Learning-Based Controlled Concurrency Testing

SUVAM MUKHERJEE, Microsoft Research, India
PANTAZIS DELIGIANNIS, Microsoft Research, USA
ARPITA BISWAS, Indian Institute of Science, India
AKASH LAL, Microsoft Research, India

Concurrency bugs are notoriously hard to detect and reproduce. Controlled concurrency testing (CCT) techniques aim to offer a solution, where a *scheduler* explores the space of possible interleavings of a concurrent program looking for bugs. Since the set of possible interleavings is typically very large, these schedulers employ heuristics that prioritize the search to "interesting" subspaces. However, current heuristics are typically tuned to specific bug patterns, which limits their effectiveness in practice.

In this paper, we present QL, a learning-based CCT framework where the likelihood of an action being selected by the scheduler is influenced by earlier explorations. We leverage the classical Q-learning algorithm to explore the space of possible interleavings, allowing the exploration to adapt to the program under test, unlike previous techniques. We have implemented and evaluated QL on a set of microbenchmarks, complex protocols, as well as production cloud services. In our experiments, we found QL to consistently outperform the state-of-the-art in CCT.

CCS Concepts: • **Computing methodologies** → **Reinforcement learning**; • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Concurrency, Systematic Testing, Reinforcement Learning

ACM Reference Format:
Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. 2020. Learning-Based Controlled Concurrency Testing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 230 (November 2020), 31 pages. https://doi.org/10.1145/3428298

1 INTRODUCTION

Testing concurrent programs for defects can be challenging. The difficulty stems from a combination of potentially exponentially large set of program behaviors due to interleavings between concurrent workers, and the dependence of a bug on a specific, and rare, ordering of actions between the workers.[1] The term *Heisenbug* has often been used to refer to concurrency bugs because they can be hard to find, diagnose and fix [Gray 1986; Musuvathi et al. 2008]. Unfortunately, traditional techniques, such as stress testing, are unable to uncover many concurrency bugs. Such techniques offer little control over orderings among workers, thus fail to exercise a sufficient number of program behaviors, and are a poor certification for the correctness or reliability of a concurrent program. Consequently, concurrent programs often contain insidious defects that remain latent

[1]Concurrency can come in many forms: between tasks, threads, processes, actors, and so on. In this paper, we will use the generic term *worker* to refer to the unit of concurrency of a program.

Authors' addresses: Suvam Mukherjee, Microsoft Research, Bangalore, India, suvamm@outlook.com; Pantazis Deligiannis, Microsoft Research, Redmond, USA, pdeligia@microsoft.com; Arpita Biswas, Indian Institute of Science, Bangalore, India, arpitab@iisc.ac.in; Akash Lal, Microsoft Research, Bangalore, India, akashl@microsoft.com.

2475-1421/2020/11-ART230
https://doi.org/10.1145/3428298

until put into production, leading to actual loss of business and customer trust [Amazon 2012; Tassey 2002; Treynor 2014].

Previous work on testing of concurrent programs can broadly be classified into *stateful* and *stateless* techniques, depending on whether they rely on observing the state of the program or not. Stateful techniques require a precise representation of a program's current state during execution. Examples of successful tools in this space include Spin [Holzmann 2011] and Zing [Andrews et al. 2004] that do enumerative exploration, as well as tools such as NuSmv [Cimatti et al. 2000; Clarke et al. 1996] that do symbolic exploration. These techniques, however, are generally applied to a *model* of the program. Each of these tools have their own input representation for programs and a user must encode their scenario in this representation. This limits the application of stateful techniques for testing real-world code; doing so would require taking precise snapshots of an executing program, which is hard to do efficiently and reliably.

Stateless techniques overcome the requirement of recording program state, and thus, are a better starting point for testing real-world code. Such techniques require taking over the scheduling in a program. By controlling all scheduling decisions of which worker to run next, they can reliably explore the program's state space. The exploration can be *systematic*, i.e., exhaustive in the limit, or *randomized*. Since the number of executions of a concurrent program (also called *interleavings*) is usually very large, schedulers leverage heuristics that prioritize searching "interesting" subspaces of interleavings where concurrency bugs[2] are likely to occur. For example, many bugs can be caught with executions that only have a few *context switches* [Musuvathi and Qadeer 2007] or a few *ordering constraints* [Burckhardt et al. 2010] or a few number of *delays* [Emmi et al. 2011], and so on. These heuristics have been effective at finding several classes of concurrency bugs in microbenchmarks and real-world scenarios [Deligiannis et al. 2016; Thomson et al. 2016]. *Controlled concurrency testing* (CCT) refers to the general setup of applying a scheduler to (real-world) programs for the purpose of finding concurrency bugs.

Our goal is to improve upon the state-of-the-art for CCT. Our technique leverages lessons from the area of *reinforcement learning* (RL) [Sutton and Barto 1998; Szepesvári 2010], which also has been concerned with the problem of efficient state-space exploration. The general RL scenario comprises an *agent* interacting with its *environment*. Initially, the agent has no knowledge about the environment. At each step, the agent can only partially observe the state of the environment, and invoke an *action* based on some policy. As a result of this action, the environment transitions to a new state, and provides a *reward* signal to the agent. The objective of the agent is to select a sequence of actions that maximizes the expected reward. RL techniques have been applied to achieve spectacular successes in domains such as robotics [Kober et al. 2013; Levine et al. 2016], game-playing (Go [Silver et al. 2016], Atari [Mnih et al. 2015], Backgammon [Tesauro 1991]), autonomous driving [Lange et al. 2012; Li et al. 2019; O'Kelly et al. 2018], business management [Cai et al. 2018; Shi et al. 2019], transportation [Khadilkar 2018; Wei et al. 2018], chemistry [Neftci and Averbeck 2019; Zhou et al. 2017] and many more.

We map the problem of CCT to the general RL scenario. In essence, the RL agent is the CCT scheduler and the RL environment is the program: the scheduler (agent) decides the action in the form of which worker to execute next, and the program (environment) executes the action by running the worker for one step, and then passing control back to the scheduler to choose the next action. RL techniques are robust to partial state observations, which implies that we only need to partially capture the state of an executing program, which is readily possible. In that sense, our technique is neither stateless nor stateful, but rather somewhere in between. The main contribution of this paper is a CCT scheduler based on the classical Q-learning algorithm [Peng and Williams

[2] A concurrency bug refers to an interleaving that causes a user-specified assertion to fail.

1996; Watkins and Dayan 1992]. To the best of our knowledge, this scheduler is the first attempt at applying learning-based techniques to the problem of CCT.

The use of RL is fundamentally very different compared to the stateless exploration techniques mentioned earlier. The latter build off empirical observations to define heuristics that optimize for a particular subspace of interleavings. This can, however, have its shortcomings when the heuristics fail to apply. The heuristics optimize for a particular bug pattern, but fall-off in effectiveness as soon as the bug escapes that pattern. This renders the testing to be brittle against new classes of bugs. For instance, a bug that is triggered by a combination of two patterns will not be found by a scheduler looking for just one of those patterns.

This problem of brittleness is further compounded in scenarios where concurrency is not the only form of non-determinism in the program. Programs can have *data* non-determinism as well (we refer to concurrency as a form of *control* non-determinism). Data non-determinism is used to generate unconstrained scalar values, for example, to model user input or choices made outside the control of the application under test. In these cases, heuristics on effectively resolving the non-determinism do not exist because the resolution can be very program specific (see more in Section 2).

Lastly, existing schedulers do not *learn* from the explorations performed in previous iterations, other than perhaps not repeating the same interleaving again. Systematic schedulers continue along their fixed exploration strategy (regardless of the program) and randomized schedulers execute an iteration independent of previously explored iterations. None of them gather feedback from the program under test.

In this paper, we show how RL does not exhibit the falling-off behavior for complex bug patterns, and systematically learns from exploration done previously, even in the presence of data non-determinism.

It is worth noting that RL, in general, has two phases. The first phase is concerned with learning a strategy for the agent through exploration, and in the second phase the agent simply applies the learnt strategy to navigate the environment. With CCT, we limit our attention to the first phase because our objective is simply to explore the state space of the program, and stop as soon as a bug is discovered.

We implemented our RL-based scheduler (which we denote as `QL`) for testing of concurrent .NET applications. We evaluated `QL` on a wide range of `C#` applications, including production distributed services spanning tens of thousands of LOC, complex protocols, and multithreaded programs. Our results show that `QL` outperforms state-of-the-art CCT techniques in terms of bug-finding ability. On some benchmarks, `QL` was the only scheduler that was able to expose a particular bug. Moreover, in many cases, its frequency of triggering bugs was higher compared to other schedulers.

The main contributions of this paper are as follows:

(1) We provide a novel mapping of the problem of testing a program for concurrency bugs onto the general reinforcement learning scenario (Section 4).
(2) We provide an implementation of the RL-based scheduling strategy for testing `C#` applications (Section 5).
(3) We perform a thorough experimental evaluation of our RL-based scheduler on a wide range of applications, including large production distributed services (Section 6).

Our work is a starting point for the application of RL-based search to CCT. We anticipate that many further improvements are possible by tuning knobs such as the reward function, partial state observations, etc. For this purpose, we make our implementation and (non-proprietary) benchmarks available open-source[3].

[3] https://github.com/suvamM/psharp-ql

The rest of this paper is organized as follows. In Section 2, we provide a high-level overview of our learning-based scheduling strategy. We cover background material on controlled concurrency testing and reinforcement learning in Section 3, and then present the QL exploration strategy in Section 4. We discuss the implementation details of QL in Section 5 and discuss our experimental evaluation in Section 6. We present related work in Section 7, limitations and future work in Section 8, and conclude in Section 9.

2 OVERVIEW

This section provides an overview of the key benefits of our learning-based scheduler. We consider a message-passing model of concurrency in our examples. That is, programs can have one or more workers executing concurrently, where each worker has its own local state and communicates with other workers via messages. When the program reaches a "bad" state (e.g., an assertion is violated), it stops and raises an error. We note that the choice of message-passing is only for illustration. Our technique applies to other forms of concurrency as well; our experiments, for instance, include shared-memory multi-threaded programs.

The goal of CCT is to find an execution of a given program that raises an error. CCT typically works by serializing the execution of the program, allowing only one worker to execute at a given point. More precisely, CCT is parameterized by a *scheduler* that is called at each step during the program's execution. The scheduler must pick one *action* among the set of all enabled actions at that point to execute next. For simplicity, assume that a worker can have at most one enabled action, which implies that picking an enabled action is the same as picking an enabled worker. Section 3 presents our formal model that also considers the possibility of each worker having multiple enabled actions in order to allow for data non-determinism.

We focus this section on two particular schedulers that have been used commonly in prior literature. The first is a pure random scheduler. This scheduler, whenever it has to make a scheduling decision, picks one worker uniformly at random from the set of all enabled workers. The second scheduler is *probabilistic concurrency testing* (PCT) [Burckhardt et al. 2010]. In PCT, each worker is assigned a unique priority. The scheduler always selects the highest-priority worker, except that at a few randomly chosen points during the program's execution, it changes (decreases) the priority of the highest-priority worker. The two schedulers are formalized in Section 3. We use additional schedulers in our evaluation (Section 6).

Learning-based scheduler. Our main contribution is QL, a learning-based scheduler. As mentioned in the introduction, we consider the scheduler to be an *agent* that is interested in exploring the *environment*, i.e., the program under test. The agent can partially observe the configuration of the environment which, in our case, is the program's current state. Whenever the agent asks the environment to execute a particular action, it gets feedback in the form of a *reward*. The agent attempts to make decisions that maximizes its reward.

QL employs an adaptive learning algorithm called Q-Learning [Watkins and Dayan 1992]. For each state-action pair, QL associates a (real-valued) quality metric called *q-value*. When QL observes that the program is in state s, it picks an action a from the set of all enabled actions with probability that is governed by the q-value of (s, a). After one run of the program finishes, QL updates the q-values of each state-action pair that it observed during the run, according to the rewards that it received, and uses the updated q-values for choosing actions in the next run.

The goal of QL is to explore the program state space, covering as many diverse set of executions as it can. In other words, the scheduler should maximize coverage. For this purpose, we set the reward to always be a fixed *negative* value (-1), which has the effect of *disincentivizing* the scheduler from visiting states that it has seen before. In other words, the more times a state s has been visited

before, the higher is the likelihood of QL to stay away from s in future runs. The choice remains probabilistic and the probabilities never go to 0. This is essential because the scheduler only gets to observe the program state partially.

Adaptive learning has important ramifications for exploration. Stateful techniques require observing the entire program configuration, whereas QL only requires a partial observation, which is more practical, especially for complex production systems. Unlike stateless strategies, QL adapts its decisions based on prior program executions. Schedulers like Random and PCT are agnostic to the program: they follow the same hard-wired rules for any program.

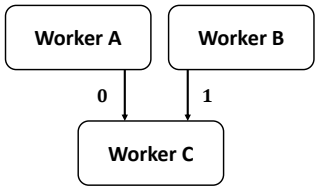

Fig. 1. Simple messaging example with 3 workers.

Table 1. B-% for Random, PCT and QL scheduling strategies for the program in Figure 1.

	B-%		
String	**Random**	**PCT**	**QL**
$\eta_1 = 0^9 1$	0.08	0.97	7.34
$\eta_2 = (01)^5$	0.08	✗	7.82
$\eta_3 = (01)^3 0^3 1$	0.08	✗	7.07

Power of state observations. Consider the program in Figure 1, with two workers A and B continually sending messages, denoted 0 and 1 respectively, to a third worker C. Worker C has a constant n-length string $\eta \in \{0,1\}^*$, as well as a counter m that is initialized to 0. If the i-th message received by C (which is 0 if sent by A or 1 if sent by B), matches the i-th character of η, then m is incremented, else it is set to -1 and is never updated again. The program reaches a bad configuration if $m = n$. Note that given any string η, there is exactly one way of scheduling between A and B so that C raises an error after n messages. We measure the effectiveness of each scheduler as the B-% value: the percentage of buggy program runs in a sufficiently large number of runs. Table 1 shows the results for Random, PCT and QL, for different choices of the string η.

The Random scheduler has similar B-% values for the various strings. It is easy to calculate this result analytically: for any string η of length n in the above example, Random has $\frac{1}{2^n}$ chance of producing that string, as it must choose between workers A and B exactly according to η. Although B-% does not change with the string, the effectiveness of finding the bug is poor because of the exponential dependence on the string length. The PCT scheduler biases search for certain types of strings. As seen in Table 1, when the string matches the PCT heuristics, its effectiveness is much better than Random, however, it has no chance otherwise.

QL is able to expose the bug for each of the strings with a much higher B-% compared to the other two schedulers. QL benefits greatly from observing the state of the program as it performs exploration. For this example, we set it up to observe just the value of counter m of worker C. QL optimizes for coverage, and because the counter is set to -1 on a wrong scheduling choice, it is forced to learn scheduling decisions that keep incrementing m, which leads to the bug.

Resilience to bug patterns. Different schedulers have their own strengths: they are more likely to find bugs of a certain pattern over others. Consider a slight extension to the Figure 1 example where we add another worker D that sends a *single* message (denoted 2) to C and then halts. In this case, Random is much more likely to find the string $\eta_4 = 2(0^9 1)$ (B-% of 0.05%) compared to the string $\eta_5 = (0^9 1)2$ (B-% of 0.003%). The reason is that Random chooses among all enabled workers with equal probability. Thus, it is more likely to schedule D earlier in the execution rather than later. PCT works much better at scheduling D late because it schedules based on priorities. If D

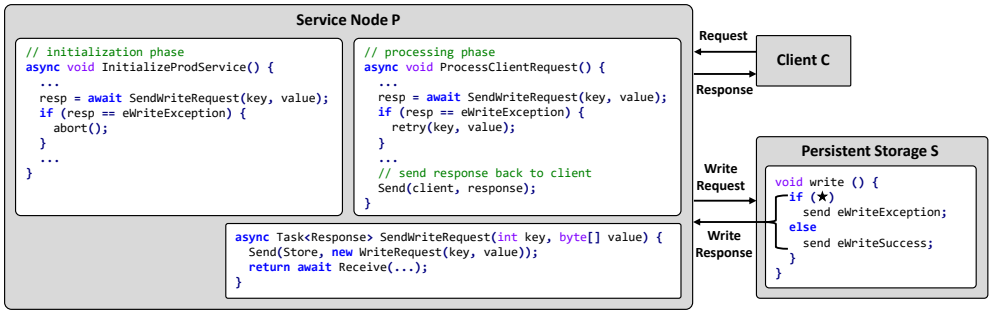

Fig. 2. Code snippets from the CacheProvider service.

gets a low priority, then it will not be scheduled until its priority is raised, which has the effect of scheduling D late in the execution. Consequently, PCT has B-$\%$ of 0.02% for η_5. Together, this implies that a string like $\eta_6 = (01)^5 2$ falls out of scope for both Random and PCT: it requires D to be scheduled late, and it requires a prefix that PCT cannot schedule. QL does not demonstrate such behavior: it has a high B-$\%$ for each of these strings (at least 1.7$\%$).

Although the program we used here is a toy example, such "patterns" do occur in real-world programs. The PoolManager service, discussed in Section 6, is one such example. Roughly, a bug in the program is triggered when a particular message m_a is processed *after* processing message m_b. However, the message m_a is enabled very early in the program's execution, somewhat like worker D in the example discussed here, and m_b is only enabled late in the program's execution. We see exactly the trend described here: Random performs poorly because it tends to schedule m_a very early, and PCT performs better because it is designed for such patterns. However, QL is able to perform even better than PCT with no explicit tuning to patterns. QL learns over time to inject m_a later and later in the execution, eventually producing a case where it is scheduled after m_b.

Data non-determinism. Programs can have non-determinism within a worker as well. A common example is the use of a Boolean ⋆ operator to model choices made outside the control of the program, for instance timeouts (that may or may not fire) or error conditions (e.g., calling an external routine may return one of several legal error codes).

Prior work on stateless schedulers does not account for such data non-determinism. The common practice is to resolve ⋆ uniformly at random: there are no heuristics that determine if one return value should be preferred over the other. With partial state observations, QL can do much better. We demonstrate this fact with a simple example. Consider rewriting the program of Figure 1 where we replace workers A and B with a single worker W that runs a loop and in each iteration of the loop, it makes a non-deterministic choice to either send 0 or 1 to C. Semantically, this program is no different from the one in Figure 1. However, PCT's performance regresses because the scheduling between workers is immaterial: only the non-deterministic choice matters, for which sampling uniformly-at-random is the only option available. Interestingly, QL retains its performance, performing just as well as it did for the program of Figure 1 because it treats all non-determinism in the same manner.

Consider the code snippet shown in Figure 2. It abstractly describes a production distributed service (CacheProvider) used in our evaluation (Section 6). The service node P uses external storage S to persist data. A read or write request sent to S can potentially fail and return an error code; this is modeled using non-deterministic choice as shown in the figure. P is expected to deal with such failed requests and take appropriate corrective action. P is designed to first go through

an *initialization phase*, followed by a *processing phase* where it handles client requests. Both phases require multiple calls to S, however, majority of the business logic of P is in the processing phase, and is the main target of testing. If a request to S fails during the initialization phase, P simply aborts by design, because it cannot proceed without proper initialization. Triggering bugs in P requires that requests to S during initialization all succeed, but may potentially fail during the processing phase. QL automatically learns to control the data non-determinism in S in such manner, whereas other schedulers perform poorly with P often aborting during initialization. Section 6 shows multiple other examples where data non-determinism is relevant to triggering bugs.

Optimizing for coverage. The QL scheduler attempts to learn scheduling decisions that increase coverage. That is, it tries to uncover new states that it has not observed before. It does not directly attempt to learn scheduling decisions that reveal the bug—the bug-finding ability is a by-product of increased coverage. Consider the "calculator" example illustrated in Figure 3.

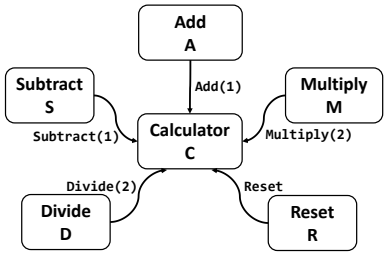

Fig. 3. A simple calculator example.

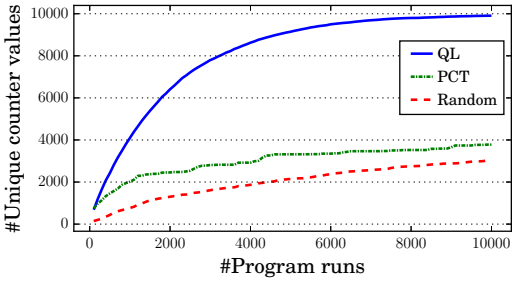

Fig. 4. Measuring coverage in the calculator example.

The worker `Calculator` maintains a counter that is initialized to 0. Each of the other workers sends exactly 100 identical messages to `Calculator`. In response to a message sent by worker `Add`, `Calculator` increments its counter by 1. Similarly, in response to a message sent by `Subtract`, `Multiply`, `Divide`, and `Reset`, the `Calculator` worker will subtract 1, multiply by 2, divide (integer division) by 2, and reset the counter to 0, respectively. The counter in `Calculator` is limited to the range $[-5000, 5000]$, for a total of 10000 unique counter valuations.

In this example, there are a large number of concurrent executions. However, their effect is such that many of them result in the same `Calculator` state. Figure 4 shows the coverage that each scheduler can achieve as we increase the number of explored program executions. Here, coverage is defined as the number of distinct counter values generated across all executions. QL dramatically outperforms both Random and PCT: it is able to traverse almost the entire state space of 10000 unique counter values. We can gain some important insights into the superior state coverage of QL via Figure 5. The subfigures show the number of times each of the `Add`, `Subtract`, `Multiply`, `Divide` and `Reset` messages are sent, in intervals of 100 program runs. The QL scheduler learns very quickly that scheduling `Reset` hampers coverage, so it de-prioritizes it. QL can learn this fact because `Reset` always takes the counter back to 0, a state that it has seen many times before. Similarly, QL also realizes that `Divide` is not that useful either for generating new counter values, although it is still more useful than a `Reset`. Note that the learning is specific to the `Calculator` example and obtained by running it repeatedly.

Figure 5 shows a uniform distribution for the Random scheduler, as expected. PCT has more variance: some runs prefer one type of action over the others. However, there is no learning; the distribution of one action is indistinguishable from the others.

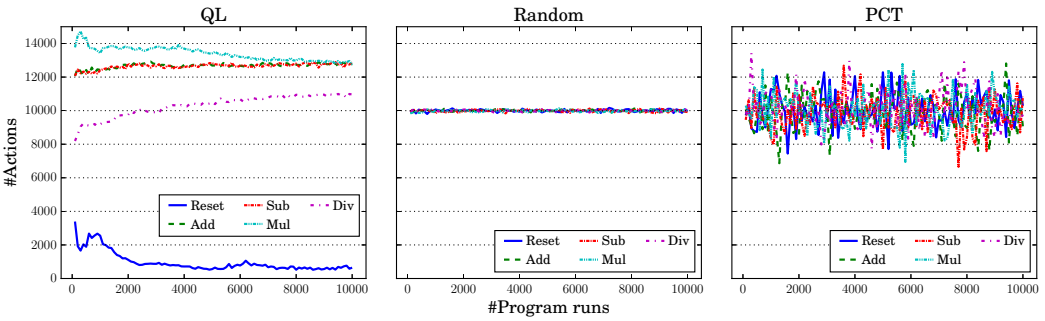

Fig. 5. Distribution of actions taken across program runs (best viewed in color).

Picking the right state abstraction. The above examples show that learning from state observations provides an effective alternative to generic (program-independent) scheduling heuristics. This leaves open the question of what constitutes a good state observation? If we make the observation too imprecise (e.g., not observing any state), then `QL` deteriorates to Random-like performance because there is nothing to learn. For the `Calculator` example, if `QL` cannot observe the calculator value (i.e., all observations return the same state) then it is only able to cover 2715 states in 10K program runs, almost the same as Random.

On the other hand, we cannot make the observation too precise either. For one, it is very difficult (and expensive) to hash the entire state of a running program. Moreover, it can also make the state space very large and slow down learning. We find that in real-world systems, often times programs have a lot of auxiliary state (e.g., logs and other debugging information) that is irrelevant to the correctness of their implementation. For instance, in the `Calculator` sample, suppose that worker `C` maintains a count of the number of operations that it has processed in a run of the program. This count starts at zero and is incremented each time `C` processes a message from any worker. Further, we hash both the calculator value as well as this counter. In this case, `QL` is only able to cover 1693 calculator states in 10K program runs, even worse than Random. The learning slows down because every observation made in a single run of the program is unique.

It is thus clear that *observing the right amount of state* is important for the performance of `QL`. This poses a stumbling block because predicting the performance of a learning-based algorithm can be challenging. Characterizing exactly when the algorithm will perform well is hard, often impossible. Our approach then is to resort to an empirical analysis to find evidence of effectiveness of `QL` in practice. For this, we rely on the following intuition: we only observe state that is *relevant* for the concurrency in the program. If `QL` is able to maximize the coverage of such relevant state, then it should also be good at finding bugs in the program.

We break the observation of state into two parts. The first, called a *default* observation, is to hash the *synchronization state* of the program that is readily available from the CCT tool itself. Any CCT tool needs to control the scheduling among the workers of a program, for which it must know which workers are *enabled* to execute a step, or correspondingly, which workers are *disabled* or *blocked*. For instance, a CCT tool for multi-threaded programs will maintain the set of locks held by each thread. (A thread that is about to acquire a lock held by another thread is *disabled*.) In a message-passing program, a CCT tool will typically maintain the set of pending message for each process. (A process with no pending message is *disabled* and the rest are *enabled*.) This synchronization state maintained by the CCT tool forms the default state observation. This state is naturally relevant because it captures the state of synchronization between workers: holding a lock

or sending a message is how one worker informs another of what it is doing. Furthermore, because the information is available from the CCT tool itself, it requires no user involvement.

The second kind of state observation is called a *custom* state observation, where a user can apply their program-specific intuition to capture additional state that might be relevant in the program. (We make these concepts concrete in Section 5.) We now illustrate the use of these state observations through a real-world example.

The Raft Protocol. `Raft` is a widely used consensus protocol [Ongaro and Ousterhout 2014]. Such protocols form the basis of many distributed systems and hence it is important to get them right. However, testing that they have been correctly implemented, especially in corner cases can be challenging, thus, they are a good target for CCT.

Each node in a `Raft` cluster can be in one of three roles: follower, candidate or leader. The leader is responsible for receiving client requests and replicating them across the rest of the nodes. Each follower keeps tracks of all requests received from the leader. A candidate can, at any time, start a leader election process. If it wins the election, it becomes a leader. (This is necessary to tolerate failures of the leader node.) An election consists of nodes exchanging *voting* messages with each other; the candidate that receives a quorum (majority) of votes becomes the leader.

`Raft` ensures that at most one node can be the leader of the cluster at any point in time. Having two leaders is a serious bug; it can result in requests from one overwriting the other and corrupting the state of the system. We consider a particular implementation of `Raft`[4] with such a bug [Akka Raft 2015]. In this case, the candidate node was missing logic that is supposed to de-duplicate votes originating from the same follower during the same leader election. This de-duplication is important. A follower node can send the same vote multiple times, if it believes that earlier votes were dropped by the network. It relies on a timeout to determine possibly-dropped messages; it cannot know for sure. If the candidate does not perform de-duplication (i.e., discard duplicate messages), then it could end up counting the same vote multiple times and transition to become the leader role without actually having achieved a quorum of votes.

To trigger the bug, in a particular election round, it must happen that candidate A receives a majority of the votes *and* a follower that voted for candidate B ends up sending multiple duplicate votes, so that both candidates end up claiming that they are the leaders. (This bug was fixed as follows. Instead of maintaining a count of the number of votes received, a candidate should maintain a set of all followers who have voted for it. Adding to the set will automatically do the de-duplication. The candidate then checks the size of the set to know if it has a quorum or not.)

For a developer interested in testing their `Raft` implementation, clearly they do not know of bugs beforehand. Instead, their goal is to write a test that covers as many diverse behaviors as possible in a given amount of budget. For this, they make relevant state available to `QL`. This consists of in-flight messages (captures the state of the network) as well as the status of each node (its role, and whom they are choosing to vote for currently). Let us call this state as "runtime state" of the protocol. Other components such as the linearized log of requests maintained by nodes are not deemed relevant.

We consider three variants of `QL` that incrementally include more relevant state. The variant $\texttt{QL}^{\texttt{i}}$ only considers in-flight messages (by hashing the inbox of each node). $\texttt{QL}^{\texttt{d}}$ additionally also considers the current role of each node. $\texttt{QL}^{\texttt{c}}$ considers the entire runtime state. (In our implementation, as we make it clear in Section 5, $\texttt{QL}^{\texttt{d}}$ is the *default* configuration where the state observation only consults the CCT runtime.) Note that `QL` only gets to observe a hash of the state and it does not know how to interpret the hashed value. It must learn that on its own.

[4]Colin Scott's blog nicely details this bug. See: https://colin-scott.github.io/blog/2015/10/07/fuzzing-raft-for-fun-and-profit/

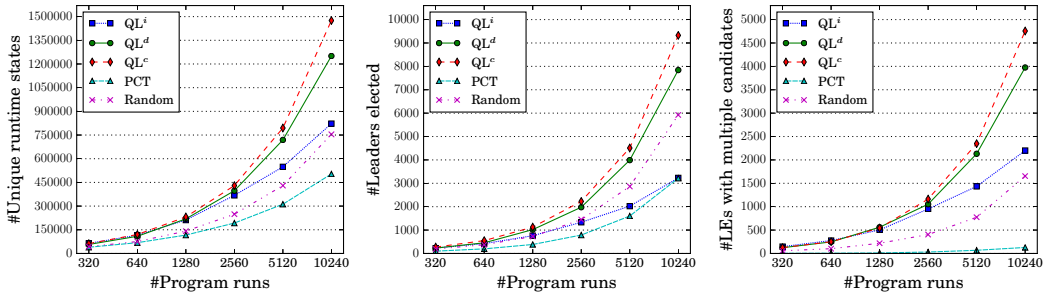

Fig. 6. Measuring coverage of runtime states (left), leader elections (middle) and multiple candidates during the same leader election (right) in the Raft implementation.

As we will see in Section 6, all variants of QL find the bug with a much higher frequency than other schedulers. We use Figure 6 to understand the effectiveness of QL. The figure shows a comparison of the coverage obtained by CCT using different schedulers over 10K program runs. The three plots in the figure are all obtained from the same set of runs, but they measure coverage in different ways. The left-most plot measures coverage in the number of runtime states. The schedulers $\texttt{QL}^{\texttt{c}}$ and $\texttt{QL}^{\texttt{d}}$ are very close to each other and offer much superior coverage compared to other schedulers and even $\texttt{QL}^{\texttt{i}}$ to a smaller extent.

The other two plots measure coverage that is more directly relevant for the particular bug in this implementation of Raft. The middle plot measures the number of leaders elected, whereas the right-most plot measures the number of times there were multiple candidates during the same leader election. Both of these are required for the bug to manifest. $\texttt{QL}^{\texttt{c}}$ and $\texttt{QL}^{\texttt{d}}$ perform best in these metrics as well. While Random is better at electing leaders than $\texttt{QL}^{\texttt{i}}$, it is worse off at having multiple candidates in an election.

Random and PCT are program agnostic: without knowing the program under test, they are unable to effectively push Raft into semantically interesting cases. QL, on the other hand, leverages state coverage as a means of generating more interesting behaviors.

We do note that increased coverage does not always imply better bug-finding ability. The state spaces of such programs can be very large, and each bug can require a very different kind of coverage. Nonetheless, we rely on empirical evidence to show that for all bugs that we consider, QL, with default observation only, consistently delivers best coverage and bug-finding ability, whereas other schedulers only tend to do well on a subset.

3 PRELIMINARIES

This section presents background material on CCT and reinforcement learning and sets up the notation that we follow in the rest of the paper.

3.1 Controlled Concurrency Testing

3.1.1 Programming Model. A program $\mathcal{P}$ is a tuple $\langle \mathcal{W}, \Sigma, \sigma_{init}, A, \Theta \rangle$, where $\mathcal{W}$ denotes a (finite) set of concurrently executing *workers* and Σ denotes the set of program *configurations*, with $\sigma_{init} \in \Sigma$ being the initial configuration. The set of *actions* that the program can execute is denoted by A. We use the meta-variables σ, a and w to range over the sets Σ, A and $\mathcal{W}$ respectively. Let $\omega : A \mapsto \mathcal{W}$ be an onto function that maps each action to the unique worker which executes it. The function $\Theta : A \mapsto 2^{\Sigma \times \Sigma}$ denotes the set of *transitions* associated with a. We write $\sigma \xrightarrow{a} \sigma'$ iff $(\sigma, \sigma') \in \Theta(a)$. Further, for simplicity, we assume that transitions associated with an action are deterministic. That

is, for each action a, if $\sigma_1 \xrightarrow{a} \sigma_2$ and $\sigma_1 \xrightarrow{a} \sigma_3$ then $\sigma_2 = \sigma_3$. (Any non-determinism must then be reflected in the choice of which action to execute.)

We define some helper functions that will be used in the rest of the paper. The function $IsError(\sigma)$ returns true if the configuration σ represents an erroneous configuration, and returns false otherwise. We define the functions $enabled(\sigma)$, $enabled^w(\sigma)$ and $\vartheta(\sigma)$ as follows:

$$enabled(\sigma) \stackrel{\text{def}}{=} \{a \in A \mid \exists \sigma' \in \Sigma : \ \sigma \xrightarrow{a} \sigma'\}$$
$$enabled^w(\sigma) \stackrel{\text{def}}{=} \{a \in enabled(\sigma) \mid \omega(a) = w\}$$
$$\vartheta(\sigma) \stackrel{\text{def}}{=} \{w \in \mathcal{W} \mid enabled^w(\sigma) \neq \emptyset\}$$

The function $enabled(\sigma)$ returns the set of all actions that $\mathcal{P}$ can execute when it is at the configuration σ, while $enabled^w(\sigma)$ returns those actions in $enabled(\sigma)$ that can be executed by a worker w. Lastly, $\vartheta(\sigma)$ returns all workers that have at least one enabled action in the configuration σ. With this notation, $|\vartheta(\sigma)| > 1$ represents *control* non-determinism: there is a choice between which worker will take the next step. When $|enabled^w(\sigma)| > 1$, it represents *data* non-determinism: the worker w itself can have multiple enabled actions.

Note that this programming model is very general. It does not assume anything about how different workers communicate. One can easily encode message-passing programs or shared-memory programs in this model.

3.1.2 Schedulers. A *schedule* ℓ of length N is defined to be a sequence of program transitions starting from the initial configuration:

$$\ell \stackrel{\text{def}}{=} \left\langle \sigma_{init} \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \ldots \xrightarrow{a_N} \sigma_N \right\rangle$$

We use $|\ell|$ to denote the length of a schedule, and write $\ell \xrightarrow{a'} \sigma'$ to denote a schedule that extends ℓ with the single (valid) transition $\sigma_N \xrightarrow{a'} \sigma'$. We also refer to the transition $\sigma_{i-1} \xrightarrow{a_i} \sigma_i$ as the i-th step of the schedule. We call a schedule *buggy* if its final configuration represents an error. Note that because transitions associated with an action are deterministic, a schedule can equivalently be represented by its sequence of actions.

For a given program $\mathcal{P}$, CCT aims to explore possible schedules of $\mathcal{P}$ using the generic exploration algorithm shown in Algorithm 1. The algorithm is parameterized by a scheduler Sch. It accepts bounds M and N on the length of each schedule, and the total number of schedules to explore, respectively. Algorithm 1 returns a buggy schedule if discovered, else it returns an empty sequence.

Algorithm 1 iteratively explores multiple schedules of $\mathcal{P}$, up to bound N (line 1). In each iteration, the scheduler is informed via a call to $PrepareNext$ to prepare for executing a new schedule (line 3). Each schedule consists of at most M steps, after which the schedule is aborted and a new one is attempted. In each step, the scheduler Sch is asked to pick an action from the set of all enabled actions via a call to $GetNext$ (line 8). The selected action is then executed (line 9): given σ and a, $Execute(P, \sigma, a)$ returns σ' such that $\sigma \xrightarrow{a} \sigma'$. The process continues until a bug is found or the algorithm hits the bound N on the number of explored schedules.

The scheduler Sch controls the exploration strategy. We describe two different scheduler instantiations next.

Random. A purely randomized exploration strategy can be obtained by setting the $PrepareNext$ function to `skip`, and making the $GetNext$ function return an action chosen uniformly at random from its given set of enabled actions $enabled(\sigma)$. Prior work has noted that such a simple strategy

Algorithm 1: Generic Exploration Algorithm.

Input: Scheduler Sch
Input: Program $\mathcal{P}$, Max-Steps M, Max-Iterations N

```
1  foreach i ∈ {1, ..., N} do
2      σ ← σ_init, ℓ ← ⟨σ_init⟩
3      Sch.PrepareNext()
4      foreach j ∈ {1, ..., M} do
5          if IsError(σ) ∨ enabled(σ) = ϕ then
6              break
7          end
8          a ← Sch.GetNext(enabled(σ))
9          σ' ← Execute(P, σ, a)
10         ℓ ← (ℓ --a--> σ')
11         σ ← σ'
12     end
13     if IsError(σ) then
14         return ℓ
15     end
16 end
17 return ⟨⟩
```

is still effective at finding bugs in practice and should be used as a baseline for future comparisons [Thomson et al. 2016].

Probabilistic concurrency testing (PCT). The PCT scheduler described here is an adaptation of the original algorithm [Burckhardt et al. 2010] to our setting. PCT is a priority-based scheduler that is parameterized by a given bound D, called the priority-change-point budget. We explain PCT with a fixed number of workers, although it is easily extended for an unbounded number of workers. Let $|\mathcal{W}| = W$. The *PrepareNext* method of the scheduler (randomly) assigns a unique initial priority to each worker in the range $\{D, D+1, \ldots, D+W\}$. It also constructs a set $Z = \{z_1, \ldots, z_{D-1}\}$ of $D-1$ distinct numbers chosen uniformly at random from the set $\{1, \ldots, M\}$. Assume that Z is sorted, so that $z_j \leq z_{j+1}$ for each $j \leq D-2$.

The idea behind PCT is to choose the highest-priority worker within the set of enabled workers at each step. Priorities remain fixed, except at priority-change points when the scheduler shifts priorities. More precisely, when *GetNext* is called for choosing the i^{th} action, it first picks the highest-priority worker w. Next, it checks if i equals z_j for some $z_j \in Z$. If so, it decreases the priority of w to j. (Note that $j \leq D-1$, so it is the least priority among all currently-assigned priorities.) It then re-picks the highest-priority worker w'. Finally, it returns an enabled action of w' chosen uniformly at random from its set of all enabled actions. (This last part deals with data non-determinism.)

Schedulers need not always be probabilistic or randomized. It is also possible to define a DFS-like scheduler that is guaranteed to explore all schedules of the program in the limit, although such systematic schedulers tend not to work well in practice [Thomson et al. 2016]. There are many other schedulers defined in prior literature [Desai et al. 2015; Emmi et al. 2011; Musuvathi and Qadeer 2007; Thomson et al. 2016].

3.2 Reinforcement Learning

The Reinforcement Learning (RL) [Heidrich-Meisner et al. 2007; Russell et al. 2003; Sutton and Barto 1998] problem, outlined in Figure 7, comprises of an agent interacting with an environment, about which it has no prior knowledge. At each step, the agent takes an action, which causes the environment to undergo a state transition. The agent then observes the new state of the environment, and receives feedback in the form of a reward or penalty. The goal of the agent is to learn a sequence of actions that maximizes its expected reward.

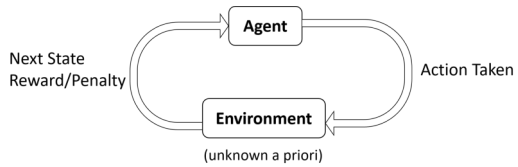

Fig. 7. The Reinforcement Learning problem.

The environment is unknown a priori, i.e., the effect of executing an action is not known. This makes the RL problem hard, but also generally applicable. In the RL literature, it is common to model the environment as a Markov Decision Process (MDP) over the partial state observation. An MDP is a stochastic state-transition model, comprising the following components:

(1) A set of states $\mathcal{S}$, representing partial observations of the environment's actual state.
(2) A set of actions A that the agent can instruct the environment to execute.
(3) Transition probabilities $\mathcal{T}(s, a, s')$, which denote the probability that the environment transitions from a state $s \in \mathcal{S}$ to $s' \in \mathcal{S}$, on taking action $a \in A$.
(4) Reward $\mathcal{R}(s, a)$ obtained when the agent takes an action a from the state s.

Assume that the environment is in some state $s \in \mathcal{S}$, and the agent instructs it to execute a sequence of actions $(a_1, a_2, \ldots)$, denoted as $\langle a_t \rangle$. Let s_{i+1} denote the state of the environment after executing the action a_{i+1} at state s_i. Then, the *expected discounted reward*, for s and $\langle a_t \rangle$ is defined as

$$\mathcal{V}(s, \langle a_t \rangle) \stackrel{\text{def}}{=} \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i \mathcal{R}(s_i, a_{i+1}) \;\middle|\; s_0 = s \right]$$

The parameter $\gamma \in (0, 1]$ is called the *discount factor*, and is used to strike a balance between immediate rewards and long-term rewards. The *optimal* expected discounted reward for a state s can be written as:

$$\begin{aligned} \mathcal{V}^*(s) &\stackrel{\text{def}}{=} \max_{\langle a_t \rangle} \mathcal{V}(s, \langle a_t \rangle) \\ &= \max_{a \in A} \left(\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') \mathcal{V}^*(s') \right) \end{aligned}$$

The Q-function $Q^*(s, a)$ denotes the expected value obtained by taking action a at state s, and is written as

$$Q^*(s, a) \stackrel{\text{def}}{=} \left(\mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') \mathcal{V}^*(s') \right) \tag{1}$$

The agent makes use of a *policy* function, $\pi : \mathcal{S} \mapsto A$, to determine the action to be executed at a given state of the environment. The *optimal* policy $\pi^*(s)$ can be obtained by computing the Q-function for each state-action pair, and then selecting the action a that maximizes $Q^*(s, a)$ (by solving the Bellman equations [Bellman et al. 1954]). However, when the MDP is unknown, instead

of *computing* the optimal policy, the agent needs to adaptively *learn* the policy from its history of interaction with the environment. RL techniques help in systematic exploration of the unknown MDP by learning which sequences of actions are more likely to earn better rewards (or incur less penalties).

One such popular RL algorithm is *Q-Learning* [Watkins and Dayan 1992; Watkins 1989b], which estimates Q^* values using point samples. Starting from an initial state s_0, Q-Learning iteratively selects an action a_{i+1} at state s_i and observes a new state s_{i+1}. Let $\langle s_t \rangle$ be a sequence of states obtained by some policy and $\langle a_t \rangle$ be the sequence of corresponding actions. Then, given an initial estimate of Q_0, the Q-learning update rule is as follows:

$$Q_{i+1}(s_i, a_{i+1}) \leftarrow (1-\alpha) \cdot Q_i(s_i, a_{i+1}) + \alpha \cdot \left(\mathcal{R}(s_i, a_{i+1}) + \gamma \max_{a'} Q_i(s_{i+1}, a') \right) \tag{2}$$

Here, $\alpha \in [0, 1]$ is called the *learning parameter*. When $\alpha = 0$, the agent does not learn anything new and retains the value obtained at the i^{th} step, while $\alpha = 1$ stores only the most recent information and overwrites all previously obtained rewards. Setting $0 < \alpha < 1$, helps to strike a balance between the new values and the old ones.

The Q-Learning algorithm does not explicitly use the transition probability distribution of the underlying MDP during the update step (unlike Equation 1), and is hence called a *model-free* algorithm. Such algorithms are advantageous when the state-space is huge and it is computationally expensive to try learning all the transition probability distributions.

Exploration methods like *random* completely ignore the historical agent-environment interactions [Whitehead 1991a,b], while *counter-based methods* [Barto and Singh 1991; Sato et al. 1988] take decisions based solely on the frequency of visited states. In contrast, Q-Learning takes informed stochastic decisions that eventually make worse actions less likely. In our work, we use Softmax [Sutton and Barto 1998; Watkins 1989a] as our policy in order to bias the exploration against unfavorable actions based on the current Q-values. According to the Softmax policy, the probability of choosing an action a (from a set of possible choices A) at state s, is given by $\frac{e^{Q(s,a)}}{\sum_{a' \in A} e^{Q(s,a')}}$. Thus, lower the $Q(s, a)$ value, lesser the likelihood of taking action a again at state s. We note that there are alternatives to the Softmax policy, such as ϵ-greedy (with probability ϵ, choose a random action and with probability $(1 - \epsilon)$ choose the action with max Q value). We found Softmax to be the most natural choice, and leave experimentation with other choices as future work.

4 Q-LEARNING-BASED CONTROLLED CONCURRENCY TESTING

This section describes our QL scheduler. Let $\mathcal{H}$ be a user-defined function that maps $\Sigma \mapsto \mathcal{S}$, where Σ is the set of program configurations and $\mathcal{S}$ is a set of *abstract states*. When the environment (program) is in the configuration σ, then $\mathcal{H}(\sigma)$ represents the observation that the agent will make about the environment. One can think of $\mathcal{H}$ as defining an abstraction over the program's configuration space. In reality, $\mathcal{H}$ is implemented as a hashing function that is applied to only a fraction of the program's configuration. The reward function $\mathcal{R} : \mathcal{S} \times A \mapsto \mathbb{R}$ is fixed to be a constant -1, that is, $\mathcal{R}(s, a) = -1$ for all states s and actions a.

Given a schedule $\ell = \langle \sigma_{init} \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \sigma_n \rangle$, let $\mathcal{H}(\ell)$ denote the *abstracted* schedule with each σ_i replaced by $s_i = \mathcal{H}(\sigma_i)$. The QL scheduler is parameterized by the abstraction function $\mathcal{H}$. Its *GetNext* and *PrepareNext* procedures are described in Algorithm 2 and Algorithm 3, respectively. The QL scheduler maintains a partial map $\mathcal{Q} : \mathcal{S} \times A \mapsto \mathbb{R}$. For an abstract state s and action a, $\mathcal{Q}(s, a)$, when defined, represents the Q-value associated with the state-action pair (s, a).

Algorithm 2 takes as input a program configuration σ, and the set of (say, n) actions $enabled(\sigma)$. We fix an arbitrary ordering among these actions and use a_i to refer to the i^{th} action in this order.

Algorithm 2: GetNext-QL

Input: Set of actions $\{a_1, \ldots, a_n\}$, Configuration σ

1 $s \leftarrow \mathcal{H}(\sigma)$
2 **foreach** $a \in \{a_1, \ldots, a_n\}$ **do**
3 **if** $Q(s, a)$ ***is undefined*** **then**
  /* Initialize Q-value of new (s, a) pair to 0 */
4 $Q(s, a) \leftarrow 0$
5 **end**
6 **end**
7 $\mathcal{D} \leftarrow \langle\rangle$ /* probability distribution over actions */
8 **foreach** $i \in \{1, \ldots, n\}$ **do**
9 $\mathcal{D}(i) \leftarrow \frac{e^{Q(s,a_i)}}{\sum_{j=1}^{n} e^{Q(s,a_j)}}$
10 **end**
11 $i \leftarrow \texttt{Sample}(\mathcal{D})$
12 **return** a_i

The *GetNext* procedure first computes $\mathcal{H}(\sigma)$ and stores it in the variable s. For each input action a, if the Q-value for (s, a) is not present in Q, then it is initialized to 0 (lines 2-6). Next, a variable $\mathcal{D}$ is initialized. Lines 8-10 creates a probability distribution $\mathcal{D}$ over the set of n enabled actions using the Softmax policy. Finally, Algorithm 2 samples from the distribution $\mathcal{D}$ (i.e., it picks i with probability $\mathcal{D}(i)$) and returns the corresponding action.

Algorithm 3: PrepareNext-QL

Input: Schedule $\ell = \langle \sigma_0 \xrightarrow{a_1} \sigma_1, \ldots, \sigma_{n-1} \xrightarrow{a_n} \sigma_n \rangle$

1 $\hat{\ell} \leftarrow \mathcal{H}(\ell)$ /* Sets $\hat{\ell}$ to $\langle s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n \rangle$ */
2 **foreach** $i \in \{n, \ldots, 1\}$ **do**
3 $\texttt{maxQ} \leftarrow \max_a Q(s_i, a)$
4 $Q(s_{i-1}, a_i) \leftarrow (1 - \alpha) \cdot Q(s_{i-1}, a_i) + \alpha \cdot \left(\mathcal{R}(s_{i-1}, a_i) + \gamma \cdot \texttt{maxQ}\right)$
5 **end**

We use the *PrepareNext* procedure to update the Q-values according to the previously executed schedule. We assume that *PrepareNext* is not called in the first iteration of Algorithm 1, and in each subsequent iteration, it is passed the completed schedule from the previous iteration. Algorithm 3 updates the Q-values for each state-action pair (s_{i-1}, a_i) in the abstracted schedule $\hat{\ell}$, according to Equation 2. Note that the negative reward causes the update rule to decrease the Q-value for each observed (s, a) state-action pair. As a result, the Softmax policy lowers the likelihood of selecting a in subsequent iterations when the state s is encountered, thereby increasing the chances of discovering newer states.

5 IMPLEMENTATION

The QL scheduler needs a setup for performing Controlled Concurrency Testing of a given program. Any CCT framework would suffice as long as it allows the following:

(1) the ability to implement the abstraction function $\mathcal{H}$ for observing the current configuration of the program,
(2) the ability to retrieve the current set of enabled actions, and

(3) the ability to execute only the chosen action from the set of enabled actions.

We used P# [Deligiannis et al. 2015] as the implementation vehicle for QL.

The P# Framework. P# is an open-source industrial-strength framework for building and testing concurrent C# applications [P# Team 2019]. It consists of two main components: a runtime library and a CCT tool called the P#-Tester. The P# runtime is a regular .NET library that offers APIs for creating actors (called *machines* in P#) and sending messages between them. The programmer simply links their C# application against this library. Instead of using threads and locks for concurrency, the programmer is expected to use P# machines that communicate with each other via messages.

A machine consists of an inbox of incoming messages along with other user-defined fields. The P# runtime executes each machine in a single-threaded fashion, where it dequeues messages from the machine's inbox and processes them sequentially one after the other. The programmer can define custom logic for processing messages as regular C# methods using the full extent of the language. These methods can update the machine's local fields, or invoke other runtime APIs that create new machines or send messages to existing ones.

The P# runtime also provides an API, called NondDet, that returns an unconstrained Boolean value; a program can use NondDet to model data non-determinism. Optionally, a machine can internally have a state-machine (SM) structure that is offered in P# for programmatic convenience. P# also provides the ability to program safety and liveness specifications of the application alongside its implementation. Violating any of these specifications or a user assertion or an uncaught exception, all constitute a *bug* in the program that should be caught before the program is put into production.

The P#-Tester. P#-Tester implements Algorithm 1. A worker is simply a machine. An enabled worker is a machine that has a message in its inbox that it has not processed already. The P# runtime maintains the inbox for each machine, hence this information is readily available to the P#-Tester. (Our QL strategy will also leverage the information maintained by the runtime for the purpose of implementing the abstraction function $\mathcal{H}$.)

P#-Tester takes control of the scheduling of machines in the P# runtime, which then allows it to serialize the execution of the program, allowing only one machine to proceed at a time. When a machine is scheduled, it executes until it reaches a *scheduling point*, at which point the control goes back to the P#-Tester and it can choose to schedule some other machine. The notion of an action is thus the execution of a machine from one scheduling point to the next.

It is generally the role of the CCT tool to ensure that there are enough scheduling points inserted in the program so that all its executions can be covered in the limit. For instance, inserting a scheduling point before each machine instruction is clearly enough (when using a sequentially-consistent memory). However, this can be very intrusive, and may considerably slow down the execution of the program. A scheduling point before non-commuting actions is enough, based on Lipton's theory of reduction [Lipton 1975]. P# assumes a data-race free program [Deligiannis et al. 2015]. In that case, the only non-commuting actions in a P# program are: (1) the creation of a machine and (2) the sending of a message. Both of these can only happen via calls into the P# runtime, at which point the P#-Tester can take control. We note that the choice of scheduling points and dealing with assumptions such as data-race freedom and sequentially-consistent memory semantics, are specific to the CCT implementation, not the scheduling strategy. These concerns are thus orthogonal to the contributions of this paper.

P#-Tester allows a user to choose between various different schedulers that it has implemented already, including Random and PCT. We implemented and added QL to P#-Tester that allows for a direct comparison with other schedulers.

The P#-Tester also inserts a scheduling point at each call to NondDet and it is up to the scheduler to pick the value to return. A machine, at any point, can have at most one enabled action, except in cases when it is at a call to NondDet, in which case it can have two enabled actions (one for each possible return value of NondDet). The resolution of NondDet in all schedulers that come with P#-Tester is purely random. QL is the first scheduler that does otherwise.

Our decision to use P# was motivated by the following reasons:

(1) There are production distributed services of Microsoft Azure that have been developed using P# [Deligiannis et al. 2020]. This allowed us to evaluate QL on highly concurrent services comprising tens of thousands of LOC. Several other benchmarks that use P# were available as well.
(2) P#-Tester contains implementations of several state-of-the-art scheduling strategies, including Random and PCT. This allowed us to compare QL against independently-implemented schedulers that have been shown to be effective in practice.
(3) P# provides the necessary scaffolding to quickly implement the QL scheduler. The framework exposes APIs to perform suitable abstractions of the program configuration, retrieve the set of actions, etc., and has been used in prior studies as well [Deligiannis et al. 2016; Mudduluru et al. 2017; Ozkan et al. 2018].

Extension to multithreaded applications. We further implemented an extension to P# that allows the creation of *tasks*. Tasks execute concurrently while communicating with other tasks via shared-memory. Tasks also have access to synchronization, such as blocking on another task to finish (*join*) or acquiring/releasing locks. This extension allows us to exercise the various schedulers of P# on multi-threaded benchmarks as well.

In this case, the task corresponds to a worker and an action is the execution of a task from one scheduling point to the next. The treatment of NondDet also remains the same. The P# runtime maintains, for each task, the set of locks currently held by it or if one task is waiting on another to finish (join). This information is used by the P#-Tester to know if a task is enabled or not. By default, the P#-Tester considers scheduling points at task creation, join points and at acquire of locks. This is sufficient to cover all executions of a data-race free program. In case the programmer feels that certain shared-memory accesses in the program may be racy, they have to manually insert a scheduling point (just before the access) via an API provided by the P# runtime.

QL *scheduler.* We experimented with several different $\mathcal{H}$ functions for obtaining state observations. Each of them were implemented as a hashing routine that mapped the current program configuration to an integer; they only differed in how much of the program state was hashed. In each case, we first constructed a per-worker (per-machine or per-task) hash and then applied a commutative hash (in our case, a simple multiplication) of these individual hashes. The commutative hashing allows us to remove distinction between two program configurations that only differ in the enumeration order of workers. (The enumeration order can be made relevant simply by including the worker's unique ID in the per-worker hash. In all the $\mathcal{H}$ functions that we considered, however, we did not include the worker ID.)

We considered three variants of QL that use a different per-worker hash, called $\texttt{QL}^{\texttt{i}}$, $\texttt{QL}^{\texttt{d}}$ and $\texttt{QL}^{\texttt{c}}$. Following the description in Section 2, the schedulers $\texttt{QL}^{\texttt{i}}$ and $\texttt{QL}^{\texttt{d}}$ use default observations, i.e., they only rely on information present in the P# runtime, whereas $\texttt{QL}^{\texttt{c}}$ accepts a custom state observation. These are described in more detail below.

(1) $\texttt{QL}^{\texttt{i}}$: hashes only the contents of the inbox of a machine. This only applies to machine-based programs.

(2) $\texttt{QL}^{\texttt{d}}$: For machine-based programs, this hashes the inbox contents, current machine operation (send or create or `NonDet`), as well as the current state of the machine's state-machine (if any). For the case of task-based applications, this only hashes the locks held by a task.
(3) $\texttt{QL}^{\texttt{c}}$: hashes a machine/task using a user-defined hashing routine that is free to look at any runtime component used by $\texttt{QL}^{\texttt{d}}$, as well as the local state of the machine/task. We added convenient APIs to the P# library to let a user provide this hashing scheme.

It is important to note that because $\texttt{QL}^{\texttt{i}}$ and $\texttt{QL}^{\texttt{d}}$ only make use of information already present in the P# runtime, they require no user involvement. They will work out-of-the-box for any P# program. Thus, these variants serve to demonstrate the general power of our scheduler, whereas $\texttt{QL}^{\texttt{c}}$ demonstrates how a user can step in to control the testing quality.

Optimizations. Our implementation of QL in P#-Tester closely mirrors Algorithm 2 and Algorithm 3, with the exception of two optimizations that we describe next. These optimizations improve upon the basic reward function defined in Section 4.

- Opt-1: The implementation maintains a map $\texttt{TF} : \mathcal{S} \rightarrow \texttt{int}$. $\texttt{TF}[s]$ records the number of times the scheduler has encountered the hash s during exploration. Our first optimization is that for a given hash s, we multiply the reward in line 4 of Algorithm 3 with $\texttt{TF}[s]$. This optimization allows QL to rapidly learn to avoid exploring a state repeatedly.
- Opt-2: The second optimization involves assigning a high negative reward (-1000) for an action a that corresponds to sending a message of a *special* type, from any state s. This change has the following effect. For some state s and a special action a, once the scheduler has explored a schedule that takes action a on state s, it will be highly discouraged to fire action a in the future when the program is in state s. This optimization helps the scheduler improve its diversity of schedules when it comes to the insertion of this special action. In our experiments, this special action is the injection of a *failure* message that is used by programmers to test the failover logic of their distributed system. A failure message is intended to bring down a set of machines, so one can test the effects of failures in their system.

These optimizations, along with the choice of the abstraction function showcase the degree of control available to a user for enhancing the testing experience for the program they care about most. In contrast, other schedulers offer no control to the user. In our evaluation, we added the scheduler $\texttt{QL}^{\texttt{b}}$, which is a basic version of $\texttt{QL}^{\texttt{d}}$ with both Opt-1 and Opt-2 turned off.

For all our experiments, we set the values of α and γ, used in Algorithm 3, to 0.3 and 0.7, respectively. We justify this choice in Section 6.

The ability to observe the program configuration is an added source of information for QL compared to stateless schedulers. To evaluate the effectiveness of the learning aspects of the algorithm, we came up with two other strategies to serve as a baseline for QL, described below.

Greedy scheduler. The `Greedy` scheduler maintains a TF map, similar to QL. When the program reaches a state s, the scheduler computes the set $\mathcal{N}_s$ of possible target states the program can transition to from s:

$$\mathcal{N}_s = \{s' \mid \exists a \in A : s \xrightarrow{a} s'\}$$

The `Greedy` scheduler chooses a state $s' \in \mathcal{N}_s$ having the lowest $\texttt{TF}[s']$ value, executes the corresponding action and increments $\texttt{TF}[s']$. In case of a tie among several s' states, `Greedy` draws the target state uniformly at random.

Iterative delay-bounding scheduler. We also compare against a variant of the *delay-bounding* (DB) [Emmi et al. 2011] scheduler. In our experiments, we found the original DB algorithm to be ineffective at exposing bugs, so we started with an optimized version of it [Desai et al. 2015] and

modified it to leverage state observations. In our variant, which we call *iterative delay-bounding* (IDB), at each scheduling decision, it either selects an action of the worker w that executed the last action, or randomly context-switches to a worker different from w. The number of such random context-switches per run is bounded (called the *delay bound*). IDB keeps tracks of the visited states as it performs exploration. It starts with a delay bound of 0 and then iteratively increments it when no new states are discovered in the last 100 runs.

6 EVALUATION

This section describes our empirical evaluation that compares QL against other schedulers in terms of coverage and bug-finding ability. We also measure robustness to data non-determinism, as well as the effect of different state observations on the performance of QL. All our experiments use the P#-Tester. We used existing implementations of schedulers when available (Random and PCT) and implemented others ourselves (Greedy and IDB). There were other schedulers [Thomson et al. 2016], such as DFS and *preemption bounding* [Musuvathi and Qadeer 2007], that performed very poorly on our benchmarks compared to all others, so we leave them out of the evaluation.

We measure the effectiveness of finding bugs using the metric $\texttt{Bugs}^{100}$ that we define next. For each program, and for each scheduler, we invoke the P#-Tester 100 times. Each invocation of the tester has a budget of exploring up to 10000 schedules, but it would stop as soon as it found a bug and move on to the next invocation. (Note that in the absence of any bug found, the experiment totals 1 million schedules per scheduler.) The metric $\texttt{Bugs}^{100}$ is the number of times a bug was exposed out of the 100 invocations. The metric captures the most important aspect of a user's experience when testing their code: $\texttt{Bugs}^{100}$ is the chance (in percentage) that an invocation of P#-Tester would find a bug when there is one.

The exact upper bound on the length of each schedule (in the number of scheduling decisions) was varied per program, and was typically in the order of 10^3. We ran all experiments on a Windows 10 machine configured with 8 Intel Xeon cores and 64GB of RAM.

6.1 Benchmarks

We consider three different categories of benchmarks: complex *protocols* that are publicly available as part of the P# framework, *multithreaded* benchmarks from SvComp [Beyer 2019] and SCTBench [Thomson et al. 2016] benchmark suites, and *production* distributed services from Microsoft Azure. All of the benchmarks are C# programs already designed to use P#, except for SvComp and SCTBench benchmarks that we had to manually port. In all cases, this exercise was straightforward. Most of the SvComp and SCTBench benchmarks contained trivial bugs, with all the schedulers reporting large $\texttt{Bugs}^{100}$ values of over 50. We report here the results for the rest of the benchmarks.

We briefly describe our benchmarks including the custom hashing scheme that we implemented for them (for use with $\texttt{QL}^{\texttt{c}}$). Our choice of the hash was *not* guided by knowledge of the bug in the program, but rather by selecting the components of the program that were central to its logic.

Protocols. Raft has been explained already in Section 2. Chord is a protocol for a peer-to-peer distributed hash table. It maintains a collection of keys; their associated values are distributed among multiple nodes. A bug occurs if a client attempts to retrieve a key, which should be present, but gets a response that it does not exist. We hash the set of keys stored in each node.

FailureDetector is a protocol for detecting node failures. It contains a heartbeat mechanism, where a manager initiates rounds where each node is expected to send heartbeats. If a node fails to send a heartbeat within a period of time, it is deemed to have failed in that round. The custom hash includes the set of alive nodes, and the set of nodes that are still deemed alive in the current round. A bug occurs if three rounds elapse and a node stays failed.

Paxos is a consensus protocol. A bug occurs if two different values are agreed upon simultaneously. We hash the set of proposed, accepted and learned values.

Multithreaded programs. The variants of Fib-Bench and Triangular benchmarks involve multiple tasks performing integer arithmetic over shared variables, and with an assertion over the value of the shared variables after all threads finish. For instance, the assertion in Fib-Bench requires the computation among the tasks to finally result in the n-th Fibonacci number.

SafeStack is an implementation of a lock-free stack [Thomson et al. 2016; Vyukov 2010]. The bug is challenging, requiring at least 5 preemptions, and was missed by all the techniques in a prior evaluation [Thomson et al. 2016]. Our custom hashing scheme only considers the contents of the stack at any given point of time.

BoundedBuffer is an adaptation of a popular example [Cargill 2009], consisting of a fixed-size buffer being simultaneously updated by producers and consumers. The bug in this code turned out to be extremely challenging and evaded detection until revealed by the original author [Wayne 2018]. Our implementation is a C# port of the original Java version of the program. Our custom hashing scheme only considers the contents of the buffer at any given point of time.

Production distributed services. These are deployed cloud services of Microsoft Azure. The services are highly concurrent, serving over *hundreds of thousands* of VMs to users of Azure. Their tests check for functional correctness as well as tolerance to failures, commonly called *failover* testing, that is often a central concern for distributed services. One of the services, including its design in P#, has been described in detail in previous work [Deligiannis et al. 2020].

We used three different services, sometimes taking multiple versions of the services. These services were designed by the product teams while routinely using P#-Tester during the development process. In some cases, running P#-Tester was mandated with every check-in to the code repository. Occasionally, the product team reported a snapshot of their code to an internal P# emailing list (before squashing their commits) when they managed to find and fix a "tricky" bug in their code. We included these snapshots in our evaluation. We, however, did not have knowledge of the bug before running P#-Tester ourselves. An important note is that the released version of P# does not yet include QL. Bugs in these services were found by existing P#-Tester schedulers, mostly PCT. In that sense, these bugs are biased towards PCT because we know that PCT already finds them in reasonable time. Nonetheless, as our results will show, QL still manages to outperform PCT.

The selected set of benchmarks span a wide range of programs, covering a wide spectrum of bug patterns. All the benchmarks have a single known bug (to the best of our knowledge).

6.2 Results

Bug-finding ability. Table 2 compares the performance of the $\texttt{QL}^{\texttt{d}}$ scheduler, on the $\texttt{Bugs}^{100}$ metric, against the other schedulers. For PCT, we used a priority-switch bound of 3 because lower bounds performed worse. Interestingly, we found that higher bounds sometimes performed better (theoretically, lower bounds have a better chance in the worst-case), so we report bounds of 10 and 30 as well. However, in all production benchmarks, PCT regressed on $\texttt{Bugs}^{100}$ with bounds higher than 10.

Our main conclusions from Table 2 are as follows. $\texttt{QL}^{\texttt{d}}$ was the *only* scheduler to find bugs in all the benchmarks. Greedy was the next best with bugs found in 12 out of the 16 benchmarks. Further, the $\texttt{Bugs}^{100}$ value for $\texttt{QL}^{\texttt{d}}$ was consistently among the highest (for 10 out of the 16 applications). In Raft-v2, $\texttt{QL}^{\texttt{d}}$ had $\texttt{Bugs}^{100}$ of 95, whereas all other schedulers had $\texttt{Bugs}^{100}$ of less than 5. For Chord, $\texttt{QL}^{\texttt{d}}$ is the *only* scheduler that exposed the bug.

The performances of PCT and IDB are very sensitive to the choice of parameters such as bounds on priority-change points and maximum schedule length. As an example, in the FailureDetector

Table 2. Results from applying various schedulers on our benchmarks. The reported statistics are: lines of code (LoC), maximum number of concurrent workers (#T) and the $\texttt{Bugs}^{100}$ metric (higher is better) for various schedulers. The last row gives the geometric mean of the $\texttt{Bugs}^{100}$ metric for each scheduler (only over non-zero values), as well as the number of bugs found (out of 16).

				$\texttt{Bugs}^{100}$							
	Benchmarks	**LoC**	**#T**	**QL^{b}**	**QL^{d}**	**Random**	**Greedy**	**PCT-3**	**PCT-10**	**PCT-30**	**IDB**
Protocols	Raft-v1	1194	17	99	99	**100**	83	✗	12	45	28
	Raft-v2	1194	17	28	**95**	4	3	✗	✗	✗	1
	Paxos	849	10	16	66	8	20	19	91	**92**	33
	Chord	859	7	✗	**34**	✗	✗	✗	✗	✗	✗
	FailureDetector	692	5	✗	99	✗	✗	11	**100**	99	31
Multithreaded	Fib-Bench-2	55	3	**100**	**100**	**100**	**100**	✗	82	**100**	**100**
	Fib-Bench-Longest-2	55	3	99	**100**	**100**	**100**	✗	✗	✗	**100**
	Triangular-2	73	3	85	**100**	86	**100**	✗	✗	2	70
	Triangular-Longest-2	73	3	✗	**100**	✗	79	✗	✗	✗	✗
	SafeStack	253	6	32	1	43	23	✗	✗	21	**46**
	BoundedBuffer	284	9	✗	**53**	12	36	✗	✗	✗	✗
Production	CacheProvider	56649	27	74	**79**	14	24	37	29	25	23
	PoolManager-v1	33827	15	✗	**100**	✗	✗	**100**	**100**	37	✗
	PoolManager-v2	33827	28	**100**	97	✗	✗	**100**	36	✗	✗
	ResourceProvider-v1	18663	17	✗	92	**100**	16	76	96	80	✗
	ResourceProvider-v2	19771	17	✗	**100**	**100**	10	64	**100**	90	✗
	G-Mean $\texttt{Bugs}^{100}$			$59.1_{(9)}$	$\mathbf{63.9}_{(16)}$	$37.4_{(11)}$	$32.2_{(12)}$	$45.0_{(7)}$	$59.2_{(9)}$	$40.4_{(10)}$	$30.3_{(9)}$

benchmark, PCT-30 performs better than PCT-3, but it is the other way around for PoolManager-v1. We also found that the performance of PCT deteriorated as we increased the maximum schedule length parameter of P#-Tester. It is difficult for a user to make a good educated guess of these parameters a priori. QL is robust to such parameters, adding to its appeal for practical use.

The performance of PCT on production benchmarks was usually good. This was to be expected because these bugs were first caught using PCT by the product teams. Even then, we find that QL outperforms PCT with significantly higher $\texttt{Bugs}^{100}$ values.

Lastly, we find that exploration strategies based on state observations seem to perform well in general, even without learning. For example, the performance of the Greedy scheduler is competitive compared to PCT-3. Thus, clearly, making state observations is useful for exploration of concurrent behaviors.

The extra component of adaptive learning in QL provides it with an additional edge over Greedy. We believe that Greedy often suffered because of its deterministic nature of always picking the least visited state, whereas QL can make other choices with a smaller, but non-zero probability. Furthermore, QL also accounts for actions that are bad from an exploration point of view. For example, periodic events (such as timers) can trivially result in new states. Greedy may repeatedly fire these periodic events, filling up inboxes, but disallowing it from exploring new concurrent behaviors. However, the negative reward associated with an action, as well as the tendency to make randomized decisions, allows QL to break from this loop.

Figure 8 reports the mean number of iterations required by P#-Tester to find a bug over the protocol benchmarks. The general trend is similar to the $\texttt{Bugs}^{100}$metric reported in Table 2, however, for benchmarks such as Raft, the relative gap is even more favorable for QL.

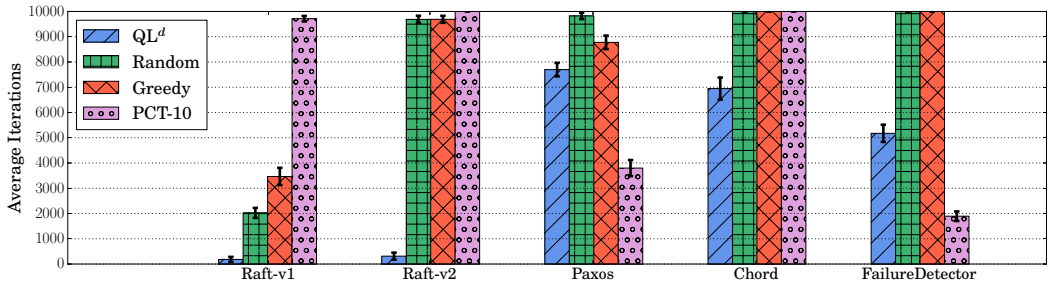

Fig. 8. Mean iterations and standard error (both lower the better) for the various schedulers on the protocols.

Effect of optimizations. The performance of $\texttt{QL}^{\texttt{b}}$ in Table 2 shows the effect of the optimizations Opt-1 and Opt-2 (outlined in Section 5) in $\texttt{QL}^{\texttt{d}}$. Note that Opt-2 plays a role only in the PoolManager and ResourceProvider production benchmarks, since these are the only applications that do failover testing.

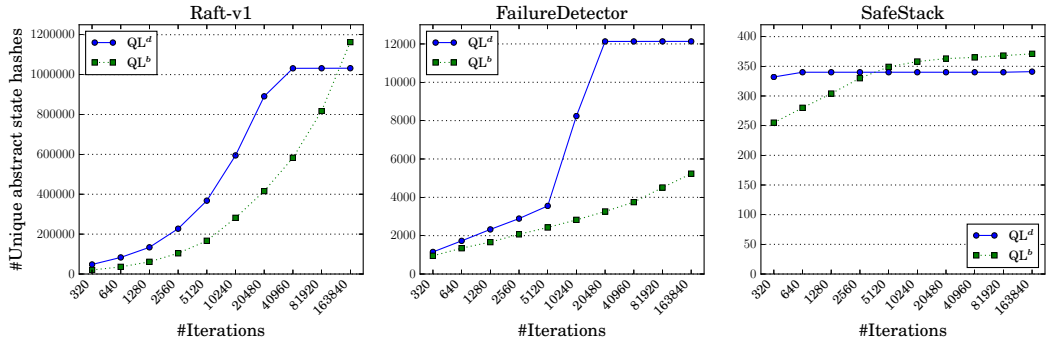

Fig. 9. Comparing coverage for QL with optimizations ($\texttt{QL}^{\texttt{d}}$) and with all optimizations turned off ($\texttt{QL}^{\texttt{b}}$).

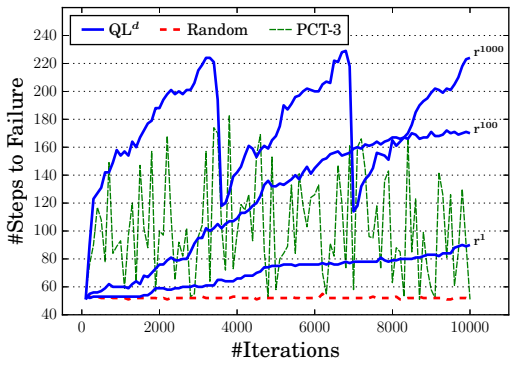

Fig. 10. Steps to failure injection across iterations. r^x denotes a penalty of x used in QL.

Opt-1: Compared to $\texttt{QL}^{\texttt{d}}$, the $\texttt{QL}^{\texttt{b}}$ scheduler regresses in 12 out of the 16 benchmarks. To investigate the reason behind this regression, we compared the coverage achieved by $\texttt{QL}^{\texttt{b}}$ against $\texttt{QL}^{\texttt{d}}$ for

a large number of iterations. Figure 9 summarizes our findings for a subset of the benchmarks. $\texttt{QL}^{\texttt{d}}$ explores a good percentage of the total discovered states quickly (within 10000 iterations). The $\texttt{QL}^{\texttt{b}}$ scheduler does achieves similar coverage but at a slower pace. For Raft-v1, $\texttt{QL}^{\texttt{b}}$ requires nearly 10× more iterations to match the coverage of $\texttt{QL}^{\texttt{d}}$, while this number is even higher for FailureDetector. For SafeStack, since the total number of states is low, $\texttt{QL}^{\texttt{b}}$ is able to explore the state space better. Overall, Opt-1 helps accelerate coverage.

Opt-2: We investigated the effect this optimization on the PoolManager benchmark. We measured the number of steps after which a scheduler injected the failure message across different iterations. Figure 10 summarizes our findings. The bugs in PoolManager manifests only if the whole system reaches a particular configuration (after a complex set of interactions between the constituent state machines) and then a failure message is sent. Due to the design of the test, if the failure message is injected too early, the application performs an early failover followed by a period of little activity. From Figure 10, we see that the `Random` scheduler injects the failure message early across all its iterations, thereby causing it to discover very few states (see Figure 11). In contrast, PCT-3 injects the failure message at different points in the execution, allowing it to discover more states (and, consequently, perform better on the $\texttt{Bugs}^{100}$ metric). With the standard penalty of -1, `QL` learns to inject the failure message at progressively later points in an execution, but the learning rate is slow. A 100× increase in the penalty increases the learning rate. A penalty of 1000 (which is Opt-2) accelerates learning even further. The failure message is injected progressively later until it reaches the end of the test, at which point it tries injecting early again and the process iterates.

Table 3. Comparing $\texttt{QL}^{\texttt{d}}$ without and with the handling of data non-determinism.

	$\texttt{Bugs}^{100}$	
Benchmarks	$\texttt{QL}^{\texttt{d}}$-NDN	$\texttt{QL}^{\texttt{d}}$
Raft-v1	86	99
Raft-v2	1	95
Paxos	19	66
Chord	✗	34
CacheProvider	22	79

Effect of handling data non-determinism in `QL`. As we highlighted in earlier sections, `QL` is the first CCT scheduler that accounts for data non-determinism in a way other than resolving it uniformly-at-random. We evaluate its benefit in Table 3. We use $\texttt{QL}^{\texttt{d}}$-NDN to denote a version of $\texttt{QL}^{\texttt{d}}$ where we turn off the handling of data non-determinism (i.e., resolve it uniformly-at-random). Table 3 highlights that $\texttt{QL}^{\texttt{d}}$-NDN clearly regresses in its ability to expose bugs. In fact, these results, taken together with those in Table 2, show that for benchmarks such as Raft-v2 and Chord, the superior performance of $\texttt{QL}^{\texttt{d}}$ over all other schedulers can be attributed to its handling of data non-determinism.

Inspecting these benchmarks, it was clear that data non-determinism plays a central role in triggering the bug. For example, in `Raft`, data non-determinism is used to control the firing of timer events that starts rounds of leader election. It is important for a timer to fire at the right time for the bug to manifest. In `Chord`, data non-determinism is used to pick an arbitrary node for termination. The terminated node must have been the one storing the key that is queried later by the test. The case for CacheProvider was already explained in Section 2.

Coverage achieved by `QL` *during exploration.* We also measured the number of unique abstract states covered by each of the schedulers across all iterations of the same run of P#-Tester. Figure 11

summarizes our findings for a subset of the benchmarks; these results are representative of our findings for the remaining benchmarks as well.

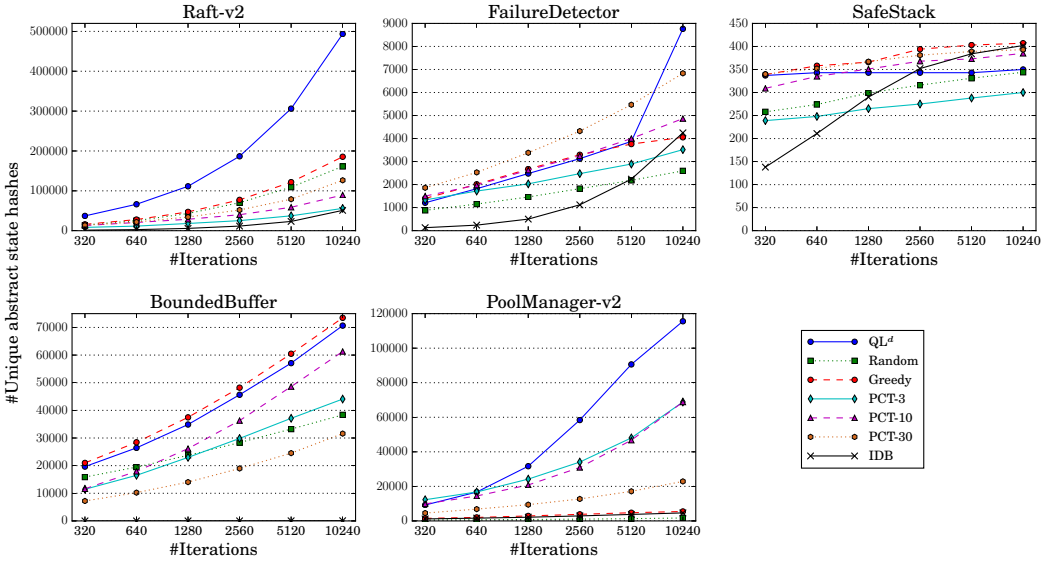

Fig. 11. Number of unique abstract states explored by various schedulers.

Coverage is not a perfect indicator for bug-finding ability. In general, because actual state space of a program can be very large, the *type* of behaviors covered matter as much as the number of behaviors covered. Nonetheless, the experiment reveals a strong signal that contributes to the effectiveness of QL in finding bugs. In many cases, QL achieves significantly better coverage (of abstract states) compared to others, although Greedy outperforms it for SafeStack and BoundedBuffer. The statistics for SafeStack show overall a very small number of abstract states (for all schedulers), perhaps because the default abstraction was too coarse. Thus, QL is not able to learn effectively and that likely contributes to its low Bugs^{100} value. We come back to this example in the next subsection when we vary the abstraction.

Effect of state abstraction on QL. We investigate the effect of using the different state abstractions with QL. Table 4 summarizes our findings. For the protocols, QL^{i}, which tracks only the contents of the inbox of the P# machines, sufficed to expose all bugs besides Chord, although QL^{d} performed better.

Hashing additional local state, as allowed by QL^{c} may actually regress the Bugs^{100} metric as can be seen for the Paxos and FailureDetector protocols in Table 4. A notable exception is SafeStack, where the QL^{d} abstraction was too coarse to expose the bug. Our custom abstraction involved tracking the exact contents of the stack, which forced QL^{c} to explore different stack contents thereby exposing the bug much more frequently. Once again, this custom abstraction involved tracking a component which is central to the logic of the program, and is *not* based on our knowledge of the bug.

Performance overhead of QL. It is common to use CCT with a fixed iteration budget as opposed to a fixed time budget. We also found this to be the typical usage scenario among P# users, which is why the experiment of Table 2 used a $10K$ iteration budget for each run of P#-Tester. This is

Table 4. Effect of using different state abstractions on QL.

	Bugs^{100}		
Benchmarks	**QL^{i}**	**QL^{d}**	**QL^{c}**
Raft-v1	100	99	100
Raft-v2	100	95	100
Paxos	30	66	33
Chord	✗	34	2
FailureDetector	38	99	78
SafeStack	n/a	1	31

useful because: (*i*) it makes the testing independent of the machine configuration and the current CPU load (with a fixed time budget, the amount of testing will depend on how fast is the machine). Also, (*ii*) it makes the testing independent of the length of each iteration, and (*iii*) the testing experience for two different developers will be the same, regardless of their machine configurations. Nonetheless, we do note that QL, in general, is *slower* than schedulers like Random or PCT that hardly do any computation to decide the next worker to execute.

Table 5 summarizes the time taken by P#-Tester to run $10K$ iterations under the various schedulers. We notice that QL makes testing around $1.49\times$ to $1.84\times$ slower compared to other schedulers. However, the overall test experience remains about the same because QL compensates for the slower testing with faster increase in coverage. Table 6 repeats the Bugs^{100} experiment with a fixed time budget of 2 minutes per invocation of P#-Tester, over a subset of the benchmarks. The overall trend displayed in the table remains the same as with a fixed-iteration budget.

Table 5. Running times (in seconds) for a single invocation of P#-Tester with a $10K$ iteration budget.

		Running Time (seconds)						
	Benchmarks	**QL^{d}**	**Random**	**Greedy**	**PCT-3**	**PCT-10**	**PCT-30**	**IDB**
Protocols	Raft-v1	292	178	237	177	171	146	164
	Raft-v2	276	187	153	126	130	151	181
	Paxos	164	126	161	223	120	111	74
	Chord	133	45	46	46	47	48	44
	FailureDetector	153	133	148	157	164	165	133
	G-Mean ×-slowdown		1.61×	1.47×	1.49×	1.67×	1.69×	1.84×

Dependence on the α and γ parameters . Two key parameters in the Q-Learning algorithm are α (learning rate) and γ (discount factor) parameters used in Equation 2. We evaluated the protocols for different combinations of α and γ, and we did not find a significant variation in the Bugs^{100} value due to the choices. We settled for $\alpha = 0.3$ and $\gamma = 0.7$ since it strikes a reasonable balance in Equation 2 between weighing immediate versus future rewards. While the default values of α and γ worked well for all our benchmarks, we do not claim that these values will generalize for all programs. The α and γ are referred to as *hyperparameters*, and a large body of work exists on hyperparameter tuning (see, for example, Fernandez and Caarls [2018] and Eriksson et al. [2003]).

Table 6. Bugs^{100} metric for the schedulers on the protocols and multithreaded benchmarks, with each invocation of P#-Tester having a 2-minute time-budget.

		Bugs^{100}						
	Benchmarks	**QL^d**	**Random**	**Greedy**	**PCT-3**	**PCT-10**	**PCT-30**	**IDB**
Protocols	Raft-v1	95	**99**	80	1	10	44	35
	Raft-v2	**91**	7	5	✗	✗	1	✗
	Paxos	65	8	19	11	93	**94**	25
	Chord	**44**	✗	✗	✗	✗	✗	✗
	FailureDetector	98	✗	1	12	**100**	99	34
Multithreaded	Fib-Bench-2	**100**	**100**	**100**	✗	**100**	**100**	**100**
	Fib-Bench-Longest-2	**100**	**100**	**100**	✗	✗	9	**100**
	Triangular-2	**100**	93	**100**	✗	✗	✗	77
	Triangular-Longest-2	**99**	✗	86	✗	✗	✗	✗
	SafeStack	2	31	26	✗	✗	15	**59**
	BoundedBuffer	**100**	11	89	✗	✗	✗	✗
	G-Mean Bugs^{100}	**61.5**$_{(11)}$	33.9$_{(8)}$	32.9$_{(10)}$	5.1$_{(3)}$	55.2$_{(4)}$	24.7$_{(7)}$	54.1$_{(7)}$

7 RELATED WORK

Controlled Concurrency Testing. Controlled concurrency testing has been the subject of extensive research, given the elusive nature of concurrency bugs and the lack of practical techniques to find them. Several tools have come out of this research; we only list a few of them here. VeriSoft [Godefroid 2005] popularized the use stateless model-checking algorithm on real software (as opposed to models of software). The Chess tool [Musuvathi and Qadeer 2008; Musuvathi et al. 2008] showed the effectiveness of iterative context-bounding, and later delay-bounding [Emmi et al. 2011], for finding bugs in multi-threaded software. These were extended and implemented in Zing for asynchronous (message-passing) systems [Desai et al. 2015]. The Cuzz tool introduced and implemented PCT [Burckhardt et al. 2010] to showcase the power of randomized testing. Several tools also exist for distributed systems, including dBug [Simsa et al. 2011], MoDist [Yang et al. 2009], and Samc [Leesatapornwongsa et al. 2014]. The presence of such a large number of tools clearly indicates the importance of CCT.

Implementing CCT for real systems naturally requires considerable engineering to ensure that all concurrency in the system can be controlled by the tool. Our concern in this paper is orthogonal to this engineering. We instead focus on the search algorithm, called a scheduling strategy in this paper. While we only used P# as the implementation vehicle, the hope is that our strategies can be adopted by any CCT tool. [Thomson et al. 2016] provides a nice survey and empirical comparison of several stateless strategies in a common framework and we also describe some of them in Section 3.

Most of the schedulers considered in this paper are *randomized*, i.e., they rely on randomness to explore the state space of the program. Systematic exploration, on the other hand, guarantees to explore all interleavings of the program in the limit. Systematic techniques often rely on a DFS-style of exploration, aided by dynamic partial-order reduction (DPOR) [Flanagan and Godefroid 2005] that skips an interleaving if an *equivalent* interleaving [Mazurkiewicz 1986] was already covered. A straightforward implementation of DFS+DPOR in P# performed quite poorly compared to other schedulers and was subsequently dropped out of support. (Among the requirements of a CCT framework, mentioned in the beginning of Section 5, implementing DPOR further requires capturing *dependency* among the actions in order to construct the partial-order representation of a trace. This imposes an additional burden on the CCT framework that other schedulers do not require.) However, more recent work has shown considerable improvements over DPOR by more aggressively pruning the space of interleavings [Chalupa et al. 2018; Huang 2015; Huang and

Huang 2017]. It would be interesting future work to compare these techniques against randomized strategies in terms of coverage or bug-finding ability.

Learning based Software Testing. Learning algorithms have been applied to the problem of *fuzzing* program inputs [Böttinger et al. 2018; Godefroid et al. 2017; Patil and Kanade 2018; She et al. 2018]. In particular, Böttinger et al. [Böttinger et al. 2018] formalizes input fuzzing as an RL problem, and applies deep Q-learning to learn mutations that yield new program inputs that are likely to maximize code coverage. Zheng et al. [Zheng et al. 2003, 2006] applies supervised learning techniques to automatically identify program features or instrumentation predicates which are likely to trigger a bug, based on a data set comprising user reports of program executions. Mariani et al. [Mariani et al. 2012] leverages Q-learning to generate test cases for testing GUI-based applications. The problem of input fuzzing is orthogonal to CCT: the former is about input values but the latter is primarily about controlling scheduling decisions.

Veanes et al. [Veanes et al. 2006] formalizes the problem of *conformance checking* between a model and an implementation as a Markov Decision Process, and uses heuristics inspired from reinforcement learning to solve the problem. Unlike our work, the paper assumes limited communication between concurrently executing agents, and also assumes the agents' actions being deterministic. Moreover, their implementation handles a small toy example, whereas our implementation can scale to production code.

Baskiotis et al. [Baskiotis et al. 2007] aims to maximize path coverage in sequential programs by identifying distinct feasible paths in the control flow graph with high probability, using an adaptive sampling mechanism. In contrast, our QL scheduler can handle concurrent programs and does not require an explicit control flow graph representation.

8 LIMITATIONS AND FUTURE WORK

The performance of QL crucially depends on the state abstraction being used. While we showed in Section 6 that the default state abstraction provided by P# sufficed in our experiments for a range of benchmarks, we do not claim that this generalizes to other settings. Further experimentation would be required, for instance with custom state abstractions, but the default one should likely be used as a baseline. Similarly, the parameters α and γ may also need to be tuned for a particular application, in general.

An interesting direction to explore would be the coupling of QL with static analyses to guide the selection of the best state abstraction. Evaluating the performance of other exploration techniques from the reinforcement learning literature (such as SARSA [Rummery and Niranjan 1994]) and other policies (such as ϵ-greedy) is also interesting future work.

9 CONCLUSION

In this paper, we proposed a controlled concurrency testing (CCT) scheduler, called QL, which leverages Q-Learning to explore a user-defined abstraction of a program's state space. QL is geared towards maximizing coverage and adapts to the application under test. It is effective at finding concurrency bugs without using any pre-defined biases towards certain bug patterns. QL is also the first scheduler that accounts for data non-determinism. We implemented QL in an open-source industrial-strength CCT framework. In our benchmarks, comprising complex protocols and production cloud services, we showed that QL outperforms state-of-the-art CCT strategies.

REFERENCES

Akka Raft. 2015. Leader election bug in Akka Raft implementation. https://github.com/ktoso/akka-raft/issues/45.

Amazon. 2012. Summary of the AWS service event in the US East Region. http://aws.amazon.com/message/67457/.

Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. 2004. Zing: A Model Checker for Concurrent Software. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. 484–487.

Andrew G Barto and Satinder Pal Singh. 1991. On the computational economics of reinforcement learning. In *Connectionist Models*. Elsevier, 35–44.

Nicolas Baskiotis, Michèle Sebag, Marie-Claude Gaudel, and Sandrine Gouraud. 2007. A machine learning approach for statistical software testing. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. Morgan Kaufmann Publishers Inc., 2274–2279.

Richard Bellman et al. 1954. The theory of dynamic programming. *Bull. Amer. Math. Soc.* 60, 6 (1954), 503–515.

Dirk Beyer. 2019. Automatic Verification of C and Java Programs: SV-COMP 2019. In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 11429)*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer, 133–155. https://doi.org/10.1007/978-3-030-17502-3_9

Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep Reinforcement Fuzzing. In *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*. IEEE Computer Society, 116–122. https://doi.org/10.1109/SPW.2018.00026

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, James C. Hoe and Vikram S. Adve (Eds.). ACM, 167–178. https://doi.org/10.1145/1736020.1736040

Qingpeng Cai, Aris Filos-Ratsikas, Pingzhong Tang, and Yiwei Zhang. 2018. Reinforcement mechanism design for fraudulent behaviour in e-commerce. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Tom Cargill. 2009. Extreme Programming Challenge Fourteen. http://wiki.c2.com/?ExtremeProgrammingChallengeFourteen.

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2018. Data-centric dynamic partial order reduction. *Proc. ACM Program. Lang.* 2, POPL (2018), 31:1–31:30.

Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. 2000. NUSMV: A New Symbolic Model Checker. *STTT* 2, 4 (2000), 410–425.

Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vasiliki Hartonas-Garmhausen. 1996. Symbolic Model Checking. In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*. 419–427.

Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. 2015. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 154–164. https://doi.org/10.1145/2737924.2737996

Pantazis Deligiannis, Narayanan Ganapathy, Akash Lal, and Shaz Qadeer. 2020. Building Reliable Cloud Services Using P# (Experience Report). *ArXiv* abs/2002.04903 (2020).

Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. 2016. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.*, Angela Demke Brown and Florentina I. Popovici (Eds.). USENIX Association, 249–262. https://www.usenix.org/conference/fast16/technical-sessions/presentation/deligiannis

Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. 2015. Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 73–83. https://doi.org/10.1145/2786805.2786861

Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 411–422. https://doi.org/10.1145/1926385.1926432

Anders Eriksson, Genci Capi, and Kenji Doya. 2003. Evolution of meta-parameters in reinforcement learning algorithm. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, Vol. 1. IEEE, 412–417. https://ieeexplore.ieee.org/document/1250664

Franklin Cardeñoso Fernandez and Wouter Caarls. 2018. Parameters tuning and optimization for reinforcement learning algorithms using evolutionary computing. In *2018 International Conference on Information Systems and Computer Science*

(INCISCOS). IEEE, 301–305. https://ieeexplore.ieee.org/document/8564542

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. 110–121.

Patrice Godefroid. 2005. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design* 26, 2 (2005), 77–101. https://doi.org/10.1007/s10703-005-1489-x

Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 50–59. https://doi.org/10.1109/ASE.2017.8115618

Jim Gray. 1986. Why do computers stop and what can be done about it?. In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*. IEEE, 3–12.

Verena Heidrich-Meisner, Martin Lauer, Christian Igel, and Martin A Riedmiller. 2007. Reinforcement learning in a nutshell.. In *ESANN*. Citeseer, 277–288.

Gerard Holzmann. 2011. *The SPIN Model Checker: Primer and Reference Manual* (1st ed.). Addison-Wesley Professional.

Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 165–174.

Shiyou Huang and Jeff Huang. 2017. Speeding Up Maximal Causality Reduction with Static Dependency Analysis. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. 16:1–16:22.

Harshad Khadilkar. 2018. A Scalable Reinforcement Learning Algorithm for Scheduling Railway Lines. *IEEE Transactions on Intelligent Transportation Systems* 20, 2 (2018), 727–736.

Jens Kober, J Andrew Bagnell, and Jan Peters. 2013. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research* 32, 11 (2013), 1238–1274.

Sascha Lange, Martin Riedmiller, and Arne Voigtländer. 2012. Autonomous reinforcement learning on raw visual input data in a real world application. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 399–414.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. 2016. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research* 17, 1 (2016), 1334–1373.

Dong Li, Dongbin Zhao, Qichao Zhang, and Yaran Chen. 2019. Reinforcement Learning and Deep Learning Based Lateral Control for Autonomous Driving [Application Notes]. *IEEE Computational Intelligence Magazine* 14, 2 (2019), 83–98.

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721.

Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2012. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche (Eds.). IEEE Computer Society, 81–90. https://doi.org/10.1109/ICST.2012.88

Antoni W. Mazurkiewicz. 1986. Trace Theory. In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*. 279–324.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533. https://doi.org/10.1038/nature14236

Rashmi Mudduluru, Pantazis Deligiannis, Ankush Desai, Akash Lal, and Shaz Qadeer. 2017. Lasso detection using partial-state caching. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 84–91. https://doi.org/10.23919/FMCAD.2017.8102245

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. 446–455.

Madanlal Musuvathi and Shaz Qadeer. 2008. Fair stateless model checking. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 362–371. https://doi.org/10.1145/1375581.1375625

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert

van Renesse (Eds.). USENIX Association, 267–280. http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf

Emre O Neftci and Bruno B Averbeck. 2019. Reinforcement learning in artificial and biological systems. *Nature Machine Intelligence* 1, 3 (2019), 133–143.

Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, Russ Tedrake, and John C Duchi. 2018. Scalable end-to-end autonomous vehicle testing via rare-event simulation. In *Advances in Neural Information Processing Systems*. 9827–9838.

Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*. USENIX, 305–319.

Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *PACMPL* 2, OOPSLA (2018), 160:1–160:28.

P# Team. 2019. P#: A framework for rapid development of reliable asynchronous software. https://github.com/p-org/PSharp.

Ketan Patil and Aditya Kanade. 2018. Greybox fuzzing as a contextual bandits problem. *CoRR* abs/1806.03806 (2018). arXiv:1806.03806 http://arxiv.org/abs/1806.03806

Jing Peng and Ronald J Williams. 1996. Incremental multi-step Q-learning. *Machine Learning* 22, 1-3 (1996), 283–290.

Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering.

Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. 2003. *Artificial intelligence: a modern approach*. Vol. 2. Prentice hall Upper Saddle River.

Mitsuo Sato, Kenichi Abe, and Hiroshi Takeda. 1988. Learning control of finite Markov chains with an explicit trade-off between estimation and control. *IEEE transactions on systems, man, and cybernetics* 18, 5 (1988), 677–684.

Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2018. NEUZZ: Efficient Fuzzing with Neural Program Learning. *CoRR* abs/1807.05620 (2018). arXiv:1807.05620 http://arxiv.org/abs/1807.05620

Jing-Cheng Shi, Yang Yu, Qing Da, Shi-Yong Chen, and An-Xiang Zeng. 2019. Virtual-taobao: Virtualizing real-world online retail environment for reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4902–4909.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489. https://doi.org/10.1038/nature16961

Jirí Simsa, Randy Bryant, and Garth A. Gibson. 2011. dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6823)*, Alex Groce and Madanlal Musuvathi (Eds.). Springer, 188–193. https://doi.org/10.1007/978-3-642-22306-8_14

Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. MIT press.

Csaba Szepesvári. 2010. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning* 4, 1 (2010), 1–103.

Gregory Tassey. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, Planning Report 02-3* (2002).

Gerald Tesauro. 1991. Practical Issues in Temporal Difference Learning. In *Advances in Neural Information Processing Systems 4, [NIPS Conference, Denver, Colorado, USA, December 2-5, 1991]*, John E. Moody, Stephen Jose Hanson, and Richard Lippmann (Eds.). Morgan Kaufmann, 259–266. http://papers.nips.cc/paper/465-practical-issues-in-temporal-difference-learning

Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2016. Concurrency Testing Using Controlled Schedulers: An Empirical Study. *TOPC* 2, 4 (2016), 23:1–23:37. https://doi.org/10.1145/2858651

Ben Treynor. 2014. GoogleBlog – Today's outage for several Google services. http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html.

Margus Veanes, Pritam Roy, and Colin Campbell. 2006. Online Testing with Reinforcement Learning. In *Formal Approaches to Software Testing and Runtime Verification*, Klaus Havelund, Manuel Núñez, Grigore Roşu, and Burkhart Wolff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 240–253.

Dmitry Vyukov. 2010. Bug with a context switch bound 5. https://social.msdn.microsoft.com/Forums/en-US/91c1971c-519f-4ad2-816d-149e6b2fd916/bug-with-a-context-switch-bound-5?forum=chess.

Chris Watkins. 1989a. Models of Delayed Reinforcement Learning. In *PhD thesis, Psychology Department, Cambridge University*.

Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

Christopher John Cornish Hellaby Watkins. 1989b. Learning from delayed rewards. (1989).

Hillel Wayne. 2018. Augmenting Agile with Formal Methods. https://www.hillelwayne.com/post/augmenting-agile/.

Hua Wei, Guanjie Zheng, Huaxiu Yao, and Zhenhui Li. 2018. Intellilight: A reinforcement learning approach for intelligent traffic light control. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2496–2505.

S Whitehead. 1991a. *A Study of Cooperative Mechanisms for Faster Reinforcement Learning Univ. Rochester, Rochester.* Technical Report. NY, Tech. Rep. TR-365.

Steven D Whitehead. 1991b. Complexity and cooperation in Q-learning. In *Machine Learning Proceedings 1991*. Elsevier, 363–367.

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. 213–228.

Alice X. Zheng, Michael I. Jordan, Ben Liblit, and Alexander Aiken. 2003. Statistical Debugging of Sampled Programs. In *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf (Eds.). MIT Press, 603–610. http://papers.nips.cc/paper/2371-statistical-debugging-of-sampled-programs

Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006 (ACM International Conference Proceeding Series, Vol. 148)*, William W. Cohen and Andrew Moore (Eds.). ACM, 1105–1112. https://doi.org/10.1145/1143844.1143983

Zhenpeng Zhou, Xiaocheng Li, and Richard N Zare. 2017. Optimizing chemical reactions with deep reinforcement learning. *ACS central science* 3, 12 (2017), 1337–1344.