explore the concatenation CC', nor a concatenation $C_1 C'_1$ where C_1 is a non-empty prefix of C and C'_1 is non-empty prefix of C'.

The notion of avoiding interleavings is related to the techniques of partial order reduction [Godefroid and Wolper 1994] and dynamic partial order reduction [Flanagan and Godefroid 2005; Sen and Agha 2006], but our algorithm uses it as a heuristic that does not take into account the user-defined properties being checked. Note that user-defined properties in CrystalBall can arbitrarily relate local states of different nodes. In the example of disjoint nodes above, a partial order reduction technique would explore CC' but, assuming perfect independence information, would not explore C'. Figure 9 illustrates this difference; the Appendix presents a larger example. The preference for a longer execution makes partial order reduction less appropriate for our scenario where we need to impose a bound on search depth.

Note that $hash(n, s)$ in Figure 8 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

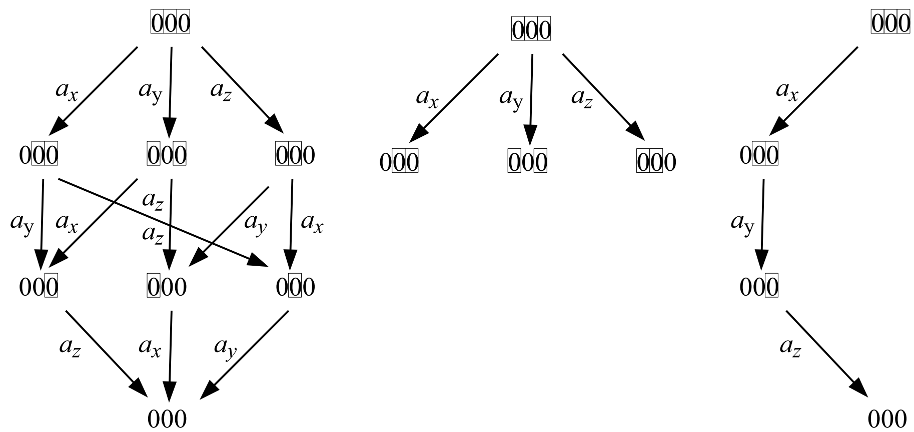

Fig. 9. Full state space, consequence search, and partial order reduction in an example with internal actions of three distinct nodes

or she does not want to be included in the hash function, improving the performance of Consequence Prediction (our experiments in Section 5 make use of this mechanism). Instead of holding all encountered hashes, the hash table could be designed as a bounded cache of explored hash entries to fit into the L2 cache or main memory, favoring access speed while admitting the possibility of re-exploring previously seen states.

Note that consequence prediction algorithm does not prune any message processing events. For example, if a node resets, the other nodes in the system can detect this reset by trying to communicate with the node and receiving an RST signal.

Although simple, the idea of removing from the search the internal actions of nodes with previously seen states eliminates many (uninteresting) interleavings from the 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 the 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 5).

3.3 Execution Steering

CrystalBall's execution steering mode enables the system to avoid entering an erroneous state by steering its execution path away from the 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. Moreover, in the new programming model [Yabandeh et al. 2009] which has been suggested recently, the developer can explicitly define multiple handlers for a given event. At run-time, the run-time support module picks the one that satisfies the user-defined objective, such as consistency. In the present paper, however, we focus on generic runtime mechanisms that do not require the developer to change the programming model or explicitly specify corrective actions.

3.3.1 *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 could lead to an erroneous state, CrystalBall installs an *event filter*, which temporarily blocks the invocation of the state machine handler for the messages from the relevant sender.

The rationale for event filters 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 safety property. The joining node can later retry the procedure with an alternative potential parent and successfully join the tree. Similarly, if handling a message causes 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.

3.3.2 *Granularity of Filters.* There is a trade-off in the granularity of the filter that we can install to steer away from the inconsistency. By using a coarse granularity filter, we can cover more cases that can possibly lead to the same inconsistency and hence reduce the false negative rate. However, doing so might increase the false positive rate by filtering other events which are not threatening the consistency of the system. On the other hand, using a fine granularity filter would decrease the false positive rate as it affects only the events that are more likely to reach the inconsistency point. Doing so could result in a higher false negative rate since there might be other erroneous paths which the model checker could not find in time.

There has been related work that tries to determine the right granularity for the filter by further processing around the erroneous path using symbolic execution and path slicing [Costa et al. 2007]. Although these techniques can be effective in general, they are not directly applicable to execution steering, because the filter needs to be installed very quickly to catch the erroneous events in time.

The granularity we adopt in this article is at the level of a handler identifier. In particular, we filter on the handler whose execution in the model checker led to an inconsistency. In the case of handlers for network messages, we include the source address of the received message in the filter. It is worth noting that we can have multiple handlers for the same event. They differ in the guard condition (i.e., the boolean expression defined over the state variables) and the received message header that determines which handler is allowed to execute.

3.3.3 *Point of Intervention.* In general, execution steering can intervene at several points in the execution path. Given several choices with equal impact in terms of predicted errors, our policy is to steer the execution as early as possible (subject to non-disruptiveness criteria described below). 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.

We choose the earliest-point-of-intervention policy because it gives the system more choices (a longer execution) to adjust to the steering action. Moreover, we have a separate mechanism, the immediate safety check (described below), that enables a node to perform last-minute corrections to its behavior.

3.3.4 *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 finite executions *with* filtering are a subset of possible finite executions *without* filtering. Consequently, if a finite execution appeared in the presence of execution steering, it was also possible (even if less likely) for it to appear in the original system. The breaking of a TCP connection is common in a distributed system using TCP. 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. The sound filter for timer events is to avoid invoking the timer handler and to reschedule the timer firing. This is a sound filter because there is no real-time guarantees for invoking the scheduled timers on time, hence the timer execution could have been postponed during normal system operation.
(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.

3.3.5 *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.

3.3.6 *Immediate Safety Check.* CrystalBall also supports *immediate safety check*, a mechanism that avoids inconsistencies which 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 error 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 would be 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 16).

3.3.7 *Liveness Properties.* The notion of safe filtering (presented above) ensures that no new *finite* executions are introduced into the system by execution steering. It is possible, in principle, that applying an event filter would affect liveness properties of a distributed system (i.e. introduce new infinite executions). In our experience, due to a large amount of non-determinism (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; according to a negative result by Fischer, Lynch, and Paterson [Fischer et al. 1985], it is anyway impossible to have safety and liveness in an asynchronous system. (For example, the Paxos [Lamport 1998] protocol guarantees safety but not liveness.)

3.4 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 5 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 would 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 [Chandy and Lamport 1985]. 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 5 show that it works well in practice, but we expect that further enhancements are possible.

Applicability to Less Structured Systems. CrystalBall uses the Mace framework [Killian et al. 2007] and presents further evidence that higher-level models make the development of reliable distributed systems easier [Dagand et al. 2009; Killian et al. 2007]. Nevertheless, consequence prediction algorithm and the idea of execution steering are also applicable to systems specified in a less structured way. While doing so is beyond the scope of this article, recently proposed tools such as MODIST [Yang et al. 2009] could be combined with our approach to bring the benefits of execution steering to a wider range of distributed system implementations. Given the need for incomplete techniques in systems with large state spaces, we believe that consequence prediction remains a useful heuristics for these scenarios.

4. IMPLEMENTATION

Our CrystalBall prototype is built on top of Mace [Killian et al. 2007]. Mace allows distributed systems to be specified succinctly, and it outputs high-performance C++ code. We run the model checker as a separate process that communicates future inconsistencies to the runtime. Our implementation includes a checkpoint manager, which enables each service to collect and manage checkpoints to generate consistent neighborhood snapshots based on the notion of logical time. It also includes implementation of consequence prediction algorithm, with the ability to replay the previously discovered paths which led to some inconsistencies. Finally, it contains implementation of the execution steering mechanism.

4.1 Checkpoint Manager

To collect and manage snapshots, we modified the Mace compiler and the runtime. We added a `snapshot on` directive to the service description to inform the Mace compiler and the runtime that the service requires checkpointing. The presence of this directive causes the compiler to generate the necessary code. For example, it automatically inserts a checkpoint number in every service message and adds the code to invoke the checkpoint manager when that is required by the snapshot algorithm.

The checkpoint manager itself is implemented as a Mace service, and it compresses the checkpoints using the LZW algorithm. To further reduce bandwidth consumption, a node checks if the previously sent checkpoint is identical to the new one (on per-peer basis), and avoids transmitting duplicate data.

4.2 Consequence Prediction

Our starting point for the consequence prediction algorithm was the publicly available MaceMC implementation [Killian et al. 2007]. This code was not designed to work with live state. For example, the node addresses in the code are assumed to be of the form 0,1,2,3, etc. To handle this issue, we added a mapping from live IP addresses to model checker addresses. Since the model checker is executing real code in the event and the message handlers, we did not encounter any additional addressing-related issues.

Another change we made allowed the model checker to scale to hundreds of nodes and deal with partial system state. We introduced a dummy node that represents all system nodes without checkpoints in the current snapshot. All messages sent to such nodes are redirected to the dummy node. The model checker does not consider the events of this node during state exploration.

To minimize the impact on distributed service performance, we decouple the model checker from event processing path by running it as a separate process. On a multi-core machine this CPU-intensive process will likely be scheduled on a separate core. Operating systems already have techniques to balance the load of processes among the available CPU cores. Furthermore, some kernels (e.g., FreeBSD, Windows Vista) already have interfaces support for pinning down applications to CPU cores.

4.3 Immediate safety check

Our current implementation of the immediate safety check executes the handler in a copy of the state machine (using `fork()`), and avoids the transmission of the messages. In general, delaying message transmission can be done by adding an extra check in the messaging layer: if the code is running in the child process, it ignores the messages which are waiting in queue for transmission. However, because: 1) message transmission in the Mace framework is done by a separate thread and 2) by running `fork()` we only duplicate the active thread, adding the extra check would be redundant.

Our implementation must be careful about the resources that are shared between the parent (primary state machine) and the child process (copy). For example, the file descriptors shared between two processes might cause a problem for the incoming packets: they might be received either by the parent or the child process. The approach we took is to close all the file descriptors in the child process (the one doing immediate safety check) after calling `fork()`. The `fork()` happens after the system has reacted to a message and the execution of the handler does not read further messages. In principle, file descriptors can be used for arbitrary I/O, including sending messages and file system operations. In our implementation however, messaging is under Mace's control and it is straightforward to hold message transmission. In our experiments, only Bullet′ [Kostić et al. 2005] was performing file system I/O and we manually implemented buffering for its file system operations. A more flexible solution could be implemented on top of a transactional operating system [Porter et al. 2009].

The other shared resources that we need to consider are locks. After forking, the locks could be shared between parent and child process and waiting on them in the child process might cause undesirable effects in both the parent and the child process. We address this issue by adding an extra check in the Mace library files which operate on locks; if the code is running as child process, the library does not invoke locking/unlocking functions. This choice does not affect the system operation because the locks are used for synchronization between threads. In the child process, there is only one thread, which eliminates the need for using locks.

Since modern operating system implementations of `fork()` use the copy-on-write scheme, the overhead of performing the immediate safety check is relatively low (and it did not affect our applications). In case of applications with high messaging/state change rates where the performance of immediate safety check is critical, one could obtain a state checkpoint [Srinivasan et al. 2004] before running the handler and rollback to it only in case of an encountered inconsistency. Another option would be to employ operating system-level speculation [Nightingale et al. 2005].

Upon encountering an inconsistency in the copy, the runtime does not execute the handler in the primary state machine. Instead, it employs a sound event filter (Section 3.3).

4.4 Replaying Past Erroneous Paths

To check whether an inconsistency that was reported in the last run of model checker can still occur when starting from the current snapshot, we replay past erroneous paths. Strictly replaying a sequence of events and messages that form the erroneous path, on the new neighborhood snapshot might be incorrect. For example, some messages could have only been generated by the old state checkpoints and would thus be different from those the new state can generate. Our replay technique therefore replays only timer and application events, and relies on the code of the distributed service to generate any messages. We then follow the causality of the newly generated messages throughout the system.

The high fidelity replay of erroneous paths necessitates a deterministic replay of pseudo-random numbers. Recall that the model checker systematically explores all possible return values of the pseudo-random number generator. This function is called in three cases: i) in the protocol implementation by the developer to get a random value, ii) in the simulator to decide whether to simulate a fault injection in the current event, and iii) in the model checker to pick one of the enabled events to be simulated in the next round. As explained above, deterministically replaying all these values would not satisfy our requirements, because it is very likely that the exact explored path by previous run of the model checker does not work for the newly received state checkpoints. Hence, in deterministically replaying pseudo-random number generation, we only aim to cover the first two cases.

Toward this end, we instrumented the pseudo-random number generator function. During simulation of events, we record the values returned by the pseudo-random number generator; these values are then appended to the recorded information corresponding to the erroneous path. Later on, while replaying the erroneous path, we supply these values instead of calling the pseudo-random number generator.

4.5 Event Filtering for Execution Steering

Execution steering is driven by the reports received from the model checker; reports are sequences of events which could lead to inconsistencies. The CrystalBall controller then picks a subset of these events and installs the corresponding filters. In general, the nodes could collaborate to install disjoint sets of filters corresponding to each reported erroneous path. We have used a simple but effective approach for picking the event to filter on: the node which discovers the erroneous path looks for its first contribution to this path and installs the filter for that event. Recall that filters are designed for protocol-specific events, hence the events like node reset or message loss which are beyond the control of the protocol, will be ignored.

Upon checking the existence and the potential impact of a corrective action, the CrystalBall controller installs an event filter into the runtime.

4.6 Checking Safety of Event Filters

To check the safety of event filters, we modified our baseline execution steering library. When the execution steering module wants to check the safety of an event filter, it checks that if the code is running inside model checker, then it randomly selects both safe and unsafe choices. Since a call to the pseudo-random number generator causes a branch in the model checker, the model checker explores both cases: i) where handling the event is safe and its corresponding handler will be called, and ii) where handling the event is detected to be unsafe and the filter will be installed.

Although important for checking the safety of event filters, this technique has a negative impact on the model checker efficiency as the number of branches that the model checker needs to check increases. We assume that erroneous paths would show up rarely. Therefore, we need to check the effects of the filters only when an inconsistency is reported. However, there is a challenge here: the depth first search algorithm is designed based on the assumption that the paths will be re-explored deterministically. Hence pausing branch exploration and exploring it later can have unpredictable effects on the search algorithm. We address this issue by isolating the search of the branch caused by the event filter. For example, the states that are explored in this branch would not be reflected in the model checker data structures.

5. EVALUATION

Our experimental evaluation addresses the following questions:

(1) Is CrystalBall effective in finding inconsistencies in live runs?
(2) Can any of the inconsistencies 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?

5.1 Experimental Setup

We conducted our live experiments using ModelNet [Vahdat et al. 2002]. 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, respectively. 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 service 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 [Chang et al. 2002] 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.

5.2 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 [Stoica et al. 2003]). These implementation were not only manually debugged in both local- and wide-area settings, but were also model checked using MaceMC [Killian et al. 2007]. We have also used our tool to find inconsistencies in Bullet′, a file distribution system that was originally implemented in MACEDON [Rodriguez et al. 2004], and then ported to Mace. We found 13 new subtle bugs in these three systems that caused violation of safety properties.

In 7 of inconsistencies, the violations were beyond the scope of exhaustive search by the existing software model checker, typically because the errors manifested themselves at depths far beyond what can be exhaustively searched.

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

Table I. 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 I 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.[2] 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

[2]During this time, the model checker ran concurrently with a normally running 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.

complexity, we detail four out of 13 inconsistencies we found in the three services we examined.

5.2.1 *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. The first identified scenario by CrystalBall that violates this property is the scenario from Figure 2 in Section 1.2. 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 require changes in other handlers.

Scenario #2 exhibiting inconsistency. During live execution, A is initially the root of the tree and parent of B. R tries to join the tree by sending a join request to A. A accepts the request and decides that R should be the root of the tree, because it has a smaller identifier. Therefore, A joins under R. After receiving the JoinReply message, R is the root of the tree and A's parent. At this point, consequence prediction detects the following scenario. Node B resets, but its TCP RST packet to A is lost. Then, B sends a Join request to R, which is accepted by R because it has a free space in its children list. Accordingly, R sends a sibling update message to its child, A. Upon receipt of this message, A updates its sibling list by adding B to it, while the children list still includes B.

Scenario #3 exhibiting inconsistency. During live execution, R is the root of the tree and parent of A and C. Then, B joins the tree under A. At this point, consequence prediction detects the following scenario. B experiences a node reset, but its TCP RST packet to A is lost. Then, B sends a Join request to R and R forwards the request to C. After that, R experiences a node reset and resets the TCP connections with its children A and C. Upon receiving the error signal, each of them removes R from its parent pointer and promotes itself to be the root. In addition, each of A and C, sets its join timer to find the real root. Join timer of A expires and sends a Join message to C, which is accepted by C because it has a free space in its children list. Upon receipt of JoinReply message, A adds B to its sibling list, while B is in its children list as well.

Scenario #4 exhibiting inconsistency. During live execution, R is the root of the tree and the parent of A and C. At this point, consequence prediction detects the following scenario. Upon receiving the error signal, both nodes remove R from its parent pointer and promote themselves to be the root. In addition, both A and C start their join timer to find the real root. The join timer of A expires and A sends a Join message to C, which is accepted by C because it has a free space in its children list. Recall that A is still a sibling of C.

Possible corrections. In the message handler, check the children list before adding a new sibling. In the case of a conflict, we can either simply trust the new message and remove the conflicting node from the children list or query the offending node to confirm its state.

Root is Not a Child or Sibling. CrystalBall found violation of the property that the root node should not appear as a child, identifying a node 69 that considers node 9 both as its root and its child.

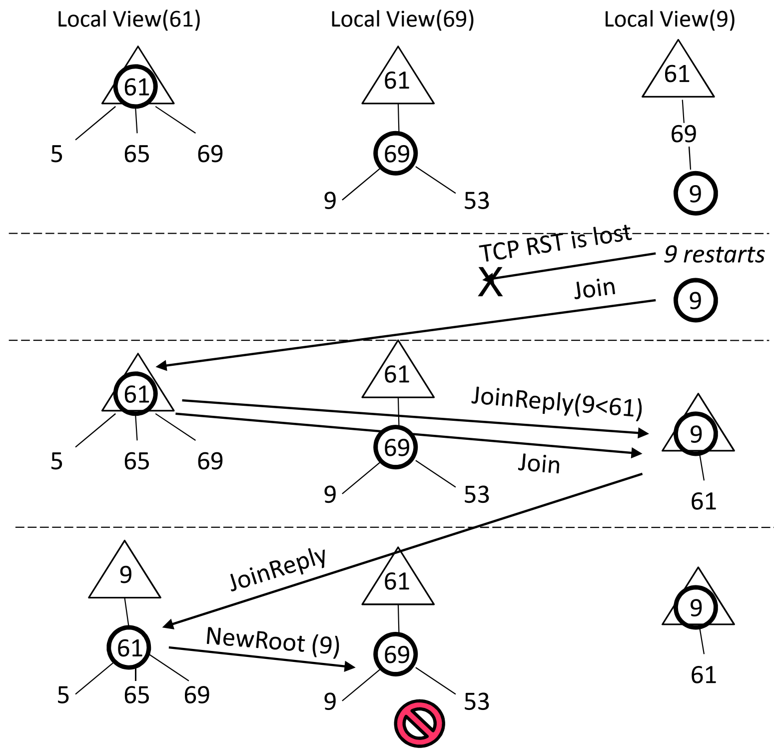

Fig. 10. An inconsistency in a run of RandTree. Root (9) appears as a child.

Scenario exhibiting inconsistency. During live execution, node 61 is initially the root of the tree and parent of nodes 5, 65, and 69; 69 is also parent of 9. At this point, consequence prediction detects the following scenario. Node 9 resets, but its TCP RST packet to its parent (69) is lost. 9 sends a Join request to 61. Based on 9's identifier, 61 considers 9 more eligible and selects it as the new root and indicates it in the JoinReply message. Besides, it also sends a Join message to the new root which would be 9. After receiving a JoinReply from 9, 61 informs its children about the new root (9) by sending NewRoot messages to them. However, 69 still thinks 9 is its child, which causes the inconsistency.

Possible correction. Check the children list whenever installing information about the new root node.

Root Has No Siblings. CrystalBall found violation of the property that the root node should contain no sibling pointers, identifying a node A that considers itself a root but at the same time has an address of another node B in its sibling list.

Scenario exhibiting inconsistency. During live execution, A is initially the root of the tree and parent of B and C. Node R sends a Join request to A. Based on R's identifier, A considers R more eligible to be the root and thus informs it of its new role. A notifies its children about the new root by sending them the NewRoot messages. At this point, consequence prediction detects the following scenario.

A experiences a node reset and resets the TCP connections with its children B and C. Upon receiving the error signal, B removes A from its parent pointer and promotes itself to be the root. However, it keeps its stale sibling list, which causes the inconsistency.

Possible correction. Clean the sibling list whenever a node relinquishes the root position in favor of another node.

Recovery Timer Should Always Run. An important safety property for RandTree is that the recovery timer should always be scheduled. This timer periodically causes each node to send Probe messages to the members of its peer list with which it does not have an open connection. It is vital for the tree's consistency to keep the nodes up-to-date about the global structure of the tree. The property that checks whether the recovery timer is always running was written by the authors of MaceMC [Killian et al. 2007] 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 occurs when another node B with a 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.

5.2.2 *Example Chord Bugs Found.* We next describe violations of consistency properties in Chord [Stoica et al. 2003], 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 [Jain et al. 2008; Rhea et al. 2005; Rowstron and Druschel 2001].

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 the list of its successors. 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 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.

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 its predecessor but has another node B as its successor. This violation happens

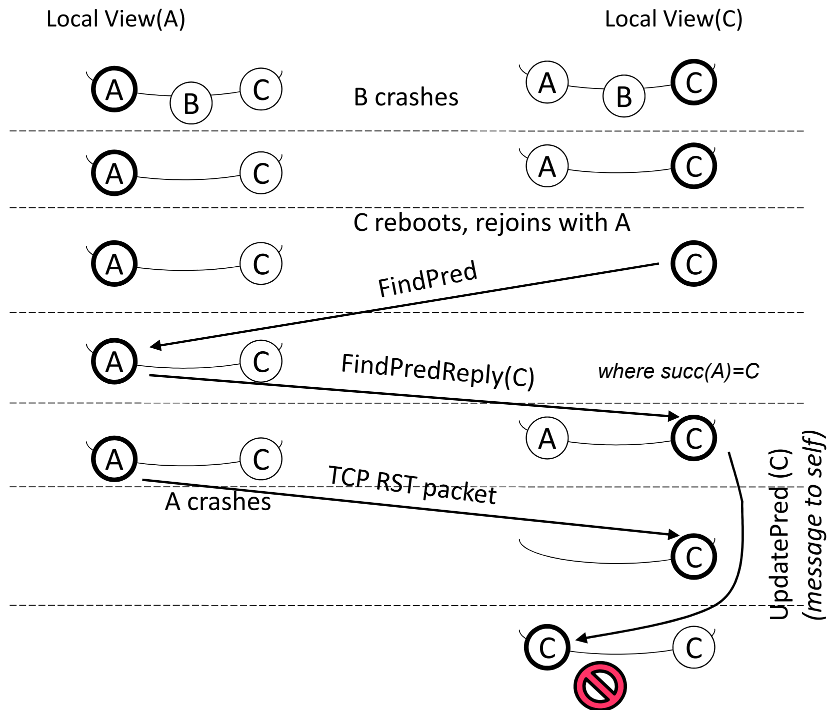

Fig. 11. An inconsistency in a run of Chord. Node C has its predecessor pointing to itself while its successor list includes other nodes.

at depths that are beyond those reachable by exhaustive search from the initial state. Figure 11 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 the 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

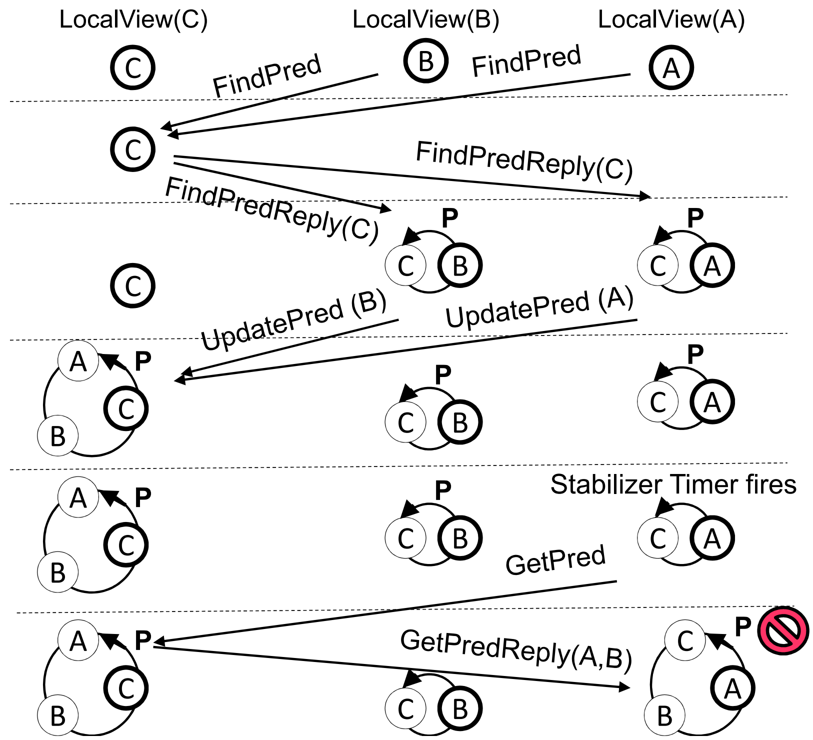

Fig. 12. An inconsistency in a run of Chord. For node A, its successor and predecessor do not obey in ordering constraint.

itself while its successor list includes other nodes.

Consequence of the Inconsistency. Services implemented on top of distributed hash tables rely on its ability to route to any system participant. An incorrect successor can therefore disrupt the connectivity of the entire system by disconnecting the Chord ring.

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.

Node Ordering Constraint. According to the Chord specification, a node's predecessor pointer contains the Chord identifier of the immediate predecessor of that node. Therefore, if a node A has a predecessor P and one of its successor is S, then the id of S should *not* be between the id of P and the id of A.

Scenario exhibiting inconsistency. CrystalBall found a safety violation where node A adds a new successor B to its successor list while its predecessor pointer is set to C and the id of B is between the id of A and C. The scenario discovered is as follows (Figure 12). The id of B is less than the id of A and the id of A is less than the id of C. During live execution, first, node C joins the ring. Then, nodes A and B both try to join with C by sending FindPred messages to it. Node C

sends FindPredReply back to A and B with exactly the same information. Upon receipt of this message, nodes A and B set their predecessor and successor to C and send UpdatePred message back to C. Finally, node C sets its predecessor to A and successor to B.

In this state, consequence prediction discovers the following actions. Stabilizer timer of A fires and this node queries C by sending it a GetPred message. Node C replies back to A with a GetPredReply message that shows the C's predecessor to be A and its successor list to contain B. Upon receiving this message, A adds B to its successor list while its predecessor pointer still points to C.

Possible correction. The bug occurs because node A adds information to its successor list but does not update its predecessor list. The bug could be fixed by updating the predecessor after updating the successor list.

Bound on data structure size. An implicit property of the data structure of a protocol is that there should be a reasonable limit on its size. Usually, this limit is internally enforced by the data structure. Chord keeps a queue called findQueue to store the received join requests while the node has not joined the ring. The node feeds requests as loop back messages to itself to respond to them. We set a property that puts a limit on the size of this queue.

Scenario exhibiting inconsistency. CrystalBall found a safety violation where the size of this queue gets unreasonably large. This causes both i) a performance problem for processing all of the messages, and ii) increase in the size of the node checkpoint from a few KB to 1MB. The bug occurs when the potential root of the tree (i.e. R) has not yet joined and other nodes of the tree stubbornly send join requests to it. When R finally joins the ring, it has a very long list of request to process.

Possible correction. The bug exists because the Mace Chord implementation does not avoid inserting duplicate messages in this queue. This bug was not observable using MaceMC because it does not model check more than several nodes (because of the state explosion problem). Our live 100-node run using CrystalBall uncovered the bug.

5.2.3 *Example Bullet′ Bug Found.* Next, we describe our experience of applying CrystalBall to the Bullet′ [Kostić et al. 2005] 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 three bugs we found were results of inconsistencies involving the variants of property 1). Since the conditions that led to the inconsistencies are similar, we describe a bug involving the following property:

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.

Consequence of the Inconsistency. Having some receivers not learn about certain blocks can cause incomplete downloads because of the missing blocks (nodes cannot request blocks that they do not know about). Even when a node can learn about a block from multiple senders, this bug can also cause performance problems because the request logic uses a rarest-random policy to decide which block to request next. Incorrect file maps can skew the request decision toward blocks that are more popular and would normally need to be retrieved later during the download.

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.

5.3 Comparison with MaceMC

To establish the baseline for model checking performance and effectiveness, we installed our safety properties in the original version of MaceMC [Killian et al. 2007]. We then ran it for the three distributed services for which we had identified safety violations. After 17 hours, exhaustive search did not identify any of the violations caught by CrystalBall. 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.

Figure 13 illustrates the performance of MaceMC when is used for exhaustive search. As depicted in figure, the exponential growth of elapsed time in terms of search depth hardly lets it search deeper than 12-13 steps. 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

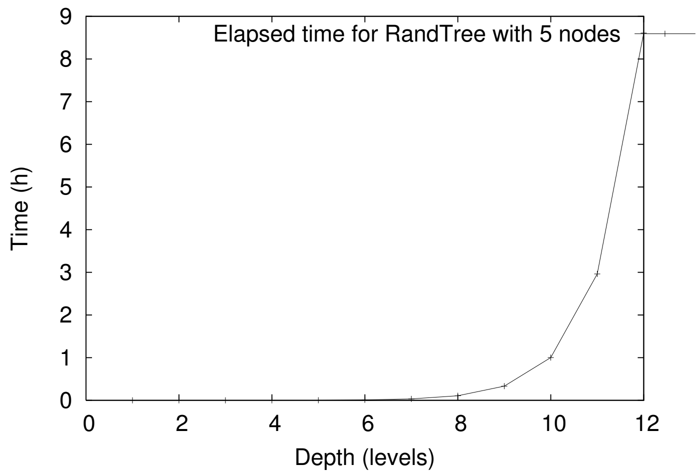

Fig. 13. MaceMC performance: the elapsed time for exhaustively searching in RandTree state space.

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.

5.4 Execution Steering Experience

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

5.4.1 *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 5.2.1). 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.

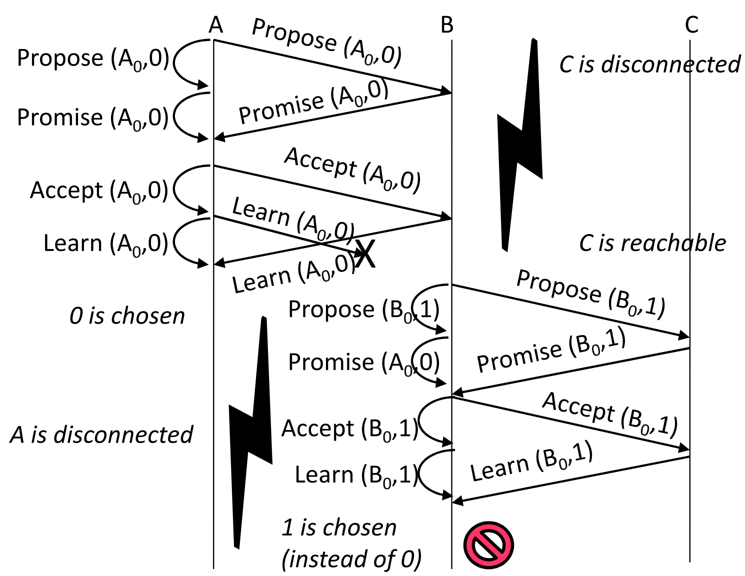

Fig. 14. Scenario that exposes a previously reported violation of a Paxos safety property (two different values are chosen in the same instance).

5.4.2 *Paxos Execution Steering.* Paxos [Lamport 1998] is a well known fault-tolerant protocol for achieving consensus in distributed systems. Recently, it has been successfully integrated in a number of deployed [Chandra et al. 2007; Liu et al. 2007] and proposed [John et al. 2008] 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 [Chandra et al. 2007; Liu et al. 2007]. 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.

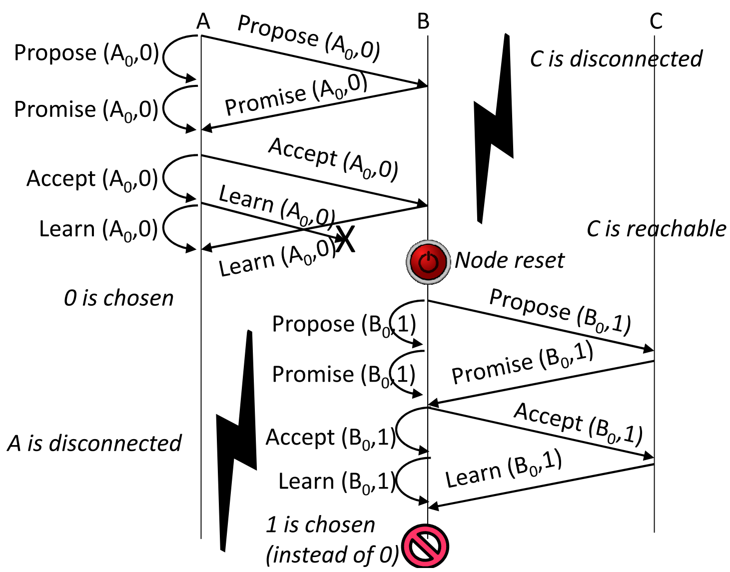

Fig. 15. Scenario that includes *bug2*, where node B resets but after reset forgets its previously promised and accepted values. This leads to violation of the main Paxos safety property (two different values are chosen in the same instance).

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. To speed up Consequence Prediction in the Paxos experiments, we annotated the unnecessary state variables to exclude them from the state hash.

The first bug we injected [Liu et al. 2007] 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 (other than those introduced by the network). As illustrated in Figure 14, 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. Node B proposes a new value (1) but its messages are received by only nodes B and C. Node B responds by a promise message containing value 0, because this value was accepted by node B in the previous round. However, node C was disconnected in previous round and responds back by the same value proposed by node B (1). Here the bug *bug1* shows up and node B upon receipt of the Promise of node C with value 1, broadcasts the Accept message with this value (1). At the end of this round, the value proposed by B (1) is chosen by B itself. In summary, this scenario shows how a buggy Paxos implementation can choose two different values in the same instance of consensus.

The second bug we injected (inspired by [Chandra et al. 2007]) involves keeping a promise made by an Acceptor, even after crashes and reboots. As pointed out in [Chandra et al. 2007], it is often difficult to implement this aspect correctly, especially under various hardware failures. Hence, we inject an error in the way an accepted value is not written to disk (we refer to it as *bug2*). To expose this bug we use a scenario similar to the one used for *bug2*, with the addition of a reset of node B. This scenario is depicted in Figure 15. The first round is similar to the first round in the scenario of *bug2*. At the end of this round, A chooses the value proposed by itself (0). Then, node B resets, but because of the *bug2* explained above, it forgets the values that were promised and accepted before the reset. In the second round, communication between A and the other nodes is broken. Node B proposes a new value (1) but its messages are only received by nodes B and C. Thus, both nodes B and C accept the default value proposed by B because: 1) node C was disconnected and did not know about the chosen value and 2) node B has forgotten its accepted value (0) after the reset (due to the injected *bug2*). At the end of this round, the value proposed by B (1) is chosen by B itself. Consequently, two different values have been chosen in the same Paxos instance.

To stress test CrystalBall's ability to avoid inconsistencies at runtime, we repeat the live scenarios 200 times in the cluster (100 times for each bug) while varying the time between rounds uniformly at random between 0 and 20 seconds. As we can see in Figure 16, 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

Fig. 16. 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.

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.

To evaluate the performance of CrystalBall's execution steering over latencies and packet drops typical of wide area networks, we run the same Paxos experiment, but this time across the wide area network emulated by ModelNet. The results are depicted in Figure 17. Overall, execution steering works well over wide area networks as well. However, the number of cases where execution steering and immediate safety check fail to detect the inconsistencies and prevent them from occurring are slightly higher. This occurs because higher latency and loss rate of wide area networks increases the chance that the model checker does not receive the consistent neighborhood snapshot in time. In execution steering, missing the updated checkpoint makes the model checker not to be able to predict the actions which are the consequences of the updated state. Similarly, the immediate safety check cannot detect the violation because of the stale snapshots.

CrystalBall causes a two-fold increase in CPU utilization. One CPU core becomes almost 100% utilized to run consequence prediction (up from zero utilization). On the core running the application itself, the CPU utilization is 8% peak burst com-

Fig. 17. In this experiment we run Paxos across an emulated wide area network using ModelNet. The experiment contains 200 runs in which the same two errors as in Figure 16 were injected. CrystalBall successfully avoided the inconsistencies in all but 6 and 3 cases, respectively.

pared to 6% without CrystalBall (in a one-second window). Given the availability of additional CPU cores and the fact that the model checking process is decoupled from the application, we consider this increase acceptable. Our approach is therefore in line with related efforts to improve reliability by leveraging increasing hardware resources (e.g. [Bayazit and Malik 2005; Arnold et al. 2008]).

5.4.3 *Chord.* In the final set of our execution steering experiments, we stress test CrystalBall's ability to avoid violations of Chord safety properties. We use a scenario that repeatedly exposes a violation described in 5.2.2. Figure 18 demonstrates that CrystalBall's execution steering is successful in avoiding this inconsistency at runtime 100% of the time. Execution steering avoids 71% of the cases, while immediate safety check avoids the rest.

5.5 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 19 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, because the effective

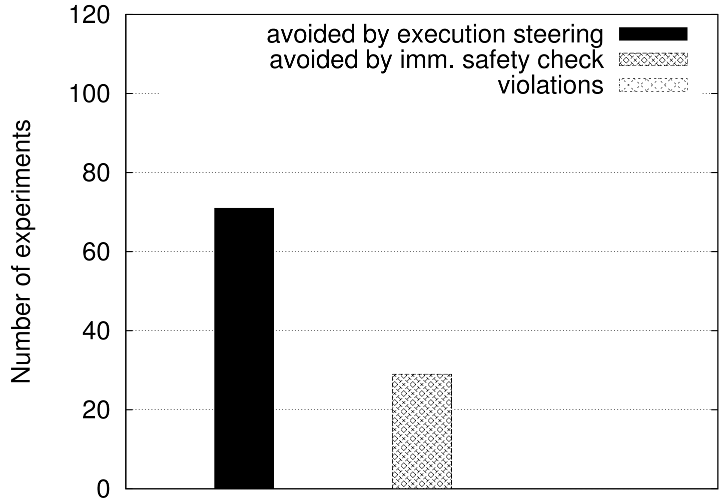

Fig. 18. In 100 runs that expose a Chord safety violation we identified, CrystalBall successfully avoided the inconsistencies in all cases.

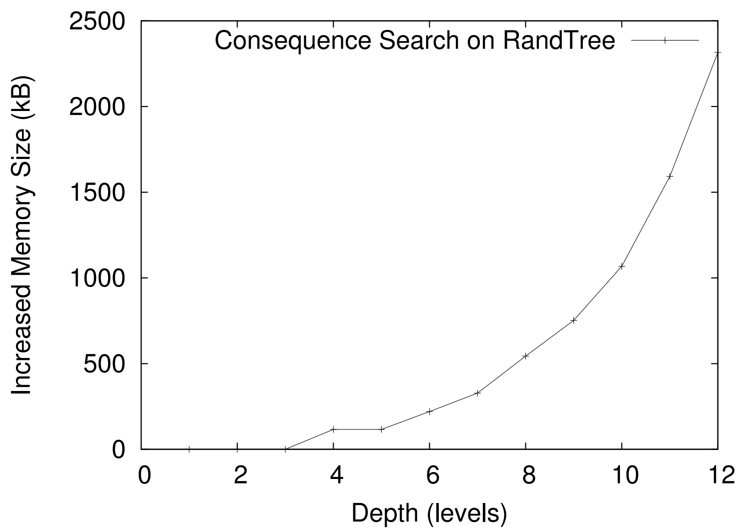

Fig. 19. The memory consumed by consequence prediction (RandTree, depths 7 to 8) fits in an L2 CPU cache.

CrystalBall's search depth is less than 7 or 8, the consumed memory by the search tree is less than 1 MB and can thus easily fit into 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.

To precisely measure the consumed memory per each visited state by consequence

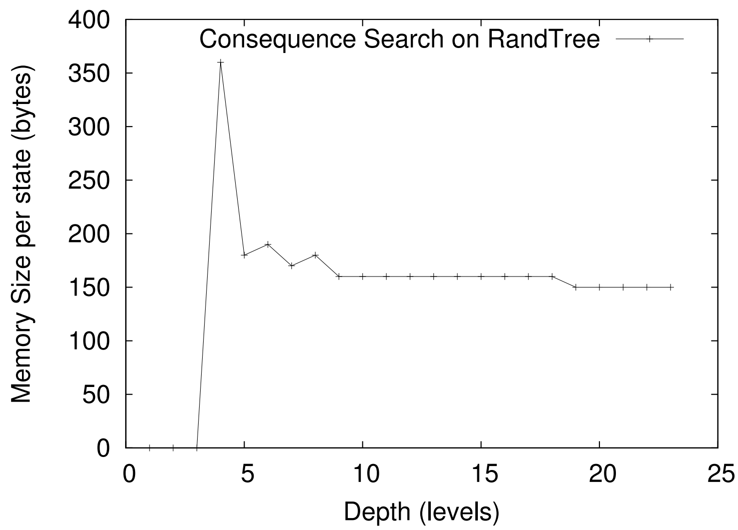

Fig. 20. The average amount of memory consumed by each explored state.

prediction algorithm, we divided the total memory used by search tree by the number of visited states. As illustrated in the Figure 20, the per-state memory gets stable at about 150 bytes as we take more states into consideration and amortize the fixed amount of space used by the model checker.

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 also 10 seconds.

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 numbers 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 neighborhood 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 21 shows that in this case, using CrystalBall had a negative impact on performance by less than 5%. Compressed

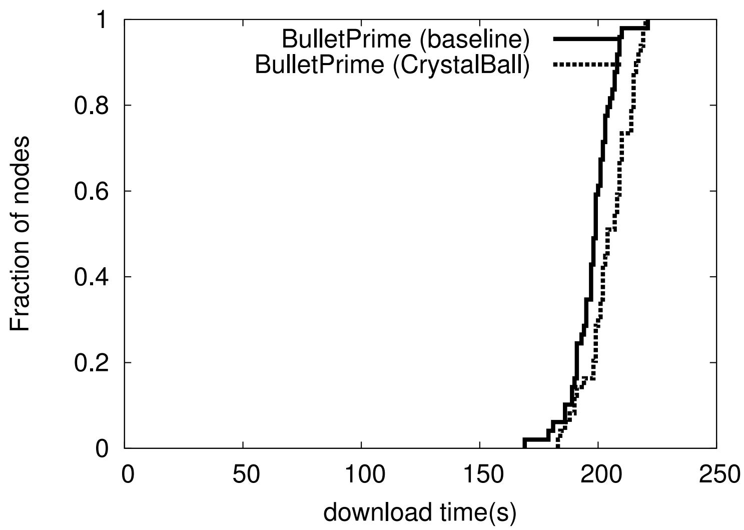

Fig. 21. CrystalBall slows down Bullet′ by less than 5% for a 20 MB file download.

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.

6. 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.

6.1 Collecting and Analyzing Logs

Several approaches (Magpie [Barham et al. 2004], X-trace [Fonseca et al. 2007], Pip [Reynolds et al. 2006]) 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.

6.2 Deterministic Replay with Predicate Checking

Friday [Geels et al. 2007] 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 [Liu et al. 2007] 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.

6.3 Online Predicate Checking

Singh *et al.* [Singh et al. 2006] 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 [Singh et al. 2006] declarative language/runtime combination. D^3S [Liu et al. 2008] and MaceODB [Dao et al. 2009] enable developers to specify global predicates which are then automatically checked in a deployed distributed system. By using binary instrumentation, D^3S 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 2) it enables execution steering to automatically prevent previously unidentified bugs from manifesting themselves in a deployed system.

6.4 Model Checking

Model checking techniques for finite state systems [Holzmann 1997; J.R. Burch et al. 1990] 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. Several systems have used sound abstraction techniques to verify sequential software as opposed to its models [Ball and Rajamani 2002; Henzinger et al. 2002; Chaki et al. 2003]. Recently, techniques based on bounding the number of context switches in multithreaded programs have been applied to single-node software systems [Musuvathi and Qadeer 2007; Musuvathi et al. 2008]. Doing so enables deterministic and systematic testing of concurrent programs, which is useful, for example in finding and reproducing so-called "heisenbugs". Bounding the number of context switches stands in contrast to consequence prediction, which follows chains of actions across any number of different nodes. Early efforts on explicit-state model checking of C and C++ implementations [Musuvathi et al. 2002; Musuvathi and Engler 2004; Yang et al. 2006] have primarily concentrated on a single-node view of the system. A continuation of this work, eXplode [Yang et al. 2006], makes it easy to model-check complex, layered storage systems, in some cases involving a client and a server.

MODIST [Yang et al. 2009] and MaceMC [Killian et al. 2007] represent the state-of-the-art in model checking distributed system implementations. MODIST [Yang et al. 2009] 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 MaceMC [Killian et al. 2007] 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 [Ball et al. 2001] 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 prediction 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) [Godefroid and Wolper 1994; Flanagan and Godefroid 2005]. Our initial investigation suggests that a POR algorithm takes considerably longer than the consequence prediction algorithm.

6.5 Runtime Mechanisms

In the context of operating systems, researchers have proposed mechanisms that safely re-execute code in a changed environment to avoid errors [Qin et al. 2007]. 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 [Dunagan et al. 2004] 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 [Costa et al. 2005] and Bouncer [Costa et al. 2007], end hosts cooperate to detect and inform each other about worms that exploit even previously unknown security holes. These systems deploy detectors that use a combination of symbolic execution and path slicing to detect infection attempts. Upon detecting an intrusion, the detector generates a Self-Certifying Alert (SCA) and broadcasts it quickly over an overlay in an attempt to win the propagation race against the worm that spreads via random probing. There are no false positives, since each host verifies every SCA in sandbox (virtual machine), after receiving it. After verification, 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 [Janjua and Mycroft 2006] by inserting locks in an approach that does not employ running static analysis at runtime. Another static approach to modifying program behavior is [Jobstmann et al. 2005] where the authors formalize the problem as a game. Approaches that modify state of a program at runtime include

[Demsky and Rinard 2003; Rinard et al. 2004]; these approaches enforce program invariants or memory consistency without computing consequences of changes to the state.

A recent approach [Wang et al. 2008; Wang et al. 2009] first uses offline static analysis to construct a Petri net model of the multi-threaded application. Authors then apply Discrete Control Theory to identify potential deadlocks in the model, and synthesize control logic that enables the instrumented application to avoid deadlocks at runtime. This approach is applicable to multi-threaded applications and the particular property of avoiding deadlock. In contrast, CrystalBall is designed to avoid general safety properties in distributed systems.

7. 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 Barbara Jobstmann for useful discussions, James Anderson for his help with the Mace Paxos implementation, and Charles Killian for answering questions about MaceMC.

This project is supported by the Swiss National Science Foundation (grant FNS 200021-125140). Nikola Knežević was funded in part by a grant from the Hasler foundation (grant 2103).

REFERENCES

Arnold, M., Vechev, M., and Yahav, E. 2008. Qvm: an efficient runtime for detecting defects in deployed systems. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. ACM, New York, NY, USA, 143–162.

Ball, T., Podelski, A., and Rajamani, S. K. 2001. Boolean and Cartesian Abstraction for Model Checking C Programs. In *TACAS*.

Ball, T. and Rajamani, S. K. 2002. The SLAM project: debugging system software via static analysis. In *POPL*.

Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. 2004. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*.

Bayazit, A. A. and Malik, S. 2005. Complementary use of runtime validation and model checking. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*. IEEE Computer Society, Washington, DC, USA, 1052–1059.

Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., and Singh, A. 2003. Splitstream: High-bandwidth Content Distribution in Cooperative Environments. In *SOSP*.

Chaki, S., Clarke, E., Groce, A., Jha, S., and Veith, H. 2003. Modular verification of software components in C. *TSE 30,* 6.

Chandra, T. D., Griesemer, R., and Redstone, J. 2007. Paxos Made Live: an Engineering Perspective. In *PODC*.

Chandy, K. M. and Lamport, L. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst. 3,* 1, 63–75.

Chang, H., Govindan, R., Jamin, S., Shenker, S., and Willinger, W. 2002. Towards Capturing Representative AS-Level Internet Topologies. In *SIGMETRICS*.

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

Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M. 2007. Bouncer: Securing Software by Blocking Bad Input. In *SOSP*.

Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. 2005. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*.

Dagand, P.-E., Kostić, D., and Kuncak, V. 2009. Opis: Reliable distributed systems in OCaml. In *ACM SIGPLAN TLDI*.

Dao, D., Albrecht, J. R., Killian, C. E., and Vahdat, A. 2009. Live Debugging of Distributed Systems. In *Compiler Construction*.

Demsky, B. and Rinard, M. 2003. Automatic Detection and Repair of Errors in Data Structures. In *OOPSLA*.

Dunagan, J., Harvey, N. J. A., Jones, M. B., Kostić, D., Theimer, M., and Wolman, A. 2004. FUSE: Lightweight Guaranteed Distributed Failure Notification. In *OSDI*.

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

Flanagan, C. and Godefroid, P. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*.

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. 2007. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*.

Geels, D., Altekar, G., Maniatis, P., Roscoe, T., and Stoica, I. 2007. Friday: Global Comprehension for Distributed Replay. In *NSDI*.

Godefroid, P. and Wolper, P. 1994. A Partial Approach to Model Checking. *Inf. Comput. 110,* 2, 305–326.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. 2002. Lazy abstraction. In *POPL*.

Holzmann, G. J. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng. 23,* 5, 279–295.

Jain, N., Mahajan, P., Kit, D., Yalagandula, P., Dahlin, M., and Zhang, Y. 2008. Network Imprecision: A New Consistency Metric for Scalable Monitoring. In *OSDI*.

Janjua, M. U. and Mycroft, A. 2006. Automatic Correction to Safety Violations. In *Thread Verification (TV06)*.

Jobstmann, B., Griesmayer, A., and Bloem, R. 2005. Program repair as a game. In *CAV*. 226–238.

John, J. P., Katz-Bassett, E., Krishnamurthy, A., Anderson, T., and Venkataramani, A. 2008. Consensus Routing: The Internet as a Distributed System. In *NSDI*. San Francisco.

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

Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. 2007. Mace: Language Support for Building Distributed Systems. In *PLDI*.

Killian, C. E., Anderson, J. W., Jhala, R., and Vahdat, A. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*.

Kostić, D., Braud, R., Killian, C., Vandekieft, E., Anderson, J. W., Snoeren, A. C., and Vahdat, A. 2005. Maintaining High Bandwidth under Dynamic Network Conditions. In *USENIX ATC*.

Kostić, D., Rodriguez, A., Albrecht, J., Bhirud, A., and Vahdat, A. 2003. Using Random Subsets to Build Scalable Network Services. In *USITS*.

Lamport, L. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Com. of the ACM 21,* 7, 558–565.

Lamport, L. 1998. The part-time parliament. *ACM Trans. Comput. Syst. 16,* 2, 133–169.

Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M. F., and Zhang, Z. 2008. D^{3}S: Debugging Deployed Distributed Systems. In *NSDI*.

Liu, X., Lin, W., Pan, A., and Zhang, Z. 2007. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*.

Manivannan, D. and Singhal, M. 2002. Asynchronous Recovery Without Using Vector Timestamps. *J. Parallel Distrib. Comput. 62,* 12, 1695–1728.

Musuvathi, M. and Engler, D. R. 2004. Model Checking Large Network Protocol Implementations. In *NSDI*.

Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. 2002. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev. 36,* SI, 75–88.

Musuvathi, M. and Qadeer, S. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*. 446–455.

Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*.

Nightingale, E. B., Chen, P. M., and Flinn, J. 2005. Speculative Execution in a Distributed File System. In *SOSP*.

Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. 2009. Operating systems transactions. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, New York, NY, USA, 161–176.

Qin, F., Tucek, J., Zhou, Y., and Sundaresan, J. 2007. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. *ACM Trans. Comput. Syst. 25,* 3.

Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. 2006. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*.

Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., and Yu, H. 2005. OpenDHT: A Public DHT Service and Its Uses. In *SIGCOMM*.

Rinard, M. C., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and Beebee, W. S. 2004. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*.

Rodriguez, A., Killian, C., Bhat, S., Kostić, D., and Vahdat, A. 2004. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*.

Rowstron, A. and Druschel, P. 2001. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *SOSP*.

Schneider, F. B. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv. 22,* 4, 299–319.

Sen, K. and Agha, G. 2006. Automated systematic testing of open distributed programs. In *FASE*. 339–356.

Singh, A., Maniatis, P., Roscoe, T., and Druschel, P. 2006. Using Queries for Distributed Monitoring and Forensics. *SIGOPS Oper. Syst. Rev. 40,* 4, 389–402.

Srinivasan, S. M., K, S., Andrews, C. R., and Zhou, Y. 2004. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX ATC*.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. 2003. Chord: a Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw. 11,* 1, 17–32.

Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., and Becker, D. 2002. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*.

Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., and Mahlke, S. A. 2008. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *OSDI*.

Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., and Mahlke, S. 2009. The Theory of Deadlock Avoidance via Discrete Control. In *POPL*.

Yabandeh, M., Knežević, N., Kostić, D., and Kuncak, V. 2009. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*.

Yabandeh, M., Vasić, N., Kostić, D., and Kuncak, V. 2009. Simplifying distributed system development. In *HotOS*.

Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., and Zhou, L. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*.

Yang, J., Sar, C., and Engler, D. 2006. EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors. In *OSDI*.

Yang, J., Twohey, P., Engler, D., and Musuvathi, M. 2006. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst. 24,* 4, 393–423.

...

A. EXAMPLE RUN OF CONSEQUENCE PREDICTION ON A SMALL SERVICE

Figure 23 shows a small service where each node has a local counter and two states. In the initial state, the node issues a Query message to another node. Upon receiving a Response message, the node stops its operation. We use this example to illustrate the differences between Consequence Search and dynamic partial order reduction.

Figures 22, 24, and 25 show, respectively, the entire space of reachable states, the transitions explored by a dynamic partial order reduction algorithm, and transitions explored by Consequence Prediction. There are 16 reachable states in the system, 20 paths, and 24 edges. A dynamic partial order reduction algorithm explores 11 of these edges, and consequence search explores 14 edges. Note that partial order reduction explores first one complete path, then inserts branches to consider certain interleavings. In contrast, consequence search explores event chains initiated by internal actions, as well as sequences of these chains where the last action of one sequence changes the state in which next local action occurs.

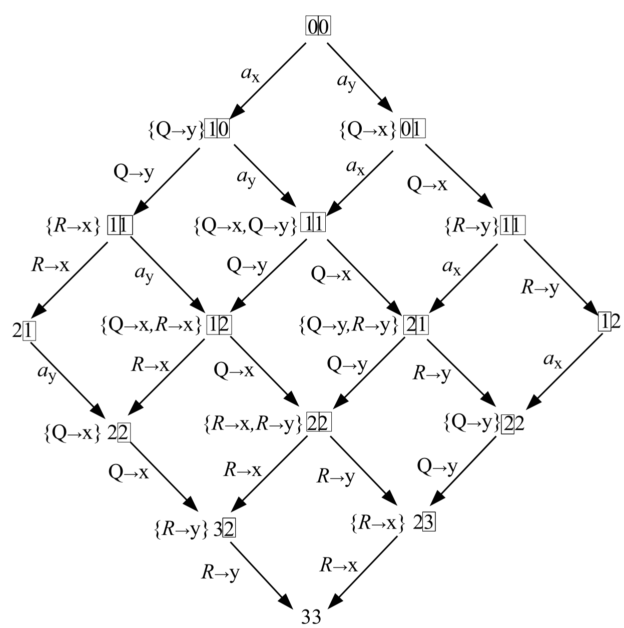

Fig. 22. Reachable states of the system in Figure 23

```
downcall (state == init) maceInit(MaceKey ip) {
  localCounter=0;
  state=started;
  peerId = ip;
}

downcall (state == started) doQuery() {
  localCounter++;
  state=querySent;
  downcall_route(peerId, Query());
}

upcall (state != init) deliver(const MaceKey& src, const MaceKey& dest,
                               const Query& msg) {
  localCounter++;
  downcall_route(src, Response());
}

upcall (state != init) deliver(const MaceKey& src, const MaceKey& dest,
                               const Response& msg) {
  localCounter++;
  state=finished;
  downcall_route(src, Response());
}
```

Fig. 23. Example Service in Mace language

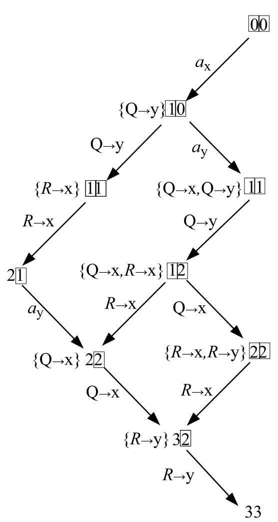

Fig. 24. Partial Order Reduction

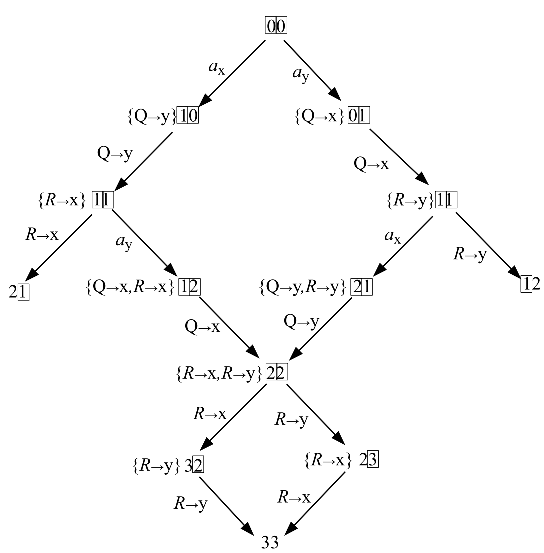

Fig. 25. Consequence Search