Uncovering Distributed System Bugs during Testing (not Production!)

Pantazis Deligiannis[1], John Erickson[2], Cheng Huang[3], Akash Lal[3]
Matt McCutchen[4], Shaz Qadeer[3], Wolfram Schulte[2], Paul Thomson[1]

[1]*Imperial College London,* [2]*Microsoft,* [3]*Microsoft Research*
[4]*Massachusetts Institute of Technology*

Abstract

Testing distributed systems is very challenging due to multiple sources of nondeterminism, such as arbitrary interleavings between event handlers and unexpected node failures. Stress testing, commonly used in industry today, is unable to deal with this kind of nondeterminism, which results in the most tricky bugs being missed during testing and only getting exposed in production.

We present a new methodology for systematically testing distributed systems. Our approach involves the use of P#, an extension of C# that combines a flexible environmental modeling approach with a concurrency testing framework, which can capture and systematically explore all sources of nondeterminism. We present two case studies of using P# to test production distributed systems inside Microsoft. Using P#, we managed to uncover a very subtle bug that was haunting developers for a long time, as they did not have an effective way to reproduce it. P# uncovered the bug in a very small setting, which made it easy to examine traces, identify and fix the problem. Pantazis: we now have many bugs in Matt's system too.

1 Introduction

Distributed systems are notoriously hard to design, implement and test [1, 11]. This is due to many well-known sources of *nondeterminism* [2], such as race conditions in the asynchronous interaction between system components, the use of multithreaded code inside a component, unexpected node failures, unreliable communication channels and data losses, and interaction with (human) clients. All these sources of nondeterminism translate into *exponentially* many execution paths that a distributed system might potentially execute. A bug might hide deep inside one of these paths and only manifest under extremely rare corner cases [4, 13].

Classic techniques that product groups employ to test, verify and validate their systems (such as code reviews, unit testing, stress testing and fault injection) are unable to capture and control all the aforementioned sources of nondeterminism, which causes the most tricky bugs being missed during testing and only getting exposed after a system has been put in production. Discovering and fixing bugs in production, though, is bad for business as it can cost a lot of money and many dissatisfied customers.

We interviewed engineers from the Microsoft Azure team regarding the top problems in distributed system development, and the unified response was that the most critical problem today is how to improve *testing coverage* to find bugs *before* a system goes in production. The need for better testing techniques is not specific to Microsoft; other companies, such as Amazon and Google, publicly acknowledge [14] that testing methodologies have to improve to be able to reason about the correctness of increasingly more complex distributed systems.

Amazon recently published an article [14] that describes their use of TLA+ [9] to detect distributed system bugs and prevent them from reaching production. TLA+ is a powerful specification language for verifying distributed protocols, but it is unable to verify the code that is actually being executed. The implied assumption is that a model of the system will be verified, and then the programmers are responsible to match what was verified with the source code of the real system. Although many design bugs can be caught with this approach, there is *no guarantee* that the real distributed system will be free of bugs.

In this work, our goal is to *test what is being executed*. We present a new methodology for testing legacy distributed systems and uncovering bugs before these systems are released in the wild. We achieve this using P# [3], an extension of the mainstream language C# that provides two key capabilities: (i) a flexible way of modeling the environment using simple language features; and (ii) a systematic concurrency testing framework that is able to capture and take control of all the nondeter-

minism in a real system (together with its modeled environment) and systematically explore execution paths to discover bugs.

We present three case studies of using P# to test production distributed systems for Windows Azure inside Microsoft: a distributed storage management system and a live migration protocol. Using P#, we managed to uncover a very subtle bug that was haunting developers for a long time as they did not have an effective way to reproduce the bug and nail down the culprit. P# uncovered this bug in a very small setting, which made it easy to examine traces, identify and eventually fix the problem.

To summarize, our contributions are as follows:

- We present a methodology that allows flexible modeling of the environment of a distributed system using simple language mechanisms.
- Our infrastructure can test production code written in C#, which is a mainstream language.
- We present three case studies of using P# to test production distributed systems, finding bugs that could not be found with traditional testing techniques.
- ...

2 Distributed Systems

Distributed systems typically consist of two or more components that communicate *asynchronously* by sending and receiving messages through a network layer [8]. Each component has its own input message queue, and when a message arrives, the component responds by executing an appropriate *message handler*. Such a handler consists of a sequence of program statements that might update the internal state of the component, send a message to another component in the system, or even create an entirely new component.

2.1 Challenges in testing

In a distributed system, message handlers can interleave in arbitrary order, because of the asynchronous nature of message-based communication. To complicate matters further, unexpected failures are the norm in production systems: nodes in a cluster might fail at any moment, and thus programmers have to implement sophisticated mechanisms that can deal with these failures and recover the state of the system. Moreover, with multicore machines having become a commodity, individual components of a distributed system are commonly implemented using multithreaded code, which adds another source of nondeterminism.

All the above sources of nondeterminism (as well as nondeterminism due to timeouts, message losses and client requests) can easily create *heisenbugs* [4, 13], which are corner-case bugs that are difficult to detect, diagnose and fix, without using advanced *asynchrony-aware* testing techniques. Techniques such as unit testing, integration testing and stress testing are heavily used in industry today for finding bugs in production code. However, these techniques are not effective for testing distributed systems, as they are not able to capture and control the many sources of nondeterminism.

The ideal testing technique should be able to work on unmodified distributed systems, capture and control all possible sources of nondeterminism, systematically inject faults in the right places, and explore all feasible execution paths. However, this is easier said than done when testing production systems.

A completely different approach for reasoning about the correctness of distributed systems is to use formal methods. A notable example is TLA+ [9], a formal specification language that can be used to design and verify concurrent programs via model checking. Amazon recently published an article describing their use of TLA+ in Amazon Web Services to verify distributed protocols [14]. A limitation of TLA+, as well as other similar specification languages, is that they are applied on a model of the system and not the actual system. Even if the model is verified, there is no guarantee that the code that will actually execute is free of bugs.

2.2 Types of bugs

We can classify most distributed system bugs in two categories: *safety* and *liveness* property violations [7].

Safety A safety property checks that an erroneous program state is *never* reached, and is satisfied if it *always* holds in each possible program execution.

Liveness A liveness property checks that some progress *will* happen, and is satisfied if it *always eventually* holds in each possible program execution.

A safety property can be specified using an *assertion* that fails if the property gets violated in some program state. An example of a generic safety property for message passing systems is to assert that whenever a message gets dequeued there must be an action that can handle the received message.

Liveness properties are much harder to specify and check since they apply over entire program executions and not just individual program states. Normally, liveness checking requires the identification of an infinite fair execution that never satisfies the liveness property [16, 12]. Prior work [16] has proposed that assuming

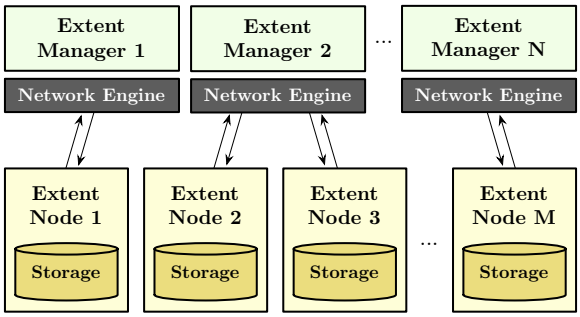

Figure 1: Top-level components of a distributed extent management system for Windows Azure.

a program with finite state space, a liveness property can be converted into a safety property. Other researchers proposed the use of heuristics and only exploring finite executions of an infinite state space system using random walks to identify if a liveness property is violated [6].

3 Case Studies in Microsoft

3.1 Azure Storage vNext

Azure Storage vNext is the next generation storage system for Windows Azure. Azure Storage vNext consists of multiple extent managers, extent nodes, and network engines that are able to send messages across the network, and enqueue any received messages in the input queue of the corresponding node (see Figure 1). Each extent manager is responsible for managing a subset of the extent nodes. Each extent node stores its corresponding extent in a local storage, and sends periodical heartbeats and synchronization messages to the extent manager. When the extent manager receives a synchronization message it is responsible to update extent nodes with the latest extent.

There are multiple *extent managers* (EMgrs) that are in charge of managing a subset of the extents. Each EMgr communicates asynchronously (via remote procedure calls) with a number of *extent nodes* (ENs) that store the extents. Each EN sends: (i) a *heartbeat* every 5 seconds, to notify the EMgr that it is still available; and (ii) a *sync report* every 5 minutes, to synchronize its extent with the rest of the nodes. The EMgr contains two data structures: an *extent center* (ECtr), which is updated every time an EN syncs; and an EN map, which is updated every time it receives a heartbeat from an EN. The EN map runs an EN expiration loop that is responsible for removing ENs that have expired from the EN map and also delete them from the ECtr. Finally, the EMgr runs an extent repair loop, which examines all contents of all extents in the ECtr and schedules repairs of extents if they are out-of-date (via a repair request).

The liveness property that must be always eventually satisfied in the Azure Storage vNext system is that a user-defined N number of extent nodes must be always eventually available with the latest extent. There was an actual bug in the system, that led this property to fail for some very rare executions. Although this buggy behavior would be observed from time to time during stress testing, there was no way to reproduce the bug. We discuss later in this paper, how we are able to detect and reproduce this bug using our approach.

Azure Storage vNext is very challenging to test. All the unit test and integration test suites of the Azure Storage vNext project successfully pass every single time. However, the developers found that the stress test suite, which constantly kills and launches extent nodes, could fail from time to time after very long executions. The observed failure was that the liveness property described in Section **??** would not get satisfied. The developers had no way to deterministically reproduce this bug or be able to detect what is the culprit, as the traces they were getting were very long and hard to parse.

3.2 Live Azure Table Migration

The Live Azure Table Migration is a library capable of transparently migrating a data set between tables in the Windows Azure storage service while an application is accessing the data set. MigratingTable provides a *virtual table* with an API similar to that of an ordinary Azure table, backed by a pair of *old* and *new* tables. A background *migrator* job moves all data from the old table to the new table. Meanwhile, each read or write issued to the virtual table is translated to a sequence of reads and writes on the backend tables according to a protocol we designed, which guarantees linearizability of operations on the virtual table across multiple application processes assuming that the backend tables respect their own linearizability guarantees.

The initial motivation for MigratingTable was to solve a scaling problem for Artifact Services, an internal Microsoft system with a data set that is sharded across tables in different Azure storage accounts because it exceeds the limit on traffic supported by a single Azure storage account. As the traffic continues to grow over time, we will need to reshard the data set across a greater number of Azure storage accounts without interrupting service. During such a resharding, our sharding manager will identify each key range that should migrate to a different table, and we will use a separate MigratingTable instance for each such key range to actually perform the migration (Figure 2). MigratingTable may also be useful to migrate data to a table with different values of configuration parameters that Azure does not support changing

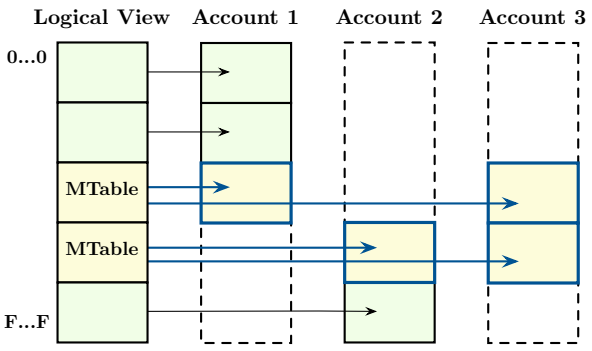

Figure 2: Resharding a data set when a third Azure storage account is added. Two key ranges are each migrated to the new account using a MigratingTable instance (abbreviated MTable).

on an existing table, such as geographic location.

3.3 Azure Service Fabric

Azure Service Faric (or *Fabric* for short) is a platform and API for creating reliable services that execute on a cluster of machines. The developer writes a service that receives requests (e.g. from some client program via HTTP) and mutates its state based on these requests. In order to make the service *reliable*, Fabric launches several copies (*replicas*) of the service, where each copy runs as a separate process on different nodes in the cluster. One replica is selected to be the *primary* which serves client requests; the rest are *secondaries*. The primary replicates state changes to the secondaries so that all replicas eventually have the same state. If the primary fails (e.g. if the node on which the primary is running crashes), Fabric elects one of the secondaries to be the new primary and launches another secondary; the new secondary will receive a full or partial copy (depending on whether persistent storage is used) of the state of the new primary in order to "catch up" with the other secondaries. Fabric provides a name-resolution service so that clients can always find the current primary. Fabric exposes several different APIs that developers can use to write services.

4 Testing production systems with P#

Our goal in this work is to *test what is being executed*. To achieve this we developed a new software engineering methodology for testing distributed systems, implemented on top of the P# framework. The original P# framework is explain in Section **??**, while the new approach is explained in Section **??**.

4.1 The P# framework

P# [3] provides an *event-driven asynchronous programming* language and a *systematic concurrency testing* engine for developing highly-reliable distributed systems.

The P# language is an extension of C#, built on top of Microsoft's Roslyn[1] compiler, that enables asynchronous programming using communicating state-machines. P# machines can interact asynchronously by sending and receiving events,[2] an approach commonly used to develop distributed systems. This programming model is similar to actor-based approaches provided by other asynchronous programming languages (e.g. Scala [15] and Erlang [18]).

A P# machine consists of an input event queue, states, state transitions, event handlers, fields and methods. Machines run concurrently with each other, each executing an event handling loop that dequeues an event from the input queue and handles it by invoking an appropriate event handler. This handler might update a field, create a new machine, or send an event to another machine. In P#, a send operation is non-blocking; the message is simply enqueued into the input queue of the target machine, and it is up to the operating system scheduler to decide when to dequeue an event and handle it. All this functionality is provided in a lightweight runtime library, build on top of Microsoft's Task Parallel Library [10].

Because P# is built on top of C#, the programmer can blend P# and C# code; this not only lowers the overhead of learning a new language, but also allows P# to easily integrate with legacy code. Another advantage is that the programmer can use the familiar programming and debugging environment of Visual Studio.

A key capability of the P# runtime is that it can run in *bug-finding mode*, where a embedded systematic testing engine captures and takes control of all sources of nondeterminism (such as event handler interleavings, failures, and client requests) in a P# program, and then systematically explores all possible executions to discover bugs.

P# is available as open-source[3] and is currently used by various teams in Microsoft to develop and test distributed protocols and systems.

Pantazis: Do we want to actually mention (and compare with) P in this paper?

4.2 Methodology

In previous work [3], we approached the problem of testing legacy distributed systems as follows. First, we ported the system to P#, then we modeled its environment as P# state machines, and finally we tested the

[1] `https://github.com/dotnet/roslyn`
[2] We use the word "event" and "message" interchangeably.
[3] `https://github.com/p-org/PSharp`

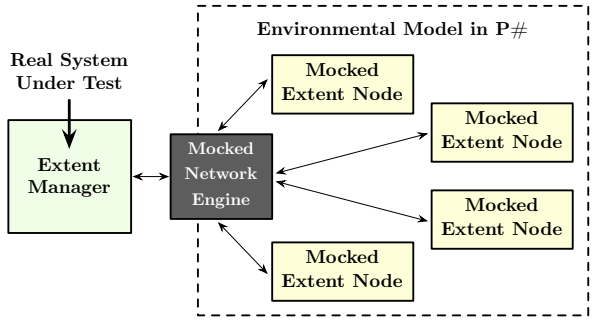

Figure 3: The real environment of the Extent Manager is replaced with a mocked version for testing.

ported system and its environmental model using the P# systematic concurrency testing engine. The limitation of this approach is that it does not allow us to directly test a legacy system, as it has to be re-implemented first in P#. However, such endeavor is very costly and time consuming, and thus is not realistic for testing an existing production system, such as the Azure Storage vNext. Also, unless the code under test is the one that will actually execute, there is no guarantee that the real system will be bug-free.

In order to solve this problem, and allow P# to be used for testing legacy distributed systems, we decided to take a radically different modeling approach. We provide the capability to model the environment of a system using P#, and then allow the developer to take advantage of existing language features, such as *method dispatch*, to connect the system under test with the environmental model, and finally test it using the P# systematic concurrency testing engine.

We argue that our approach is *flexible* since it allows the user to model *as much* or *as little* of the environment as required to achieve the desired level of testing. We also argue that our approach is *generic* since a programmer can build on top of it to test more complicated use cases (see Section **??**). Furthermore, the language features that are required to be used to connect the real code with the modeled code, are already being heavily used in production for testing purposes, which makes this method approachable to product groups.

4.3 Modeling the environment

Draft.

4.4 Using method dispatch for modeling

Method dispatch is the process of selecting which method, from a set of available methods with the same interface, should be invoked during a program's execution. There are two types of method dispatch: *static*, which is resolved during compilation; and *dynamic*, which is resolved in runtime. C# (and thus P#) supports both static and dynamic dispatch, and provide the `virtual` modifier that can be used to declare a method which can be *overridden* during runtime by an inheriting class. This capability is provided by the common language runtime (CLR) of Microsoft's .NET framework, and is a key feature of C# as well as other mainstream object-oriented languages.

Using method dispatch for modeling is straightforward. The system under test exposes a set of APIs as *virtual methods*. The developer can then *override* these APIs and replace them with *mocks* that will execute instead of the original implementations during systematic testing with P#.

We now give an example of using dynamic dispatch to model the network engine of an extent manager in the Azure Storage vNext case study (see Figure 4). The network engine is responsible for sending to and receiving messages from the various components of the system. During real execution, the network engine uses a custom remote procedure call (RPC) .NET library for communication. For testing, though, it is desirable to replace all calls to this RPC library with P# send and receive operations, which can be captured and systematically interleaved to find bugs. We easily achieved this by exposing the original send message operation of the network engine as a virtual method, and then overriding it for testing. In the overridden method, we created a P# event and then we wrapped the original message in this event's payload. Then, instead of invoking the RPC library, we invoke the `PSharpRuntime.Send(...)` method, which asynchronously sends the event (containing the original message) to the target extent node machine.

For mocking the receive operation, we take advantage of the implicit receive of events in P# machines. When a extent node machine receives an event, an appropriate event handler is invoked, which extracts the original message from the payload and then handles it accordingly.

Another important operation that we had to model in order to capture the nondeterminism in the Azure Storage vNext system, and be able to detect the liveness violation, was to model the timer operation that sends a heartbeat every 5 seconds and a sync message every 5 minutes.

4.5 Extensibility of the P# framework

Async/Await

custom schedulers, etc

Logs/traces -¿ user can extend them?

Pantazis: mention dependency injection pattern?

```
// Public interface of the real network engine
class NetworkEngine {
  public virtual void SendMessage();
  public virtual void EnqueueMessage();
}

// The mocked network engine used during testing
class MockedNetEngine : NetworkEngine {
  ExtentManager EM; // Handle to actual system under test
  MachineId Env; // Handle to modeled environment

  public MockedNetEngine(ExtentManager em, MachineId env) {
    this.EM = em;
    this.Env = env;
  }

  public override void SendMessage(Socket s, Message msg) {
    PSharpRuntime.Send(this.Env, new MsgEvent(), s, msg);
  }

  public override void EnqueueMessage(Message msg) {
    this.EM.ProcessMessage(msg);
  }
}
```

Figure 4: The mocked network engine used for testing the Azure Storage vNext system.

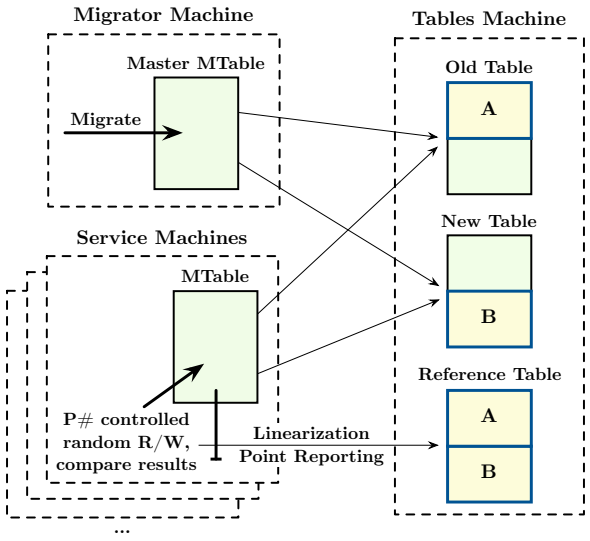

Figure 5: MigratingTable P# correctness test environment.

5 Experience Report

5.1 Distributed Extent Management System

We used P# to test the *distributed extent management* component of the Windows Azure vNext distributed storage system. This component is responsible for managing the partitioned extent metadata and works as follows.

5.2 Live Azure Table Migration

We also used P# to test the MigratingTable library, which is capable of transparently migrating a data set between tables in the Windows Azure storage service while an application is accessing the data set.

Since we were designing a new concurrent protocol that we expected to become increasingly complex over time as we add optimizations, we planned from the beginning to maintain a P# test harness along with the protocol to maintain confidence in its correctness.

MigratingTable implements an interface called IChainTable2, which provides the core read and write functionality of the real Azure table API with one exception: it provides a weaker consistency property for multi-page reads, since the original property would have been difficult to achieve for no benefit to applications we could foresee. MigratingTable requires that its backend tables also implement IChainTable2, and we wrote a simple adapter to expose physical Azure tables as IChainTable2. Our goal was then to verify that when multiple application processes issue "input" read and write calls to their own MigratingTable instances with the same backend tables, the behavior complies with the specification of IChainTable2 for the combined input history.

Since the specification is deterministic under sequential calls except for the results of multi-page reads, we decided the easiest way to formulate it for automated testing was to write an in-memory reference implementation called SpecTable. Given a multi-page read, SpecTable can actually produce a list of all valid results. Our correctness property is then:

> For every execution trace of a collection of MigratingTables backed by the same pair of SpecTables (which nondeterministically choose one of the valid results for each multi-page read) in parallel with the migrator job, there exists a linearization of the combined input history such that the output in the original trace matches the output of a "reference" SpecTable on the linearized input.

We instrumented MigratingTable to report the *linearization point* of each input call, which in our case is always one of the corresponding *backend calls* to the backend tables (often the last). Specifically, after each backend call completes, MigratingTable reports whether it was the linearization point, which may depend on the result of the call. This makes it possible to verify the correctness property as the system executes. We have a P# *tables machine* containing all three SpecTables, a collection of *service machines* containing identically configured MigratingTables, and a *migrator machine* that performs the background migration (Figure 5). Each service

machine issues a random sequence of input calls to its MigratingTable, which sends backend calls to the tables machine. When MigratingTable reports the linearization point of an input call, the service machine sends that input call to the reference table. When an input call completes, the service machine checks that the results from the MigratingTable and the reference table agree. P# controls the interleaving of the backend calls. To ensure that the reference table is never observed to be out of sync with the backend tables, after the tables machine processes a backend call, it enters a state that defers further backend calls until MigratingTable has reported whether the backend call was a linearization point and (if so) the call to the reference table has been made. We use the P# nondeterminism API to generate the input calls, so in principle an exhaustive P# behavior exploration strategy such as DFS could be used to exhaustively test MigratingTable up to some bound.

We wanted to implement the core MigratingTable algorithms in C# "async" code, like most of Artifact Services. Async code is readable like traditional procedural code but is translated by the compiler to an event-driven state machine using the Task Parallel Library, gaining most of the performance benefits of that style. By default, all event processing occurs on the .NET thread pool.

We had to arrange for the continuations to execute within the context of a single P# machine, which we did by installing a custom SynchronizationContext. Then we implemented async RPC between machines in terms of message passing, using RealProxy to avoid most of the marshaling boilerplate.

6 Evaluation

We compared the bug-finding effectiveness of P# systematic concurrency testing to traditional "stress testing" without systematic concurrency control on several benchmark bugs, described below. We report the length of time it took for the test to fail and remark on the effort needed to diagnose the bug from the test output. Matt: Overview of actual results?

6.1 Experimental Setup

All experiments were run on Matt: insert MSR machine, or perhaps CloudDevR1S04 if John/Matt are able to grant access to the rest of the collaborators. Matt: specs

6.2 Benchmarks

Pantazis: We probably do not need this now that we have many bugs from the case studies – benchmarks that are not production code (e.g. multipaxos) that can be used to evaluate testing

	#LoC		P# statistics			#Bugs
Azure Systems	**Real**	**Model**	**#M**	**#ST**	**#AB**	**found**
Azure Storage vNext	0	684	5	11	17	1
Live Table Migration	0	0	0	0	0	>10
Fabric User Service	0	0	0	0	0	0

Table 1: Statistics.

Cheng's case study with the actual bug

After fixing all the bugs we found in MigratingTable, we added an option to conditionally reintroduce each of the following bugs:

1. Filtering on a user-defined property did not retrieve non-matching rows from the new table that might shadow a matching row from the old table.

2. ...

6.3 Stress testing vs P# testing

Comparison of traditional stress testing VS what we do with P#

How long you had to run the test to find the bug (if you can find it) and how long are the traces

How easy it is to pinpoint the error using the P# traces, comparing to traditional stress testing

Effort required to use the system

Controlled random scheduling has proven to be efficient for finding concurrency bugs [17, 3].

7 Related Work

A significant amount of research has been conducted on how to analyze and test distributed systems [9, 16, 6, 5, 20], but a lot of these techniques either have significant limitations, or cannot be easily applied in a production environment, due to many complexities that are outside the scope of a research project.

In Verdi [19], a distributed system is written and verified in Coq, and then OCaml code is produced for execution. Verdi cannot find liveness bugs. P# is also more production-friendly (works on a mainstream language). Pantazis: say few more stuff

8 Conclusion

Draft.

	Stress Testing		P# Testing (Random Scheduler)			
Bugs	**Time (s)**	**Bug found?**	**Time to Bug (s)**	**#SP**	**%Buggy**	**Bug found?**
ExtentNodeLivenessViolation	0	✗			x%	✓
QueryAtomicFilterShadowing	0	✗	9.17	188	0.023%	✓
QueryStreamedFilterShadowing	0	✗	0.49	79	24.908%	✓
QueryStreamedLock	0	✗	2094.09	181	0.001%	✓
QueryStreamedBackUpNewStream	0	✗			x%	✓
DeleteNoLeaveTombstonesEtag	0	✗			x%	✓
DeletePrimaryKey	0	✗	1.65	151	5.958%	✓
EnsurePartitionSwitchedFromPopulated	0	✗	23.27	85	x%	✓
TombstoneOutputETag	0	✗			x%	✓
MigrateSkipPreferOld	0	✗			x%	✓
MigrateSkipUseNewWithTombstones	0	✗			x%	✓
InsertBehindMigrator	0	✗			x%	✓

Table 2: Results from 100000 iterations of P# random walk scheduler.

References

[1] CAVAGE, M. There's just no getting around it: you're building a distributed system. *ACM Queue 11*, 4 (2013), 30–41.

[2] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 398–407.

[3] DELIGIANNIS, P., DONALDSON, A. F., KETEMA, J., LAL, A., AND THOMSON, P. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), ACM, pp. 154–164.

[4] GRAY, J. 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* (1986), IEEE, pp. 3–12.

[5] GUPTA, D., VISHWANATH, K. V., AND VAHDAT, A. DieCast: Testing distributed systems with an accurate scale model. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2008), USENIX, pp. 407–421.

[6] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation* (2007), USENIX, pp. 18–18.

[7] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering 3*, 2 (1977), 125–143.

[8] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[9] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (1994), 872–923.

[10] LEIJEN, D., SCHULTE, W., AND BURCKHARDT, S. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (2009), ACM, pp. 227–242.

[11] MADDOX, P. Testing a distributed system. *ACM Queue 13*, 7 (2015), 10–15.

[12] MUSUVATHI, M., AND QADEER, S. Fair stateless model checking. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008), ACM, pp. 362–371.

[13] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX, pp. 267–280.

[14] NEWCOMBE, C., RATH, T., ZHANG, F., MUNTEANU, B., BROOKER, M., AND DEARDEUFF, M. How amazon web services uses formal methods. *Commun. ACM 58*, 4 (2015), 66–73.

[15] ODERSKY, M., SPOON, L., AND VENNERS, B. *Programming in Scala*. Artima Inc, 2008.

[16] SCHUPPAN, V., AND BIERE, A. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer 5*, 2-3 (2004), 185–204.

[17] THOMSON, P., DONALDSON, A. F., AND BETTS, A. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2014), ACM, pp. 15–28.

[18] VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. *Concurrent Programming in Erlang*, 2nd ed. Prentice Hall International, 1996.

[19] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), ACM, pp. 357–368.

[20] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (2009), USENIX, pp. 213–228.