Lasso Detection using Partial-State Caching

Rashmi Mudduluru*, Pantazis Deligiannis*, Ankush Desai†, Akash Lal*, Shaz Qadeer*
*Microsoft Research, {t-rasmud, pdeligia, akashl, qadeer}@microsoft.com
†UC Berkeley, ankushdesai@gmail.com

***Abstract*—We study the problem of finding liveness violations in real-world asynchronous and distributed systems. Unlike a safety property, which asserts that certain bad states should never occur during execution, a liveness property states that a program should not remain in a bad state for an infinitely long period of time. Checking for liveness violations is essential to ensure that a system will always make progress in production.**

The violation of a liveness property can be demonstrated by a finite execution where the same system state repeats twice (known as lasso). However, this requires the ability to capture the state precisely, which is arguably impossible in real-world systems. For this reason, previous approaches have instead relied on demonstrating a long execution where the system remains in a bad state. However, this hampers debugging because the produced trace can be very long, making it hard to understand.

Our work aims to find liveness violations in real-world systems while still producing lassos as a bug witness. Our technique relies only on partially caching the system state, which is feasible to achieve efficiently in practice. To make up for imprecision in caching, we use retries: a potential lasso, where the same partial state repeats twice, is replayed multiple times to gain certainty that the execution is indeed stuck in a bad state.

We have implemented our technique in the P# programming language and evaluated it on real production systems and several challenging academic benchmarks.

***Index Terms*—Liveness checking, Distributed systems, Lasso detection, Testing**

I. Introduction

Concurrent programming is essential in modern software development, especially as the data and computing requirements grow beyond what is possible on a single processor core. Concurrency can be found either in the form of multi-threaded programs for multi-core processors, or as asynchronous distributed programs for multiple interconnected machines. In either case, writing *correct* concurrent programs is challenging. Subtle interactions between concurrently-running computations, such as thread interleavings or message reorderings, can lead to unexpected behaviors. Further, the non-deterministic nature of concurrency, often not controlled by the programmer, makes testing for such erroneous behaviors very difficult.

The expectations of correct behavior for a concurrent program (or any program for that matter) comes in two flavors: *safety* and *liveness* properties. Safety properties assert that a program never enters a *bad* or undesired state. The most natural form of a safety property is an *assertion*, a construct that is provided by most programming languages. Liveness properties, which are the focus of this paper, assert that the program does not *stay* in a bad state for an indefinite amount of time. (We are going to use the term *hot state* instead of *bad state* when discussing liveness properties.) Liveness properties are used to ensure that the program always makes progress. While safety specifications can be asserted and tested, there is no natural way to express liveness properties in programs, making it important to develop tools that help catch liveness violations.

As an example, consider the design of the Azure Storage vNext system [5]. As a typical cloud-storage system, vNext is a distributed program that stores user data reliably even under machine or disk failures. At a high level, it comprises of two main components: an extent node, which stores data on the local disk, and an extent manager that manages a set of extent nodes making sure that each piece of data lives on at least three extent nodes. Because machines or disks can fail at any time, it is possible that an extent node goes down. In such a case, the extent manager must detect the failure and use one (or both) of the other replicas to recreate another extent node with that data. Thus, while it is possible that the system enters a state where user data is not present on three nodes (a hot state), it must always eventually recover provided there are no more failures. A liveness violation for vNext is that the system remains in a hot state for an indefinite amount of time, even when there are no additional failures. Such kinds of bugs are very hard to find using traditional methods of testing [5]. The goal of our work is to build tools that help find liveness violations.

The model checking community has long studied this problem. The work concentrates on finding a *lasso*: a program execution that visits the same program state (say, s) twice. A lasso indicates the presence of an infinite execution because the execution from s to s can be repeated infinitely often. If, further, the lasso is hot, i.e., it is continuously in a hot state as it goes from s to s (including the state s itself) then it indicates a violation of the liveness property. A hot lasso naturally provides the user exact information on the execution segment that fails to make progress, according to the definition of what constitutes a hot state. For example, for vNext, the cycle in a potential violation would indicate the steps that the system is taking when some replica has gone down, but they fail to create a new replica.

There are several algorithms in this space that look for a hot lasso.[1] These algorithms are either exhaustive [3] or randomized [11], however they require access to the complete state of the program to know that the cycle in a lasso can be repeated indefinitely. For a distributed program, for instance,

[1]More generally, the algorithms look for violations of properties written in a temporal logic like LTL.

the state would include the individual states of all concurrent processes as well as all the messages on the network. It may even have to include the state of the operating system if the program uses system resources (e.g., disk). This is an unrealistic task in a real-world setting, especially because it has to be done at each step of the program's execution. State-of-the-art tools like SPIN [12] and ZING [1] thus work on *models* of actual systems. The creation of the model is the user's responsibility. However, programmers are often reluctant to write models or maintain them as the software evolves given the time pressures of a fast-moving software industry.

A different approach of checking liveness is to directly execute the program (with some modifications to make executions deterministic) instead of relying on a model of the program. Without capturing the program state, these approaches are unable to find a lasso. Instead, they attempt to find a sufficiently long execution with a hot suffix (i.e., all states in the suffix are hot). Instances of this approach are implemented in the MACEMC tool [13] for distributed programs and the CHESS tool [14] for multi-threaded programs. We refer to this approach as the *temperature method*. (The system is additionally tagged with a *temperature* property. The temperature goes up by a unit when the system is in a hot state and it goes to zero when it transitions to a non-hot state. When the temperature exceeds a threshold, a violation is reported.) The temperature method requires the user to set the temperature threshold. Setting this threshold too low can result in false positives and setting it too high will produce long and hard-to-understand traces because there is no lasso or cycle that tells the user where the program failed to make progress.

Our goal is to provide the user with a lasso *without* relying on lengthy executions or the ability to cache the entire program state. There are two key ingredients to our approach. First, we use a *partial-state caching* mechanism that only captures a small part of the program state. By default, we capture partial details of each concurrently executing process and the *types* of the messages currently in flight between the processes. (We do provide convenient APIs so that users can *optionally* capture additional state.) Second, we execute the program (not a model) while taking over the program scheduling and message delivery. In each execution, we use the partial-state cache to find a repeating state. Because the caching is partial, we cannot say for sure if we have found a lasso or not. We overcome this limitation by taking the potential cycle and repeatedly re-executing it. If we are able to successfully re-execute the cycle (while continuing to stay in a hot state) for a large number of iterations then we flag this as a liveness violation and show the lasso (without subsequent re-executions) to the user.

We have implemented our approach and integrated it with the P# suite [4], [16]. P# is an extension of the C# language meant for developing asynchronous systems. P# comes with tools for thorough systematic testing of programs written in the language. P# is currently in use inside Microsoft for developing production systems. Using a collection of challenging academic benchmarks as well as production systems, we report on several interesting aspects of our approach and its comparison against the temperature method:

- We present a case study (§V) to show the advantage of inspecting a lasso, as compared to looking at a long trace.
- We evaluate and compare the algorithms on a set of benchmarks (§VI). We find that our lasso detection is more robust in terms of finding liveness violations; it has higher number of true positives and fewer false positives. The partial-state caching mechanism incurs an overhead of 2X in the running time on average compared to the temperature method which does not require any caching.

The rest of the paper is organized as follows. Section II sets up the notation and Section III formally describes our liveness checking algorithm. Section IV discusses our implementation based on P#. Section V presents a case study on the utility of having a lasso. Section VI presents our experimental results. Section VII discusses related work.

II. NOTATION AND DEFINITIONS

We consider a program as consisting of concurrently executing processes that communicate with each other via message passing. We formally model a program as a transition system. In particular, an asynchronous program P is a tuple $(S, \mathit{Pid}, T, \mathit{Hot}, s_0)$ where:

- S is the set of states of P.
- *Pid* is the set of process identifiers in the system.
- T: $\mathit{Pid} \times S \rightarrow S$ is the transition function of the program. T is a partial function. If $T(m, s) = s'$, then the process m can execute a step to take the program from state s to s'. We also say in this case that (s, s') is a transition of P and m is the *scheduled* process of that transition.
- *Hot*: $S \rightarrow \mathit{bool}$ is a function that maps a state to a Boolean indicating whether the state is *hot*.
- s_0 is the initial program state.

For the sake of convenience (and without loss of generality) we assume that for all states s_1 and s_2, if $T(m_1, s_1) = s_2$ and $T(m_2, s_1) = s_2$ then $m_1 = m_2$. Thus, a transition is uniquely identified by the source and target states. Let *Scheduled* be a partial function that maps a pair of states (s_1, s_2) to the unique process identifier that takes state s_1 to s_2. Formally, if $\mathit{Scheduled}(s_1, s_2)$ is defined then $T(\mathit{Scheduled}(s_1, s_2), s_1) = s_2$. Let *Enabled* be a function that maps a state s to the set of all processes *enabled* in that state. Formally, $\mathit{Enabled}(s) = \{m \mid \exists s'.T(m, s) = s'\}$. An example depicting the transition system of a program is shown in Figure 1. For instance, $\mathit{Enabled}(s_5) = \{p_2, p_3\}$ and $\mathit{Scheduled}(s_1, s_3) = p_2$.

An *execution trace* of P is a sequence of states $s_0, s_1, \cdots, s_n$ such that $\forall i \in \{0, 1, \cdots, n-1\}, \exists m \in \mathit{Pid} : T(m, s_i) = s_{i+1}$. A *lasso* L is an execution trace $s_0, \cdots, s_n$ such that for some i with $0 \leq i < n$, $s_n = s_i$. In this case, the execution trace $s_0, \cdots, s_{i-1}$ is called the *stem* of the lasso and the sequence $s_i, \cdots, s_n$ is the *cycle* of the lasso. The presence of a lasso indicates an infinite execution because the cycle can be repeated infinitely often. L is additionally a *hot* lasso if all states in its cycles are hot. Formally, $\mathit{HotLasso}(L)$ holds if $\forall k : i \leq k < n \Rightarrow Hot(s_k)$. Similarly, L is called a *fair* lasso

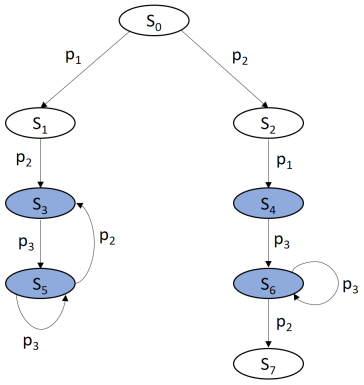

Fig. 1. A transition system. Nodes are program states and edges are transitions labelled with scheduled process name. The shaded nodes depict *hot* states.

if all processes that are enabled at some point in the cycle are scheduled in the cycle as well.[2] Formally, *FairLasso*(L) holds if $\bigcup_{k=i}^{n-1} \mathit{Enabled}(s_k) \subseteq \{\mathit{Scheduled}(s_k, s_{k+1}) | i \leq k \leq n-1\}$. A *liveness violation* of a program is a lasso that is both hot and fair.

Fairness is an important criteria while considering liveness properties. For unbounded runs, it is unlikely that a real system will starve an enabled process from executing. Programmers expect fairness and design their progress guarantees under this assumption. Thus, a hot lasso that is unfair will be a false positive from the user's perspective.

The example of Figure 1 has multiple lassos, for example $L_1 = s_0, s_1, s_3, s_5, s_3$, $L_2 = s_0, s_1, s_3, s_5, s_5$, and $L_3 = s_0, s_2, s_4, s_6, s_6$. All of these are hot lassos but only L_1 is fair. Lassos L_2 and L_3 are not fair because their cycles have p_2 enabled but it is never scheduled. For the purpose of this paper, only L_1 constitutes a liveness violation.

III. Liveness Checking Algorithms

This section outlines the two algorithms used for exploring the state space of a program in an attempt to find a liveness violation. We present the algorithm for the temperature method, inspired by previous work [13], [14]. Though this algorithm is not one of our contributions, we have implemented and presented it here mainly for the purpose of comparison. We then present our algorithm based on partial-state caching. A key hurdle in these algorithms is that, unlike a typical model-checking scenario, we cannot capture or store the entire state of a program. Instead, we only have the ability to inspect the current state, identify enabled processes and schedule an enabled process to make the program take a step. We cannot checkpoint the state, thus, we cannot identify a lasso by detecting a state seen previously in the execution.

Our algorithms will be parameterized by a *scheduler*, represented as a method GetNext that takes a state s as input and returns a process $m \in \mathit{Enabled}(s)$ that should be scheduled next. This method may have its own internal logic to decide which process to schedule next. One can use different implementations of GetNext that will, for instance, do a depth-first or a breadth-first exploration of the program by keeping track of previous decisions made. A GetNext implementation can also be randomized. It can, for example, pick and return a random element of $\mathit{Enabled}(s)$ to do a pure random exploration.

[2]This property is usually termed as *strong fairness*.

Algorithm 1 ExecProgram

Input: Initial State s_0
Input: Scheduler method GetNext
Input: Method $M \in \{\mathit{TempMethod}, \mathit{PartialCaching}\}$,
Input: Maximum steps B: Int,
Input: Temperature threshold TT: Int
Input: Replay threshold RT: Int

```
1:  s ← s_0
2:  n ← 0
3:  Temp ← 0
4:  Trace ← []
5:  while Enabled(s) ≠ ∅ ∧ n < B do
6:      m ← GetNext(s)
7:      s' ← T(m, s)
8:      Trace ← Trace + (m, Enabled(s), Hot(s), Hash(s))
9:      s ← s'
10:     n ← n + 1
11:     if M == TempMethod then
12:         Temp ← CheckTemp(s, Trace, Temp, TT)
13:     else if M == PartialCaching then
14:         s, Trace ← CheckLasso(s, Trace, RT)
15:     end if
16: end while
```

Each of the two liveness detection methods repeatedly run ExecProgram (Algorithm 1) up to a user-specified bound on the maximum number of executions to explore. Each run of ExecProgram generates a program execution, according to the GetNext scheduler, and the execution is monitored for possible liveness violation. The two methods differ in how they perform the detection.

ExecProgram takes as input the initial state of the program s_0, a scheduler GetNext, the method to use for detecting violations, the maximum length of an execution B (after which exploration is truncated), and two threshold values TT and RT that we explain later.

The main loop of ExecProgram (line 5) runs as long as there are processes available to be scheduled or until the maximum length of the execution is reached. Each iteration of the loop executes the program for one step according to the scheduler (line 7) and then calls the selected detection method (lines 12 and 14).

Algorithm 2 describes the CheckTemp method. It simply increases the temperature value (line 2) if the current state is hot and checks if the threshold has been reached. If the current state is not hot, then the temperature is reset (line 7). It is easy to see that this method reports a liveness violation on a trace if the last TT steps of the trace were in a hot state.

Algorithm 2 CHECKTEMP($s, \mathit{Trace}, \mathit{Temp}, TT$)

Input: Current state s
Input: Current trace *Trace*
Input: Current temperature *Temp*: Int
Input: Threshold TT: Int
Output: Updated temperature value

```
1: if Hot(s) then
2:    Temp ← Temp + 1
3:    if Temp = TT then
4:       REPORT-LIVENESS-BUG(Trace)
5:    end if
6: else
7:    Temp ← 0
8: end if
9: return Temp
```

Algorithm 3 CHECKLASSO(s, Trace, RT)

Input: Current state s
Input: Current trace *Trace*
Input: Threshold value RT: Int
Output: New current state
Output: Updated trace

```
1: for all i: Hash(s) = hash(Trace[i]) do
2:    C ← Trace[i..length(Trace)]
3:    if Hot(C) ∧ Fair(C) then
4:       return REPLAYCYCLE(s, C, Trace, RT)
5:    end if
6: end for
```

Algorithm 4 REPLAYCYCLE(s, C, Trace, RT)

Input: Current state s
Input: Potential cycle C
Input: Current trace *Trace*
Input: Threshold RT: Int
Output: New current state
Output: Updated trace

```
1: Trace' ← Trace
2: for j = 0 to RT × length(C) − 1 do
3:    i ← j mod length(C)
4:    m ← scheduled(C[i])
5:    if m ∉ Enabled(s) then
6:       return (s, Trace)
7:    end if
8:    s' ← T(m, s)
9:    Trace ← Trace + (m, Enabled(s), Hot(s), Hash(s))
10:   s ← s'
11:   if Enabled(s) ≠ enabled(C[i + 1 mod length(C)]) ∨
      ¬Hot(s) then
12:      return (s, Trace)
13:   end if
14: end for
15: REPORT-LIVENESS-BUG(Trace')
```

The *Trace* variable captures a summary of the current execution. It is a list of *trace events*. For a program transition $T(m, s_1) = s_2$, we record the trace event $(m, \mathit{Enabled}(s_1), \mathit{Hot}(s_1), \mathit{Hash}(s_1))$ capturing the process m that was scheduled and information about the source state s_1: the set of enabled machines in the state, if the state is hot or not, and a *hash* of the state. The function *Hash* computes a fingerprint of a state by hashing partial information gleaned from the program state. The next section details the information that we hash by default in our implementation, but users also have access to convenient APIs for hashing additional program state that is relevant to their own program. For the purpose of our algorithm, we only assume that *Hash* is indeed a function, i.e., identical states must be mapped to the same value. But the more information that is hashed about a state, the less likely it becomes that two different states are mapped to the same value.

The temperature method uses *Trace* for reporting a violation. A user can use the list of scheduled processes to replay the execution. Our implementation of the temperature method optimizes *Trace* by only keeping the process names in the trace events. The *PartialCaching* method, however, makes full use of trace events.

The PartialCaching Algorithm. For a trace event e, let *scheduled*(e) be its first element, *enabled*(e) be its second element, *hot*(e) be its third element and *hash*(e) be its last element. For a trace t (a list of trace events), let $t[i]$ be its i^{th} trace event. Let *length*(t) be the length of the trace. Let $t[i..j]$ be a sub-trace consisting of trace events $t[i], \cdots, t[j-1]$. We say that a trace t is *hot* (*Hot*(t)) if for each trace event $e \in t$, *hot*(e) is *true*. We say that a trace t is *fair* (*Fair*(t)) if $\bigcup_{e \in t} \mathit{enabled}(e) \subseteq \{\mathit{scheduled}(e) \mid e \in t\}$.

The CHECKLASSO method (algorithm 3) works as follows. For each new state s in the execution, it checks if *Hash*(s) has been seen earlier in the trace (line 1). A hit in the trace corresponds to a potential cycle, however, we cannot be sure because the hashing was only partial. It first checks if the potential cycle is hot and fair (line 3). If not, then it considers some other cycle. If it finds a hot and fair (potential) cycle, then to make sure, the method tries to *replay* the cycle.

The method REPLAYCYCLE(s, C, Trace, RT) (algorithm 4) takes over the scheduling of the execution and instead of calling GETNEXT, it uses C to make scheduling decisions. It attempts to replay C for RT number of times. If successful, the input *Trace* is reported as a liveness violation, with C marked as the hot and fair cycle of the lasso. The method proceeds as follows. Line 2 is the replay loop (for RT number of times). At line 4, the process to schedule is chosen from C. If m is not currently enabled (line 5), then the replay fails. Otherwise a step is executed by scheduling m (line 8). Next, line 11 checks if the new state matches the corresponding step ($i + 1$ **mod** *length*(C)) of C. If not then the replay fails, otherwise it keeps going.

Remarks. First, REPLAYCYCLE does not check that the state hash matches with C during replay. We are only interested in replaying the scheduling decisions in C while making

sure that the same set of processes are enabled (to ensure fairness). *Second*, when replay fails, we simply continue the program execution (in algorithm 1) from where the replay failed. This is because we cannot checkpoint state to rollback the execution from where the replay had started. We can potentially re-execute the program from the beginning to simulate rollback, but it adds extra cost to the algorithm. *Third*, in CHECKLASSO there may be many potential cycles on line 1. In our implementation, we go through these in a random order. Only the first hot and fair (potential) cycle is replayed and not the rest (line 4 executes a **return**) because the trace gets extended by the time replay fails.

A comparison of the two methods. Note that the temperature method does not have a fairness check. This is not possible because it does not produce a cycle that can be checked. To avoid false positives, previous work has relied instead on the scheduler to generate executions without starvation. For instance, MACEMC uses a randomized scheduler that picks a process randomly from the set of enabled processes; this makes it probabilistically unlikely that an enabled process will be starved in a long execution. For example, in the transition system of Figure 1, it is unlikely that a random scheduler will generate the execution $s_0, s_1, s_3, s_5, s_5, s_5, \cdots$. The randomness will ensure that p_2 is scheduled in state s_5 at some point; and likely the execution will have some alternation between states s_3 and s_5. Thus, in our experiments we limit the temperature method to use the random scheduler, whereas our partial-state caching method can use any scheduler. Random scheduling helps guard against unfairness, but it can also reduce bug-finding capabilities as illustrated by the following example.

A Dining Philosophers example. Consider a program with multiple processes, playing the role of a philosopher or a fork, arranged in a ring with alternating philosophers and forks. Each philosopher tries to acquire the fork on her left followed by the fork on her right. If she succeeds in getting both forks, she (eats and) releases both forks and quits. If she does not succeed in getting both the forks, she releases any fork with her and tries over again. This program has an infinite fair execution where each philosopher first gets the fork on their left, then they release them all realizing that the fork on the right is unavailable and so on.

The temperature method, with a randomized scheduler, is unable to detect the liveness violation: the ability to generate a particular trace decays roughly exponentially with the length of the trace, thus the method is unable to generate long traces. However, our partial caching method is able to find the violation, even while using a randomized scheduler. The reason is that it only needs to find the first iteration of a cycle after which replay will take over the scheduling. Even chances of hitting the first iteration decays exponentially with the number of philosophers. For example, for 2 to 5 philosophers, our partial-state caching method reports a (correct) liveness violation in $17.3\%, 4\%, 0.4\%, 0.03\%$ of the executions, respectively. The temperature method is not able to find a violation even for two philosophers (we used $TT = 50$).

IV. IMPLEMENTATION

We have implemented our techniques in the P# language [4], [16]. P# is designed for writing asynchronous programs. A P# program is a collection of *state machines* that run concurrently and communicate with each other by passing messages. A P# state machine (or machine for short) has an *input queue* that stores received messages and it can have an arbitrary number of fields of any C# type, just like a regular C# object. A machine can have multiple *states* in the sense of a finite-state-machine. (To avoid ambiguity with the multiple uses of the word "state", we will refer to this as a *MachineState*.) The messages are handled in a FIFO order. The user defines, separately for each MachineState, how the machine will handle a message of a particular type. It can execute a handler or transition to another MachineState. A handler can execute arbitrary (but sequential) C# code that may create more machines, send messages to other machines, block until it receives a specific message, or update the internal fields of the machine.

A P# program can run inside a single process (using a thread pool) or be deployed on a cluster of interconnected machines. P# is being used internally inside Microsoft to develop production services for Azure.

A liveness property in a P# program is specified with the help of a *monitor*. A *monitor* is a state machine that can receive but not send messages and whose *MachineStates* are optionally annotated as *hot*.[3] A monitor essentially observes the execution of the program. A liveness violation occurs if the program has a monitor in a hot state for an indefinite amount of time (for fair executions).

The P# runtime has a *bug-finding mode* that serializes the program execution on a single thread and systematically explores different interleavings of the program. P# has several *scheduling strategies* that can be used for exploration [8]. P# recommends a *portfolio* mode where testing is done in parallel, with each parallel instance using a different scheduler.

Our formalism in Section II assumed that the transition system of the program is deterministic except for the choice of which process to schedule next, i.e., given a program state s and an enabled process m, the state resulting from the execution of m was fixed ($T(m, s)$). A P# program, however, can have other sources of non-determinism, such as generating non-deterministic values. Our implementation is able to handle this non-determinism in data as well by generating these values randomly and capturing the generated value in the trace to allow for replay.

By default, we compute the fingerprint of a program state by hashing together the fingerprints of each machine. For a machine, we only look at the information that is directly visible to the P# runtime: this includes the name of MachineState that the machine is in currently and the sequence of message types in its inbox. We do not take into account the internal fields of the machine or payloads of the messages, each of

[3] P# also has the notion of *cold* and *warm* states but we do not discuss this feature in this paper.

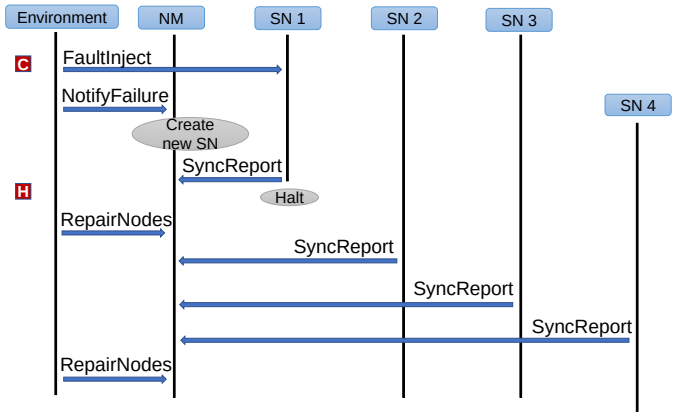

Fig. 2. ReplicatingStorage liveness bug

which can be of an arbitrary C# type, thus, hard to capture efficiently and automatically. P# offers an API by which a user can provide a more precise hash of a machine or of a message.

V. Case Study

This section compares the violations reported by the temperature method and our partial-state caching method on one benchmark. We find that a lasso expresses the liveness violation very naturally, and offers information that would otherwise be hard to deduce from a long trace.

It is important to note that even the MaceMc work [13] acknowledged that a user must be given more information than just a trace for identifying a liveness bug. MaceMc finds and displays a *critical transition* of the trace: a step of the program execution after which the program is doomed to violate the liveness property. In practice, this transition is one after which several random explorations fail to reach a non-hot state. A critical transition need not always exist (an example is the dining philosophers program), but it was present in the benchmark that follows.

The ReplicatingStorage benchmark is a simplified version of Azure Storage vNext (described in §I) that manifests a known real bug of the system. The P# program consists of a Node Manager (NM) that is responsible for handling the failure of Storage Nodes (SN) by creating new replicas. The SNs periodically send a **SyncReport** to the NM reporting a summary of the data that they currently store. We model the environment as a P# machine that randomly induces a failure to help us test the system. The program has a bug that is triggered when a SN sends a **SyncReport** to NM and then fails; the NM detects the failure, but just before it starts the repair, it gets the **SyncReport**. This causes the repair to not happen and the system continues without enough replicas. The fix is for NM to ignore **SyncReport** messages from nodes that it believes have failed.

Figure 2 shows the sequence of events that trigger the liveness bug. The Environment machine induces a node failure by sending a **FaultInject** message to SN-1. At this point, SN-1 simply enqueues this message. The Environment also sends a **NotifyFailure** to NM, which results in the creation of SN-4. Next, SN-1 sends a **SyncReport** to NM just before it handles the pending **FaultInject** message. Handling **FaultInject** causes the SN-1 machine to halt. When NM receives the **SyncReport** from SN-1, it updates its internal structures and (incorrectly) assumes that all replicas have the latest data. Subsequently, upon receipt of a periodic **RepairNodes** message (simulating a periodic callback), NM does not start a repair action and does not send the latest data to SN-4. It also ignores all the **SyncReport** messages that it receives from SN-4. As a result the system is stuck in a hot state from which it cannot recover.

In the scenario described above, the liveness monitor enters a *hot* state when SN-1 halts and the system never recovers after this. Therefore, the termination of SN-1 is the critical transition of this bug. However, this transition does not convey any useful information by itself: *any* liveness violation of this spec must start with a node failure.

In contrast, the states and messages in the cycle detected by our approach reflect the following information: NM repeatedly receives a **RepairNodes** message, which it handles, but NM does not send the latest data to any node and the SNs keep generating and sending **SyncReport** messages to NM. This information is more relevant to a user who knows that the newly created SN-4 needs to receive the latest data from NM, but it never does.

VI. Experiments

We experimented with a number of challenging academic benchmarks (which were authored by us), as well as production systems (for which we were not involved in their development). All benchmarks are written in P# [4] and are summarized in Table I, which shows lines of code (LoC), total number of machine types, total number of MachineStates and total number of message types[5].

The production systems include `PoolServer` and `Azure Storage vNext`. For the former, we picked two versions where the developers found interesting liveness violations. `Proposers` is a simplified version of the Paxos protocol obtained from previous work [9]. `Chord` [17] is a protocol implementing a distributed key-value store. `ReplicatingStorage` was described earlier (§V). `FailureDetector` is a failure detection protocol. `Process Scheduler`, `Leader Election`, and `Sliding Window` are P# versions of Spin benchmarks [12].

All benchmarks have a liveness bug, except for `Leader Election` and `Sliding Window`. For the non-production systems, a description of their liveness bugs can be found in Appendix A. The production systems are proprietary; their liveness bugs were found using P# for the first time. We performed all our experiments on a 64-bit Windows Server machine with 64 GB RAM and 16 logical cores.

Table II reports results from our experiments. All benchmarks were executed with maximum steps set to 500 (variable

[4]https://github.com/p-org/PSharp
[5]https://github.com/p-org/PSharpLab/tree/master/FMCAD17

TABLE I
BENCHMARK CHARACTERISTICS

Benchmark	LoC	#Machines	#States	#Messages
Proposers	176	3	3	5
Chord	762	3	7	22
ReplicatingStorage	757	7	20	41
FailureDetector	436	4	10	19
Process Scheduler	641	7	7	27
Leader Election	340	3	4	9
Sliding Window	344	3	4	8
PoolServer - v1	12160	10	54	49
PoolServer - v2	27908	18	139	94
Azure Storage vNext	22967	6	15	27

B in Algorithm 1). The temperature threshold (TT) was set to 250. This value was chosen after initial experimentation which revealed that smaller values led to many false positives. The threshold on cycle replay (RT) was set to 10. It is interesting to note that our algorithm was robust with respect to this value though it was set arbitrarily: there were no false positives among the traces that we manually inspected; for the remaining traces, we confirmed that once a cycle was replayed for 10 iterations, it could be also replayed for at least 10K iterations. If replay failed, it would almost always fail in the first iteration of the cycle. Each benchmark was tested for 10K executions, with 1K executions performed in parallel with 10 parallel instances. As mentioned in Section III, we use a random scheduler for the temperature method, whereas we ran a portfolio of schedulers (suggested as default to P# users) for the partial-state caching method.

Table II shows the total time taken by the two approaches (in seconds); the percentage of executions that reported bugs (along with false positive ratio, when present); and the average length of reported traces. For PartialCaching, we report the average length of the trace (L_T) along with the average length of the cycle (L_C). We also show the number of times replay failed for potential cycles that were fair and hot (D). For `PoolServer-v2`, we were surprised to not find any bugs; when we checked with the developers, we found that they were using a maximum step bound of 5000. The row `Poolserver-v2-5k` uses this setting.

The results show that the extra information tracked by the PartialCaching method incurs an overhead over the temperature method (3.5X maximum, 2X on average). The overhead is mostly due to cycle detection in the trace, but also because of the partial hash computation and failed replay attempts. However, PartialCaching method has no false positives and has consistently better bug-finding capabilities, except for `ReplicatingStorage`.

In the case of `Azure Storage vNext`, the temperature method reports nearly 88% of the executions to be buggy, whereas our partial-state caching approach reports just 0.02%. To investigate the stark difference in the number of bugs reported by both approaches, we placed checks to see if the known buggy code was reached in the executions. It turns out that in nearly 82% of the executions, the bug was never triggered. These were all false positives. The temperature threshold was reached prematurely and the execution did not get sufficient time to do the repair. We also note that this is a lower bound on the number of false positives because triggering the buggy code is a necessary but not a sufficient condition for the liveness violation. The actual number of false positives may be higher. Setting the temperature threshold to 450 still reports over 80% false positives.

PartialCaching is able to find a bug in `Proposers` and `Poolserver-v2-5k` that is not found otherwise. The former is similar to the dining philosophers example (see §III); it requires the *proposer* machines to continuously out-bid each other in alternation. Thus, generating long traces is probabilistically unlikely. For `Poolserver-v2-5k`, its because one of the portfolio schedulers (based on priority-based scheduling [2]) exposed the corner case, which the random scheduler is unable to find.

Results with just the random scheduler are given in Appendix B and results with improved state hashing are given in Appendix C.

Discussion and Summary. The temperature method has its advantages. It has been shown to work well in past work [13], and our experiments confirm this to some extent. It is also simple to implement and it was indeed the first method to be offered with P#. However, initial experience with the developers using P# indicated two shortcomings. First, it required an understanding of the temperature threshold; one must give the system enough time to recover from a hot state. Like in the case of `vNext`, a low value can result in false positives. Second, when a trace was reported, developers had to spend time identifying a "loop" in their logic to see why the system failed to make progress. Our work on the partial-state caching algorithm was directly inspired by these shortcomings, under the constraint that full-state caching would not be possible in a real setting.

Our method finds a short cycle in most cases, usually much shorter than the trace itself and points directly to why the execution failed to make progress. Further, the technique is more robust, with fewer false positives and higher true positives. It is able to find bugs (e.g., `Poolserver-v2-5k`) that would be missed otherwise, which is invaluable to the user. We believe these advantages justify the relative modest runtime overhead of the approach. It also has the added advantage of supporting multiple schedulers, which we knew from past reported experience with safety properties, that it will be useful in exposing interesting behaviors [8].

VII. RELATED WORK

Formal methods for checking liveness properties on programs is a widely studied area. The properties themselves are expressed in a temporal logic, most commonly in Linear Temporal Logic (LTL). These are compiled to a Buchi automaton, which is complemented and then intersected (via a cross-product construction) with the program. In the resulting system, the problem is then to find a lasso where the cycle contains an *accepting* state. The problem of limiting attention to fair traces is then just a matter of encoding fairness in LTL.

TABLE II
MAX STEPS: 500, ITERS:10000 (L_T : TRACE LENGTH; L_C: CYCLE LENGTH; D: DISCARDED CYCLES)

Benchmark	Time taken		% of Buggy Schedules		Trace length		
	Temperature	PartialCaching	Temperature	PartialCaching	Temperature	PartialCaching (L_T, L_C)	D
Proposers	5.87	10.23	0	**1.16**	-	21.9, 13	237
Chord	5.95	6.20	6.02	**6.03**	280.2	36.4, 3.2	0
ReplicatingStorage	45.80	160.76	**13.78**	11.96	367.7	295.4, 72.3	238
FailureDetector	48.17	74.34	0.03	**0.6**	254	78.1, 8	10076
Process Scheduler	36.77	117.3	0.33	**11.7**	411.5	236.1, 12.4	94671
Leader Election	6.83	7.38	0	0	-	-	60921
Sliding Window	35.95	125	0	0	-	-	0
PoolServer-v1	11.6	24.53	1.5	**2.57**	250.4	287.6, 26.6	14442
PoolServer-v2	46.63	68.85	0	0	-	-	0
PoolServer-v2-5k	28.75	64.11	0	**0.03**	-	624.2, 10.6	2
Azure Storage vNext	80.5	139.8	88.95 | 82.21 FP	0.02	309.6	235, 24	50

The classical algorithm for finding such a lasso is the *Nested Depth First Search* (NDFS) algorithm [3]. State-of-the-art implementations of this algorithm, with various improvements [7], [10], can be found in tools such as SPIN [12] and ZING [1]. However, this methodology requires the ability to cache the entire state (or a fingerprint of it). Consequently, both SPIN and ZING support their own input languages for writing models of actual systems. This is not readily possible in our setting.

The P programming language [6], [15] was co-designed with P#. It carries the same state-machine and message-passing structure as P#. However, unlike P# which is an extension of the C# language, P is its own programming language with its own data types and type system, designed in a manner that a P program can be compiled directly to ZING's input language. (A P program can interface with external C procedures for deployment in production, but the programmer is required to provide P models of any external procedure.) Thus, a P program can be analyzed using ZING. We coded some of our simpler benchmarks in P, where it was possible to capture the entire program state. However, ZING's performance was disappointing, often unable to find the bug in the program. This was because of two main reasons. First, the encoding of fairness in the translation to ZING introduced a lot of non-determinism in the model. Second, NDFS insists on a DFS order to explore the state space. Our benchmarks have infinite state spaces (or very large, even when the execution depth is constrained [7]). We found NDFS often getting lost in exploring sub-regions of the state space that did not have bugs and not being able to exhaustively cover the sub-region before it timed out.

Previous work on *stateless* techniques, which do not capture the program state, is usually restricted to safety properties. They advocate encoding liveness properties as safety assertions that check for progress explicitly [18]. Our approach instead automatically reports a lasso as a proof of "no progress". MACEMC [13] and CHESS [14] are stateless approaches that directly look for liveness violations. They have already been covered in the paper.

REFERENCES

[1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *Conference on Concurrency Theory (CONCUR)*, pages 1–15, 2004.

[2] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010.

[3] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Computer Aided Verification (CAV)*, pages 233–242, 1990.

[4] P. Deligiannis, A. F. Donaldson, J. Ketema, A. Lal, and P. Thomson. Asynchronous programming, analysis and testing with state machines. In *Programming Language Design and Implementation (PLDI)*, pages 154–164, 2015.

[5] P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal, R. Mudduluru, S. Qadeer, and W. Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In *File and Storage Technologies (FAST)*, pages 249–262, 2016.

[6] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *Programming Language Design and Implementation (PLDI)*, pages 321–332, 2013.

[7] A. Desai, S. Qadeer, S. Rajamani, and S. Seshia. Iterative cycle detection via delaying explorers. Technical report, March 2015.

[8] A. Desai, S. Qadeer, and S. A. Seshia. Systematic testing of asynchronous reactive systems. In *Foundations of Software Engineering (FSE)*, pages 73–83, 2015.

[9] M. Emmi and A. Lal. Finding non-terminating executions in distributed asynchronous programs. In *Static Analysis Symposium (SAS)*, pages 439–455, 2012.

[10] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Protocol Specification, Testing and Verification (PSTV)*, pages 109–124, 1993.

[11] R. Grosu and S. A. Smolka. Monte carlo model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 271–286, 2005.

[12] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[13] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Networked Systems Design and Implementation*, 2007.

[14] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Programming Language Design and Implementation (PLDI)*, pages 362–371, 2008.

[15] P: Safe asynchronous event-driven programming. https://github.com/p-org/P.

[16] P#: Safe asynchronous event-driven .NET programming. https://github.com/p-org/PSharp.

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160. ACM, 2001.

[18] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX*

TABLE III
PARTIAL STATE CACHING WITH RANDOM SCHEDULER; MAX STEPS: 500, ITERS:10000 (L_T : TRACE LENGTH; L_C: CYCLE LENGTH; D: DISCARDED CYCLES)

Benchmarks	Time (s)	%Buggy	(L_T, L_L)	D
Proposers	7.04	0.93	19.4, 11	222
Chord	8	6.04	35.63, 3	0
ReplicatingStorage	184.81	17.37	349.18 , 66.9	455
FailureDetector	70.22	0.49	36.16, 7	13324
Process Scheduler	93.53	3.97	258.72, 11.43	69464
Leader Election	8.86	0	-	83184
Sliding Window	110.77	0	-	0
PoolServer-v1	30.92	1.32	225.3 , 16.73	9961
PoolServer-v2	73	0	-	0
PoolServer-v2-5k	67.53	0	-	0
Azure Storage vNext	61.92	0	-	35

TABLE IV
PARTIAL STATE CACHING, PORTFOLIO SCHEDULER; MAX STEPS: 500, ITERS:10000 (D: DISCARDED CYCLES)

Benchmarks	Time (s)	%Buggy	D
Proposers	3.12	0	0
FailureDetector	86.34	1.89	1535
Process Scheduler	65.5	0.44	0
Leader Election	9.63	0	0

Symposium on Networked Systems Design and Implementation, pages 213–228, 2009.

APPENDIX

A. Benchmark bug descriptions

1) Proposers: This program is a simple version of the Paxos consensus protocol. It consists to two proposer machines and one acceptor machine. A proposer machine proposes an integer value and waits for a response from the acceptor. The acceptor machine receives these integer proposals and accepts the one with a greater value. Once a proposer receives the first *accept*, it again sends the accepted value to the acceptor and waits for a final *accept*. In the meanwhile, if the acceptor sees a greater value, it declines the value from this proposer. When a proposer receives a decline, it increments its proposal value and repeats the process again. This program has an infinite execution when both the proposers keep out-bidding each other in alternation. In this scenario, the probability of hitting the bug decreases with the trace length and therefore the temperature method cannot find it.

2) Chord: Chord [17] is a protocol for implementing a distributed key-value store. It forms a virtual ring among the nodes in the system. Each node is associated with a node ID. Given an input key, the protocol uses consistent hashing to map the key to a node ID. This is done as follows: if the Chord ring consists of a node whose ID is the same as the key, this key is assigned to that node. Otherwise, the key is assigned to the next node whose ID is greater than the key in the ring. We wrote this protocol in P# with the liveness spec that each request for a key lookup must eventually get a response. But the implementation has a bug: when we query for a key that is not present in any of the nodes, each of the nodes keep passing the key to its successor in an infinite loop.

According to the protocol, both the key IDs and node IDs must be in the range of 0 - 2^m where m is a configuration parameter of the protocol (it is the number of identifier bits). In our test, we accidentally forgot to enforce the constraint that the key must be in this range (or hashed to this range). This resulted in a bug in the implementation.

3) FailureDetector: This program consists of a set of nodes and a *failure-detector* machine. The failure detector machine pings the other nodes and looks out for failures by checking for responses. It then notifies a client machine of these failures. Because of a race condition in the program (in the order in which messages are delivered), a node failure can occur before the client is registered, causing a liveness violation.

4) Process Scheduler: This benchmarks consists of a client and a server machine communicating asynchronously. The server generates resources and the client consumes them. When a resource is unavailable, the client goes to sleep. When the server generates a new resource, it wakes up the client if it is blocked. Due to a race in the program, the client gets blocked indefinitely even after a resource is made available.

B. Partial-state caching without portfolio scheduling

Table III presents results for the partial-state caching approach with just random scheduler. The characteristics are quite similar for the academic benchmarks (in fact, better performance for `ReplicatingStorage`), however, bugs are missed for `PoolServer-v2-5k` and `vNext`.

C. Partial-state caching with improved hash

We added additional user-defined state hashing to some of the academic benchmarks. Our goal was not to make the caching perfect, but rather to improve over the default. The results are reported in Table IV. The improved hashing makes it more likely that potential cycles are indeed true cycles. This is confirmed in the results as the number of discarded cycles D is much smaller as compared to Table II. In fact, D is zero in many cases indicating that the hashing was perfect. However, the performance in terms of bug-finding capabilities is mixed: better for `FailureDetector` and worse for `Proposers` and `ProcessScheduler`. It is interesting that the liveness bug in `Proposers` is not found at all. It turns out that this is expected. The benchmark has a monotonically-increasing counter. Including it as part of the state hash implies that the same state will never repeat. While this program has a lasso of repeating actions, it does not have a lasso with repeating states.