Effective Concurrency Testing for Distributed Systems

Xinhao Yuan
xy2189@columbia.edu
Columbia University

Junfeng Yang
junfeng@cs.columbia.edu
Columbia University

Abstract

Despite their wide deployment, distributed systems remain notoriously hard to reason about. Unexpected interleavings of concurrent operations and failures may lead to undefined behaviors and cause serious consequences. We present Morpheus, the first concurrency testing tool leveraging *partial order sampling*, a randomized testing method formally analyzed and empirically validated to provide strong probabilistic guarantees of error-detection, for real-world distributed systems. Morpheus introduces *conflict analysis* to further improve randomized testing by predicting and focusing on operations that affect the testing result. Inspired by the recent shift in building distributed systems using higher-level languages and frameworks, Morpheus targets Erlang. Evaluation on four popular distributed systems in Erlang including RabbitMQ, a message broker service, and Mnesia, a distributed database in the Erlang standard libraries, shows that Morpheus is effective: It found previously unknown errors in every system checked, 11 total, all of which are flaws in their core protocols that may cause deadlocks, unexpected crashes, or inconsistent states.

CCS Concepts • **Software and its engineering → Software testing and debugging**; • **Theory of computation → Distributed computing models**.

Keywords distributed systems, randomized testing, conflict analysis, partial order sampling, partial-order reduction

ACM Reference Format:
Xinhao Yuan and Junfeng Yang. 2020. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378484

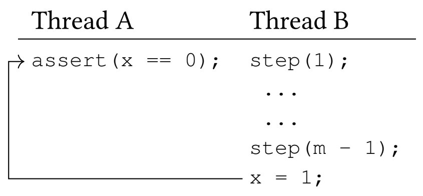
1 Introduction

Distributed systems are widely deployed and serve as a cornerstone for modern applications, but their correctness remains notoriously hard to reason about. A key reason is that, these systems need to not only perform complex operations of storage and computation, but also communicate with a high degree of concurrency and asynchrony because communication across the network can be slow and unreliable. Moreover, they must explicitly handle many types of failures, including partial ones, to ensure the correctness of the whole system, rather than simply calling `abort()` to terminate all components simultaneously. In critical scenarios, unexpected interleavings of operations and failures can lead the system into undefined states with serious consequences.

To surface corner-case errors due to unexpected interleavings, *systematic testing*, or model checking, aims to exhaustively enumerate through all possible interleavings [22, 36, 38, 53–55]. This interleaving space, however, is exponential to the number of the operations in an execution. For real-world systems, the interleaving space can be astronomical, far exceeding the limited testing budget, even with reduction techniques such as partial-order reduction [7, 8, 11, 19, 21] and interface reduction [23] that soundly reduce the interleaving space. Moreover, for exhaustiveness, systematic testing adjusts interleavings only slightly across trials. It is therefore prone to getting stuck in a homogeneous subspace, exploring mostly equivalent interleavings.

Instead of exhaustive enumeration, randomized testing tackles the intractable interleaving space via *sampling*, greatly increasing the diversity of the interleavings explored [13, 53]. The most straightforward sampling algorithm is *random walk*: at each step, randomly pick an enabled operation to execute. Previous work showed that random walk outperformed exhaustive search at finding errors in real-world concurrent programs [48]. This phenomenon is best explained by applying the *small-scope hypothesis* [24, §5.1.3] to the domain of concurrency error detection [28, 37]: errors in real-world concurrent programs are non-adversarial and can often be triggered when a small number of operations happen in specific orderings, which sampling has good probabilities to achieve.

Random walk, however, can have poor guarantees of detection for even simple errors. Consider Figure 1: The assertion in thread `A` fails if and only if it runs after "`x = 1`" in thread `B`, not before. Without prior knowledge of which



```
Thread A              Thread B
assert(x == 0);       step(1);
                       ...
                       ...
                      step(m - 1);
                      x = 1;
```

Figure 1. An example illustrating random walk's weak guarantee of error detection. Variable x is initially 0.

ordering fails the assertion, both orderings should be sampled uniformly because the probabilistic guarantee of error-detection is the *minimum* probability of sampling the two orderings. Unfortunately, random walk may yield extremely non-uniform sampling probabilities. In this example, to trigger the error, random walk has to not schedule the assertion for m times, yielding a probabilistic guarantee of only $1/2^m$.

Besides random walk, *probabilistic concurrency testing* [13] (PCT) samples interleavings with a bounded number of preemptions by changing thread priorities. PCT, however, uniformly samples different ways of preempting, despite that some of them have the same partial-order of operations and thus the same testing result. Such redundancy wastes the limited preemptions and weakens the guarantees of PCT.

To effectively reduce the bias of sampling any possible interleavings, Yuan et al. recently proposed *partial order sampling* (POS) [57] on multi-threaded programs. POS is simple yet effective: It assigns independently random priorities to operations, and schedules one operation at a time accordingly to the priorities. Compared with PCT, which changes thread priorities for limited times, the random priorities of operations in POS permute concurrent operations more uniformly without the limit. Formal analysis and empirical evaluation both showed that POS provides much stronger probabilistic guarantees than random walk and PCT to sample any partial-order of a program.

Despite the promising results, naïve application of POS in real-world distributed systems suffers from the sheer complexity increase going from multi-threaded programs. As aforementioned, real-world distributed systems often have a high degree of concurrency and asynchrony to handle failures and maintain consistent states in a distributed manner. Moreover, they often adopt a layered design in which low-level layers such as thread pooling, networking, and RPC, and high-level layers such as state-machine replication all generate many similarly looking operations. The massive amount of detailed operations prevent POS from surfacing protocol-level errors. Furthermore, unlike deterministic and exhaustive model checking, POS is history-less and has limited information to reorder operations for testing. The exploration unavoidably introduces bias in attempts to reorder useless operations whose orderings does not affect the testing result, since they cannot be identified dynamically until the end of the execution. We observed such operations the majority (more than 90%) in our tests, which greatly degraded the error-finding performance of POS.

We present Morpheus, the first tool that leverages POS to effectively find errors in real-world distributed systems. Morpheus tackles the challenges of complexity with two ideas. First, it introduces *conflict analysis* to summarize explored executions and predict what operations may conflict in future executions. Since it needs to apply POS on only conflicting operations, conflict analysis effectively accelerates error detection. Morpheus represents the history of explored executions in a succinct form whose size is proportional to the size of the program, incurring only minor bookkeeping. This idea benefits not only POS, but randomized testing in general.

Second, to focus on protocol-level errors, Morpheus targets higher-level languages and frameworks designed for distributed systems. Modern programming languages and frameworks increasingly provide first-class constructs to simplify building concurrent systems, such as Go [5], Erlang [4], Mace [25], and P# [17]. For instance, Erlang is well-known for its simple yet expressive support of distributed programming, such as messaging and fault-tolerance. It is widely adopted in large-scale distributed services such as Amazon SimpleDB [1] and the backend of WhatsApp [6]. By targeting higher-level languages and frameworks, Morpheus avoids being buried within the massive number of operations from low-level layers, and greatly boosts the chance of detecting protocol-level errors.

We implemented Morpheus for distributed systems written in Erlang. It leverages program rewriting to intercept Erlang communication primitives and explore their orderings, requiring no modifications to systems under test. Morpheus is written in Erlang, except 50 lines of modifications to the Erlang native runtime. Morpheus properly isolates itself from a tested system so that messages from Morpheus and the tested system cannot interfere. This isolation also enables Morpheus to run multiple virtual nodes on the same physical node, further simplifying checking. Morpheus provides a virtual clock [53] to check timer behaviors and to speed up testing. Whenever an error is found, Morpheus stores a trace for deterministic replay of the error.

We evaluated Morpheus on four popular distributed systems in Erlang including RabbitMQ, a message broker service, and Mnesia, a first-party distributed database in the Erlang standard libraries. Results show that Morpheus is effective: It found previously unknown errors in every system, 11 total, all of which are flaws in their core protocols, and will cause deadlocks, unexpected crashes, or inconsistent states of the systems. With POS and conflict analysis, Morpheus outperformed random walk and PCT with the overall advantage of 280.77%. The conflict analysis effectively improved the error-detection performance of Morpheus by up to 241.94%, and by 64.82% in average.

The rest of the paper is organized as follows: First we present an overview of Morpheus (§2), and describe its implementation (§3). Next we study all the errors Morpheus found (§4) and evaluate its effectiveness (§5). Finally we discuss related work (§6) and conclude (§7).

2 Overview

We first describe the workflow of Morpheus (§2.1), then explain our techniques for effectively finding errors in real-world systems (§2.2 and §2.3), and finally discuss its limitations (§2.4).

2.1 Morpheus Workflow

Testing with Morpheus requires minimal effort, almost the same as testing with the standard facilities in Erlang. A high-level workflow of Morpheus is shown in Figure 2. Normally, a test case of a Erlang project is a function with a special name, so that the Erlang testing framework can discover and run it automatically. Morpheus takes the function as the entry, repeatedly executes it for multiple trials to sample interleavings and find errors. Each trial begins with a Morpheus API call to execute the test function in a "sandbox", where Morpheus (1) isolates the test from external environment, (2) controls the execution of concurrent primitives, and (3) explores one of their possible interleavings.

During testing, it is crucial to isolate Morpheus and the tested system and avoid unexpected inference between them; otherwise errors caught by Morpheus may become not reproducible. Morpheus dynamically transforms modules, the unit of code deployment in Erlang, used in the test to isolate their namespace and references.

For execution control, Morpheus intercepts communication primitives in the test. Since Erlang uses the actor model, where code runs in user-level processes (processes for short) in Erlang virtual machines, and communicates through messages, Morpheus controls the messaging between the processes of the test. All primitives intercepted are replaced with our implementations, which (1) report the primitives to Morpheus as schedulable operations, (2) wait for Morpheus to resume the execution of the primitives, and (3) simulate the captured primitives or call the actual Erlang implementation of the primitives. Morpheus carefully handles primitives to avoid introducing impossible behaviors.

With all schedulable operations gathered from the test, Morpheus schedules one operation at a time with a randomized scheduler to explore an interleaving of the operations. Morpheus leverages POS to detect errors with high probabilities (§2.2). Each trial ends when the test function terminates or an error occurs. Morpheus performs conflict analysis (§2.3) on the explored trace to identify conflicting operations that affected the testing result. With the analysis results, Morpheus predicts conflicting operations in next testing trials,

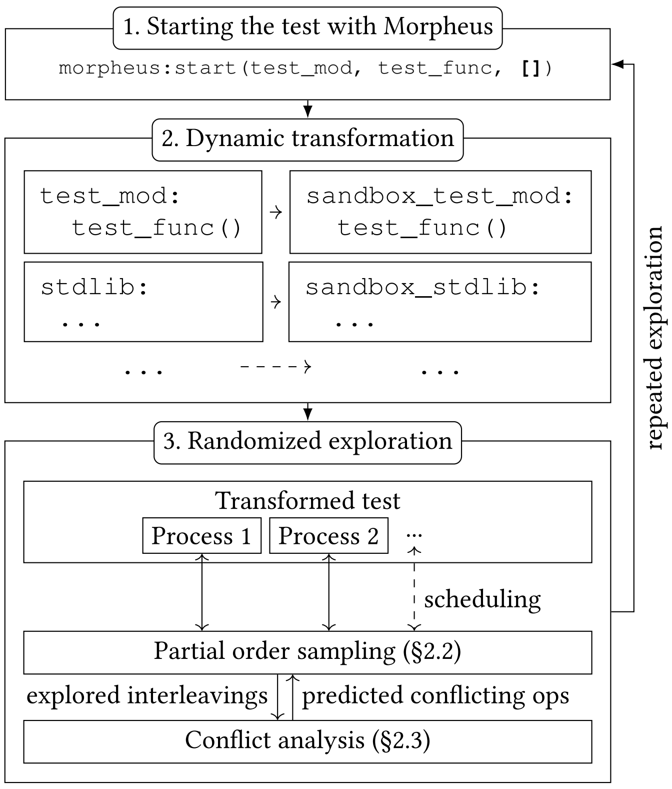

1. Starting the test with Morpheus
morpheus:start(test_mod, test_func, [])
2. Dynamic transformation
test_mod: test_func()
sandbox_test_mod: test_func()
stdlib: ...
sandbox_stdlib: ...
...
...
3. Randomized exploration
Transformed test
Process 1
Process 2
...
scheduling
Partial order sampling (§2.2)
explored interleavings
predicted conflicting ops
Conflict analysis (§2.3)
repeated exploration

Figure 2. The workflow of Morpheus.

and focuses on reordering them to improve the probabilistic guarantees of randomized testing.

Finally, the testing result is reported to the developer, together with information for replaying the test with the same exact interleaving. The test succeeds if the entry function returns normally; The test fails if (1) the entry function exited with a fault, (2) the test calls into the Morpheus reporting API to report an error, (3) the test runs into a deadlock such that all processes are blocking on receiving messages, or (4) the virtual clock (§3) or the number of executed operations exceeds their user-configurable limits. A developer may run the test with Morpheus for many trials to explore different interleavings and gain confidence.

2.2 Exploiting the Small Scope Hypothesis with POS

Previous study [28] showed that errors in real-world distributed systems have simple root causes, which sampling can manifest with high probabilities. One must carefully choose a sampling strategy, since its bias may add up and significantly degrade the probabilistic guarantees of error-detection even when the root causes are simple.

A real-world example with a simple root cause. Figure 3 shows a simplified version of a real-world error we found in `gen_leader`, a leader election protocol implemented in Erlang and used by `gproc`, a feature-rich process registry.

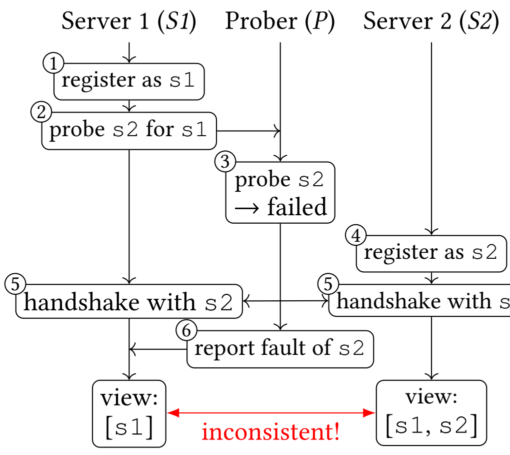


Figure 3. An inconsistent view error in `gen_leader`, a leader election protocol implemented in Erlang. "→ . . ." denotes the result of its preceding operation.

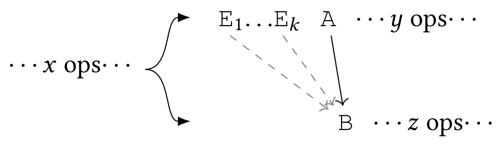


Figure 4. POS can sample the ordering of A→B depending only on the priorities assigned to A, B, and E_i, despite that there are numerous other x, y, and z operations.

This error causes the system to get stuck and handle no further requests. It is triggered by an untimely fault message from Prober (*P*) to Server 1 (*S1*), which causes *S1* and Server 2 (*S2*) to have different views of the alive servers even after a successful handshake.

In `gen_leader`, the leader election process of each node[1] spawns a prober process to monitor other servers and handles any fault message from them. In bootstrapping, however, if the prober *P* of *S1* monitors *S2*, by querying the process registered as symbol `s2`[2], while *S2* has not yet registered as `s2`, *P* would report a fault message to *S1*. This fault message may be delayed due to, e.g., OS scheduling. When later *S2* is up and handshakes with *S1*, both *S1* and *S2* enter the normal operational phase. Later *P* is resumed and sends the delayed fault message to *S1*, which will make *S1* believe that *S2* fails after the handshake. Depending on which server is elected as the leader, the delayed fault message may either make *S1* remove *S2* from its view of alive servers (if *S1* is the leader), or otherwise make *S1* roll back to election phase. In either case, further requests from `gproc` to `gen_leader` will not be handled as *S1* and *S2* disagree on the cluster membership and cannot recover. To detect this error, ④ must happen after the ①, ②, and ③; ⑥ must happen after ④ and ⑤.

[1]In Erlang, nodes are Erlang virtual machines connected with networks.
[2]Each Erlang node maintains a registry mapping symbols to processes.

Redundancies in random walk. The major drawback of random walk is the exponential bias to delay one operation after other operations. In this example, random walk takes at most $(2/3)^k$ probability to delay operation ⑥ for k times, the number of operations in ④ and ⑤. Real-world errors often require operations to delay such that random walk is unlikely to manifest due to the exponential bias.

Redundancies in PCT. As briefly described in §1, PCT [13] randomly selects a small number of operations to delay (preempt) by manipulating the priorities of processes right before the selected operations. Compared with random walk, PCT takes linear (to the number of operations) probability to delay a operation in contrast to the exponential probability. The major drawback of PCT is that it selects the operations to delay among all operations in a test. Without precise knowledge of an error, however, a test often includes redundant operations irrelevant to the error, which "distracts" PCT from delaying the desired operations, and degrades its guarantees. In this example, PCT needs to (1) initially assign the priority of *S2* lower than priorities of *S1* and *P*, and (2) select ⑥ among all operations to delay. The original test of Figure 3, however, contains operations for system initialization before ①, and further operations after ⑥ for handling the actual `gproc` requests in the test. As a result, PCT has to guess the exact operation ⑥ to delay among more than 6000 operations (as in Table 4) to find the error.

Finding errors in redundant tests with POS. Morpheus leverages the recent work of POS [57] to handle the redundancies in real-world tests. POS is simple: it assigns independently random priorities to operations, and always schedules the pending operation with the highest priority. Similar to PCT, POS also has the linear probability to delay an operation in contrast to the exponential probability of random walk. The key benefit we discovered about POS over PCT is that the ordering of a subset of operations in a test only depends on priorities assigned to the operations in this subset. Thus POS is a good match for finding errors with simple root causes, even in tests with a lot of irrelevant operations. Conceptually, consider the example in Figure 4. A prefix of x operations sets up concurrent processes executing operations A and B, where A is preceded by $E_1, \ldots, E_k$. If an error is caused by the ordering of A→B, its scope would include operations E_i as well, because the scheduler must schedule E_i before B in order to schedule A before B, but the scheduler can schedule the other $x + y + z$ operations arbitrarily without missing the error. To manifest the error, POS requires the random priority of B to only be lower than the random priorities of A and $E_1, \ldots, E_k$, which happens with the probability of $1/(2+k)$, even with the presence of numerous other operations. Back to the real error of Figure 3. Despite the operations of initialization and further user requests, POS would need only the priority of ④ lower than those of ①, ②, and ③; and the priority of ⑥ lower than those of ④ and ⑤.

A novel discovery in Morpheus is that, although the advanced version of the two previously proposed POS algorithms [57] performs 2∼3× better on multi-threaded programs, the basic version is actually (slightly) better in our evaluation. We believe the reasons are two-folds: the errors in our evaluation do not trigger the worst case of the basic POS. Moreover, the advanced POS suffers from the coarse-grained dependency tracking in Morpheus due to the complexity of message semantics in actor model compared with memory accesses. As pointed out in the original paper, in extreme cases, false dependencies could make advanced POS degenerate to random walk. Morpheus thus uses the basic POS as its default strategy.

2.3 Optimizing POS with Conflict Analysis

Although POS performs well to find errors in a small subset of operations, it still suffers from sampling redundant orderings of *non-conflicting* operations. In Erlang, two operations *conflict* if they are concurrent, and they send messages to the same process or access its state. In contrast, non-conflicting operations never conflict with others, and ordering non-conflicting operations with other operations does not change the testing outcome. Specifically in Figure 3, the operation labeled ② is non-conflicting. When POS explores the example, it assigns ② a random priority, and delays ② until ② becomes the highest-priority pending operation. Such delaying lowers POS's performance of finding errors: No error would require delaying non-conflicting operations, but some errors can be missed because of such delays - the error in Figure 3 would be missed if ② is delayed after ④. As shown in §5.3, we observed non-conflicting operations as the majority of operations in our test cases, an order of magnitude more than conflicting operations.

This shortcoming reflects a fundamental limitation of history-less randomized testing algorithms: whenever such an algorithm needs to schedule an operation, it has no idea what operations may occur in the future, therefore it has to assume that all operation can conflict with some future operations, and sample their orderings accordingly.

In contrast, partial-order reduction algorithms designed for systematic testing do not suffer from this problem. After the current trial of testing, these algorithms examine the operations executed and detect which conflicting operations could have been executed in a different ordering. They backtrack to reorder only such operations (in the optimal case). However, modern partial-order reduction approaches (e.g. [7, 19]) require the testing to be depth-first to backtrack an interleaving prefix only after exploring all interleavings with the prefix. Such requirement directly conflicts with randomized testing, therefore they cannot be applied to POS.

Conflict analysis in Morpheus represents a novel design point in the randomized testing algorithm space. Compared with completely history-less randomized algorithms and deterministic algorithms with the full history of prior trials, Morpheus keeps a concise summary of explored executions and use it to predict in future explorations. To do that, Morpheus maps operations into compact signatures, such that operations with the same signature are likely to coincide on whether they are conflicting or not, and maintains a history table for each signature whether any operation with this signature has ever conflicted in previous explorations.

To maintain the history table, Morpheus detects conflicting operations in the explored execution after each test trial. Morpheus detects conflicts by constructing the happens-before relation of the executed operations with vector clocks [35], and checking whether two operations accessing the same process have concurrent vector clocks. For any conflicts found, Morpheus records the signatures of the conflicting operations into the table. In next trials, whenever Morpheus sees a new pending operation, it queries this table and immediately schedules the operation if no prior operations with the same signature ever conflicted.

An open design trade-off is the scheme of operation signatures. Intuitively, a signature scheme with more details would have better precision (i.e. less false positives) in the prediction. Such scheme, however, would have larger overhead in time and space for bookkeeping. On the other hand, one could simply use the static code locations of the operations as the signature scheme, which may not help POS much due to its bad precision. Currently, Morpheus maps each operation into the signature of `{P, PC}`, where `P` is the process id of the operation and `PC` is its static location of code. In our evaluation, such scheme improved the precision from the naïve signature of static locations by up to 219.67% and improved the error-finding performance by up to 65.62%. Investigating more schemes for the trade-off of precision and overhead is left as future work.

Besides false positives, conflict analysis could also have false negatives, where an operation is conflicting with future operations but Morpheus predicts otherwise. Such false negatives may add slight bias for the current trial, but it does not affect the probabilistic guarantees much because, after the current trial, Morpheus will update the history table to record that conflict involving this operation. In our experiments, we rarely saw false negatives after Morpheus populated the history table in a hundred of trials.

2.4 Limitations

Morpheus focuses on concurrency testing and does not actively inject failures. Instead, Morpheus relies on test cases to manifest node failures, and explores any interleavings of the failures with normal system operations. In Erlang, it is straightforward to simulate node failures using standard

Original receive

```
receive
  {reply, R}
    when R =/= error ->
      R
end
```

Extracted pattern function

```
PatFun =
fun (Msg) ->
  case Msg of
    {reply, R}
      when R =/= error ->
        true;
    _Other ->
        false
  end
end
```

Figure 5. `receive` pattern and its transformed matching function. The branch of the `receive` is taken only if the process receives a message of a tuple with two elements, where the first element is symbol `reply`, and the second the element (assigned to `R`) is not symbol `error`.

APIs, such as `erlang:disconnect_node`. We can leverage bounded and randomized approaches [9, 12, 26, 33, 53] to inject failures and find interesting errors. Morpheus does not support low-level failures such as network packet losses.

Morpheus shares similar limitations with prior systematic or randomized testing tools. It relies on test cases to provide inputs to the tested system and high-level invariants to check in addition to fail-stop errors and infinite loops. It supports limited ways of interacting with non-Erlang code (§3).

3 Implementation

We implemented Morpheus in ~8500 lines of code in Erlang and a patch of ~50 lines to the Erlang virtual machine in C. We here describe some of its implementation details.

Instrumentation and isolation. As mentioned in §2.1, Morpheus transforms any modules used in the test to (1) collect available operations, and (2) reliably control their interleaving without interference. Morpheus transforms a module by traversing its low-level syntax tree (of CoreErlang [14]) from the module binary, available in debugging information. Morpheus traverses the syntax tree for each function definition, redirects all calls of concurrency primitives to Morpheus handlers, where their functionalities are thoroughly simulated for both isolation and controlling.

To gather all schedulable operations, Morpheus needs to know when a primitive is available to execute, most primitives in Erlang are non-blocking and always schedulable. The only exception is receiving messages, which blocks until any message meets specified patterns. Morpheus encapsulates the pattern-matching into a predicate [15, 16] that returns true if any pattern is matched, as shown in Figure 5. Morpheus maintains virtual inboxes to keep all pending messages in order, and uses the predicate to find the first matched messages in the inbox. If no message meets the predicate, the handler reports as blocking, and waits until new messages arrive or it times out.

To reliably control the interleaving, Morpheus always waits for all processes in the test to give back control before scheduling any operation. Sometimes an operation may have concurrent side-effects. Morpheus enforces a deterministic ordering for such side-effects.

To isolate the test, Morpheus dynamically translates names referred in the test to avoid interference with Morpheus. Modules used in tests are dynamically transformed and loaded with new names with special prefixes (e.g. "`sandbox_`"). External references, e.g. files, are left uncontrolled for simplicity, as they usually can be configured in tests to avoid interference. If stricter isolation is required, a developer may use a container or virtual machine to execute the test.

Timing semantics. Real-world systems inevitably depend on real-time conditions to function correctly, and it is important to handle them in a desired way, otherwise we may spin with false errors impossible in practice. Currently Morpheus simulates the ideal semantic of timing by maintaining a virtual clock. All operations run infinitely fast and cost no virtual time, and time-outs happen in the strict order of the their deadlines. All deadlines of time-outs are maintained in a sorted queue. At any state, a timing operation is disabled unless its deadline is met. When there is no operation available but pending deadlines, Morpheus fast-forwards the virtual clock to the nearest deadline (similar to [53]), and triggers the time-outs whose deadlines are met.

To detect liveness errors where a system takes forever to finish the test, Morpheus limits the time of deadlines: Morpheus will not trigger any deadline after the limit. Thus any execution requiring more time than the limit will be reported as a deadlock. A few liveness errors lead to infinite loops without time-outs, for example, two processes keep sending/receiving messages with each other without handling test requests. Morpheus limits the number of operations to execute to detect such infinite loops. We believe that the timing semantics in Morpheus is a balanced trade-off between the program intention, testing effectiveness, and efficiency.

Simulating a distributed environment. Many tests of distributed systems require a distributed environment of multiple nodes. Instead of running physically distributed tests with Morpheus, error-prone and potentially having unwanted overhead, Morpheus simulates virtual nodes in a single physical node. Thanks to the unified process communication in Erlang, no extra transformation is needed for putting virtually remote processes in the same physical node. Morpheus needs to, however, properly translate remote name references. Doing so in pure Erlang is non-trivial due to the language restrictions — we instead modified the Erlang VM minimally.

For simplicity, Morpheus only simulates a fully connected cluster of nodes. As mentioned in §2.4, Morpheus does not actively inject network failures. A developer can inject a

Table 1. Summary of the distributed systems tested and errors founded by Morpheus

Name	Description	KLOC	Errors
`locks`	Lock manager	4.1	2
`gproc`	Process registry	7.3	3
`gen_leader`	Leader election	1.7	
`mnesia`	DBMS	27.3	2
`rabbitmq`	Message broker	60.7	4
`ra`	Replicated log	8.6	
Total		109.7	11

node failure in a test by shutting down all processes in the node, easy and covering all of our testing needs.

Non-Erlang code. Erlang systems commonly involve non-Erlang code for the sake of performance (e.g. cryptographic computation). Morpheus identifies all interactions with non-Erlang code during module transformation and runtime, and carefully handles them without introducing impossible executions. After each interaction with non-Erlang code, Morpheus waits for a small amount of time (50ms by default) to let the interaction stabilize. Morpheus assumes that external interactions are non-blocking, stateless, and deterministic. Morpheus ignores the conflicts involving non-Erlang code.

4 Errors Found

We applied Morpheus to four distributed systems in Erlang, and found 11 previously unknown errors. The target systems and errors found are summarized in Table 1. We targeted these systems because of their popularity and well-defined APIs. Most of the tests are written by us, since most of the original tests are non-concurrent or stress tests (except for `gproc`, where we used their test scenarios).

Creating tests for these systems were simple: we quickly studied their APIs by looking at their documents and original tests, identified intuitively error-prone scenarios, such as adding replicas to a node while concurrently shutting down the node, and implemented those scenarios using a few API invocations. We created the tests without knowing the system internals or the errors. Each of these test scenarios contains 15-30 lines of code.

Among all the 11 errors we found, three are confirmed by developers and all of them are reproduced afterward without Morpheus. We now describe the errors.

4.1 `locks`: a Decentralized Lock Manager

`locks` is a decentralized lock manager. It provides a transactional locking API for clients to request locks, detect deadlocks caused by circular waits, and surrender locks to avoid deadlocks. To acquire a lock `x`, a client sends a "request `x`" message to the lock server process in `locks`. The server in turn replies with the queue of the clients acquiring `x`, the head of which is the current holder of the lock.

Although the authors claimed that the core locking protocol of `locks` has been model checked despite minor differences in the implementation [52], we managed to find multiple errors in it, described as follows.

Prematurely resetting version (`locks-1`). In `locks`, the lock servers and clients maintain the wait queues of locks in lock entries, tagged with monotonic version numbers to identify outdated entries in delayed messages. When a lock server *S* sees an available lock without any process waiting to acquire, it will remove the entry of the lock from its local state, effectively resetting the version of the entry. On the other hand, according to its protocol, a client may propagate its entries of some locks to another client that may or may not be waiting for the locks, in order to detect circular waits in a decentralized way. A concrete example is showed in Figure 6, where three clients *A*, *B*, and *C* are acquiring locks `x` and `y` with server *S*.

Initially *A* and *B* acquire locks `x` and `y` separately. Then *A* and *C* attempt to acquire `y`, and *S* notifies all three nodes that *B* is the current holder of `y`, and *A* and *C* are waiting in `y`'s queue in this order. Later, *B* requests `x` and detects a circular wait, so *B* surrenders `y`. It also propagates its entry of lock `x` to *C* in ① according to the protocol, even though *C* has not yet requested to acquire `x`. This design of the protocol is presumably for performance. After *A* and *B* exit, *S* removes the entry of lock `x` in ② because no one is requesting the lock, while the client *C* still has the now stale entry it received from *B*. When later *C* requests `x` from *S*, the request will never complete, because *S* will reply with a fresh entry of `x` "older" than *C*'s version. *C* will ignore the reply, and keep waiting for the "up-to-date" entry that would never arrive.

Atomicity violation in server probing (`locks-2`). In a cluster of nodes, when a client tries to acquire a global lock, the client needs to communicate with remote lock servers on other nodes. Since the remote servers may start later than the request, the client spawns a prober process in the remote node to wait for the server. The prober process first queries for any process registered as `server`. If not, the prober registers itself as `watcher`, and the server will send notification to any process named `watcher` once it starts. If the server starts after the probe query of `server`, but sends notification before the prober registers `watcher`, as shown in Figure 7, the notification will get lost and the prober will never detect the server, causing a deadlock.

4.2 `gproc` and `gen_leader`: an Extended Process Registry with Leader Election

`gproc` is a feature-rich process registry, subsuming the primitive process registry of Erlang. It maintains the registry across a cluster of nodes on a leader elected by `gen_leader`, an implementation of leader election protocol. We tested

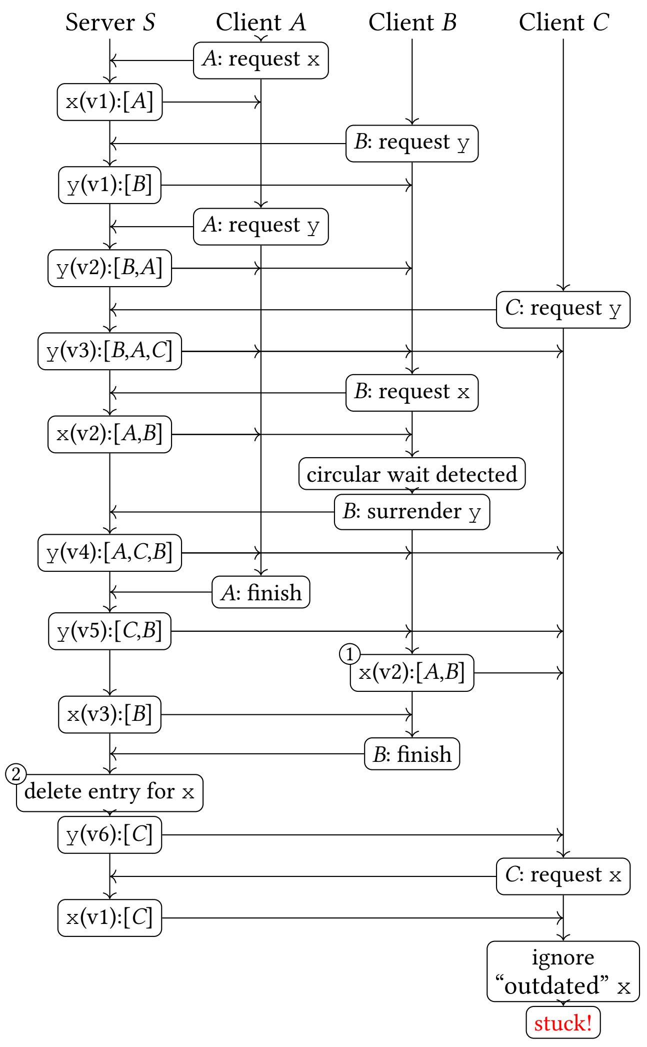


Figure 6. The deadlock caused by prematurely version resetting. x and y are names of locks. Messages such as "x(v2): [A,B]" represent lock queuing information with a version number: The example is a message with version two, where *A* holds the lock and *B* is waiting for it.

gproc and gen_leader and found errors in two of our tests and one original test.

***Delayed fault reports* (gproc-1).** We tested the bootstrapping procedure of gproc by setting up the distributed environment of three nodes, and then run a simple request of registering a process. We found that gen_leader could not handle some corner cases in bootstrapping, and went in to an unrecoverable state where some nodes can never join the cluster. The simplified root cause of this error is showed in Figure 3 and described in §2.

***Reverted registration* (gproc-2).** We found an error of the reg_other API, which registers a name with a remote process in a different node from the caller. This error reverts

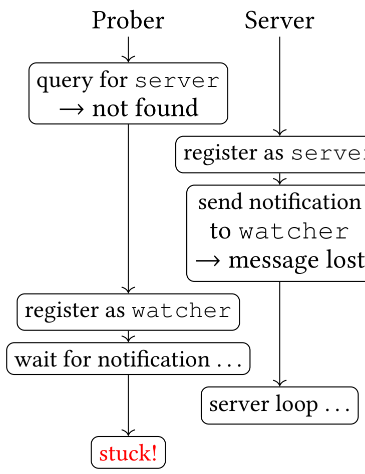

Prober
Server
query for server → not found
register as server
send notification to watcher → message lost
register as watcher
wait for notification . . .
server loop . . .
stuck!

Figure 7. The atomicity violation in server probing. "→ . . ." denotes the result of its preceding operation.

a successful registration, silently dropping all messages sent to the registered name. It happens as follows. Before the a node *A* joins a cluster (maintained by gen_leader), it has no knowledge of the cluster process registry because no node in the cluster sends *A* updates. However, a process in another node of the cluster can register *A*'s process *P* with reg_other. Once *A* joins the cluster, it believes that it has the most up-to-date registry about its own processes, and removes the entry of *P* from the cluster registry. After that, all messages to the previously registered name will be silently dropped, breaking the contract of reg_other.

***Lost requests with dying leader* (gproc-3).** It is important for leader election to tolerate node failures. We created a test where we first set up a cluster of three nodes, and then concurrently kill the leader node (by shutting down the gen_leader processes in the node) while issuing a gproc request on another node. Normally, gproc would use gen_leader to forward the request to the leader server. If the leader is dead, the cluster would receive notifications and go back to election, and the request would be buffered until the new election is done. If the request is sent after the leader is dead, but before the fault notification is received, the request would be lost and never be replied.

4.3 mnesia: a Distributed Database System

mnesia is a feature-rich database system that comes with the Erlang standard libraries (OTP). It provides ACID transactions and replication across distributed nodes, and has been the standard storage solution for distributed applications in Erlang. We found two errors when mnesia handles requests with simultaneous faults in a cluster.

***Atomicity violation in initialization* (mnesia-1).** We tested the functionality of adding replicas of a table *T* by

```
sync_schema_commit(_Tid, _Store, []) ->
  ok;
sync_schema_commit(Tid, Store, [Pid | Tail]) ->
  receive
    {?MODULE, _, {schema_commit, Tid, Pid}} ->
      ?ets_match_delete(Store,
          {waiting_for_commit_ack, node(Pid)}),
      sync_schema_commit(Tid, Store, Tail);
    % [MORPHEUS] Bug! Failure notification
        treated the same as commit ack
    {mnesia_down, Node} when Node == node(Pid) ->
      ?ets_match_delete(Store,
          {waiting_for_commit_ack, Node}),
      sync_schema_commit(Tid, Store, Tail)
  end.
```

Figure 8. The sync code for `mnesia` schema change, where the message of fault notification "`{mnesia_down, Node}`" is also treated as an acknowledgment of the commit.

requesting to replicate *T* from node *A* to node *B* while restarting `mnesia` in *B*. We found a deadlock when `mnesia` handles the replication request in *B* during initialization. When the test requests to replicate table *T*, `mnesia` will produce transactions to update the list of replicas of *T* in all nodes of the cluster. Concurrently, when *B* restarts and initializes `mnesia`, it will process multiple transactions to iteratively merge its schemas with other nodes. During the test, `mnesia` in *B* will process two transactions *X* and *Y* for schema merging, and transaction *Z* for adding *B* to the list of replicas. The internal synchronization of `mnesia` makes sure that *X* always commits before *Z*, but when *Z* is processed before *Y*, *Z* will grab *T*'s schema lock in `mnesia`'s lock manager, and try to commit. Because `mnesia` in *B* is still in initialization, the committing of *Z* will be postponed in a queue. When later *Y* starts in another process, it will try to grab the same table schema lock, but *Z* is holding the lock and waiting for the initialization. *Y* and *Z* thus form a circular wait and deadlock.

Reverted copy removal (*`mnesia-2`*). We created another test where we delete a node from the replicas of a table while restarting the node. We found an error that the node reverts the schema change after the restart. When `mnesia` runs the last phase of committing the schema change, the node *A* initiating the change needs to waits for all other nodes to acknowledge their commits of the schema change. When another node *B* shuts down after entering this phase (i.e. after *A* sends the commit message to *B*), but before *B* persists the commit, *B* will send a fault notification to *A*. *A* will take the fault notification as an acknowledgment of *B* (see Figure 8), and let the removal request succeed. Later when *B* is back, it will replay its operation log without the last schema change, and bring the outdated replica schema back.

4.4 `rabbitmq` and `ra`: a Message Broker with Raft Replication

`rabbitmq` is a sophisticated message broker service that implements high-level messaging protocols with rich semantics support and high performance. It provides highly available "quorum queues" using `ra`, the implementation of Raft consensus protocol [41]. The protocol maintains a consistent log replicated across a cluster. Whenever user requests come, the unique leader of the cluster, elected by the protocol, serializes the requests, appends them into the log, and safely replicates the log even in the presence of partial failures. We created three tests for `ra` that concurrently issue requests of enqueue/dequeue operations, leader elections, and configuration changes, where we found three errors (`ra-{1,2,3}`). These errors are tricky such that the interleavings triggering them are highly complex. Once Morpheus produces these interleavings, however, diagnosing the root causes becomes easy. One of the errors happens when a `ra` server in a node is waiting for other servers to replicate the log entries missing in this server. When a concurrent election request comes to the server, `ra` will buffer this request but forget the requester information by mistake. Later when the log replication is done, `ra` handles any buffered requests on the server and replies them. Since the election request was buffered without the requester, `ra` will panic and crash unexpectedly. The other two errors involve `ra`'s incorrect assumption that, when reverting to an old view of cluster membership due to partial failures, nodes should always have the old view stored locally.

Besides `ra`, we also tested `mirrored_supervisor` module in `rabbitmq`, which maintains a replicated group of special "supervisor" processes to manage other processes across a cluster. We found an error (`ms-1`) in a test that concurrently adds and removes supervisor processes from the group. The test initially sets up a group of supervisors *A* and *B*, requests to add a new supervisor *C* to the group. To add *C* into the group, `mirrored_supervisor` acquires the current list of members (currently *A* and *B*), then it queries each member by sending messages. If *B* gets shut down after the list is acquired but before *B* handles the query. The query will produce an error unexpected by `mirrored_supervisor`, leading to a crash.

5 Evaluation

Our evaluation of Morpheus focuses on the following research questions:

1. How does Morpheus's exploration algorithm, POS, compare to systematic testing with state-of-the-art partial-order reduction techniques? Although POS has been shown effective in testing multi-threaded programs, it is unclear how much it helps testing distributed systems. (§5.1)

2. How effective does Morpheus find the errors with POS and conflict analysis, compared with other randomized strategies? (§5.2)
3. How much can conflict analysis improve Morpheus? (§5.3)
4. What is the real-time performance of Morpheus? (§5.4)

We conducted all the experiments and studies in this paper on workstations with two Intel(R) Xeon(R) E-2640 CPUs, 12 cores in total, and 64 GB of RAM, running Ubuntu GNU/Linux and our patched version of Erlang/OTP 20.3 and 21.3 (required by `ra`).

5.1 Comparison with Systematic Testing

To understand how randomized testing compares with systematic testing in distributed systems. We implemented randomized testing in Concuerror [15], a state-of-the-art systematic testing tool for Erlang equipped with advanced dynamic partial-order reduction (DPOR) techniques [7, 11]. Our modification of Concurerror is small (~500 LoC) as systematic testing shares most parts of runtime control and isolation with randomized testing, and only differs in scheduling strategies.

Concuerror is not suitable to test real-world systems for multiple limitations. First, the test isolation of Concuerror is incomplete, and cannot properly handle interactions between the target systems and some stateful modules used by Concuerror. For example, most systems in Erlang use the stateful `application` module to set up themselves, but the module is not isolated by Concuerror. Secondly, Concuerror simplifies real-time conditions by allowing time-outs to fire regardless of their deadlines. The simplification makes some common scenarios hard to test. For example, some tests need to wait for a delay after issuing the testing requests to make the result stable . We thus studied three simplified scenarios: the chain replication protocol checked by Concuerror in the previous work [10], the Erlang specification for a Cassandra error, and the "lock-1" error we studied. For a quick summary, POS outperformed systematic testing with DPOR in all cases we studied.

***`crce`*: chain replication protocol in `Corfu`.** Previous work [10] demonstrated that Concuerror was able to find violations of linearizability on the specification of the chain replication protocol [49] as a part of Corfu distributed shared log system [34]. The authors crafted four variants of the protocol specification, and they managed to find errors in two of them, namely `crce-2` and `crce-3`. We applied randomized testing on the same tests to compare with systematic testing. We profiled the number of trials for systematic testing to reach the first error. For randomized testing, we profiled each of the algorithms by the average number of trials needed for detecting an error over 100,000 trials.

Table 2 shows the results. POS had the similar performance to random walk in `crce-2`, outperformed random walk in `crce-3` and systematic testing in both cases.

Table 2. Error-detection comparison on the chain replication protocol. The "Systematic" column shows the number of trials for systematic testing to find the error for each case. The "Random walk" and "POS" columns show the average number of trials for the algorithms to find the error in each case.

Case	Systematic	Random walk	POS
`crce-2`	81	4.29	4.39
`crce-3`	119	1351.35	17.43

`C6023`*: Cassandra lightweight transactions.* Inspired by previous case study on distributed system errors [28], we studied a sophisticated error [3] in the lightweight transaction protocol of Cassandra [2]. We modeled the error in Erlang, and used Concuerror to check with systematic and randomized strategies. We performed 100,000 trials for each of the strategies. Systematic testing and random walk could not find the error in all trials, while POS was able to hit the error for 20 times.

The* `locks-1` *error. For all the errors we have found with Morpheus, only `locks-1` is supported by Concuerror. The detailed error is described in §4.1. Unfortunately, no DPOR technique implemented in Concuerror worked - they all timed out on planning the next interleavings to explore after the first trial. We were not able to diagnose the issue, and it may indicate errors in its DPOR implementations. We instead ran the standard systematic testing. We ran the test for 100,000 trials for each of the testing methods. Systematic testing found no error; Random walk detected the error for only one time; POS detected the error in 18,987 trials (i.e. ~5.27 trials per error).

5.2 Morpheus Error-detection Performance

For all the errors we have studied in §4, we used Morpheus on the tests with different randomized testing algorithms, including random walk, PCT, and POS, with and without conflict analysis (denoted with "+"). The original PCT algorithm was for multi-threaded programs, but it still applies here due to the instant message delivery in our concurrency model. For PCT, we set d, the number of preemptions, to five without the knowledge of the errors. We profiled each of the tests with 100 runs in POS to estimate the total number of operations for PCT. Besides the basic POS, we also evaluated the advanced version, which not only assigns random priority to operations, but also re-assigns the priorities of pending operations when any conflicts are observed for the operations during the scheduling. We denote the advanced version as "POS*" hereafter.

For each combination of testing algorithms and tests, we performed 10,000 trials in 10 parallel tasks, so each task ran 1000 trials. Each task starts with no history for conflict analysis. We collected for each combination the hit-ratio, the

Table 3. Error-detection performance of different randomized testing algorithm. "+" means with conflict analysis. "**Mean**" row summarizes each of algorithms with geometric means. "**Ratio to POS+**" row shows the ratio of overall performance compared with POS+. "**CA Improvement**" row shows the improvements of conflict analysis on the average performance for each algorithm. The summary of random walk is not available due to the missed errors.

Case	RW	RW+	PCT	PCT+	POS	POS+	POS*	POS*+
`locks-1`	0.0042	0.0312	0.0895	0.1164	0.1521	0.2087	0.1562	0.2239
`locks-2`	0.0210	0.0117	0.0022	0.0071	0.0073	0.0124	0.0103	0.0140
`gproc-1`	0	0.0001	0.0015	0.0018	0.0031	0.0106	0.0008	0.0023
`gproc-2`	0	0.0190	0.0496	0.0781	0.0605	0.1170	0.0416	0.0648
`gproc-3`	0	0.0002	0.0431	0.0385	0.0156	0.0450	0.0027	0.0094
`mnesia-1`	0	0.0001	0.0219	0.0223	0.0141	0.0168	0.0091	0.0124
`mnesia-2`	0	0.0008	0.0075	0.0078	0.0117	0.0253	0.0104	0.0254
`ms-1`	0	0.0061	0.3000	0.2833	0.1489	0.2420	0.1505	0.2478
`ra-1`	0	0	0.0025	0.0036	0.0070	0.0120	0.0062	0.0126
`ra-2`	0	0	0.0010	0.0011	0.0053	0.0042	0.0063	0.0044
`ra-3`	0	0.0001	0.0003	0.0002	0.0032	0.0031	0.0032	0.0033
Mean	N/A	N/A	0.0089	0.0106	0.0151	0.0249	0.0109	0.0180
Ratio to POS+	N/A	N/A	35.62%	42.66%	60.67%	100.00%	43.74%	72.01%
CA Improvement		N/A		19.77%		64.82%		64.65%

number of trials that surfaced the error divided by the total number of trials. The results are shown in Table 3. Note that all three errors we found in `ra` appear in all our tests of `ra` with different probabilities. We aggregated the results by the three errors. The hit-ratio of each algorithm is summarized in the "**Mean**" row with geometric means. Random walk cannot detect 2 of 11 errors even with conflict analysis, thus its geometric means are not available.

The basic POS with conflict analysis performed the best in error-detection on average. It outperformed our baseline, random walk, and PCT, on most of the errors, and the overall advantage is 280.77%. The results also show that, to our surprise, POS* (POS with priority reassignment) degrades the error-detection performance by 28% from POS in average, given the POS* worked better on multi-threaded programs in prior work [57]. Further experiments confirmed that POS*+ did sample more partial-orders than POS+. These results reveal a key insight: tools consider two shared memory operations conflict only if they access the same exact location (and at least one is a write), but two message sends conflict if they target the same recipient process even when the handling of the messages has independent effects. Therefore, the partial-order relations defined on distributed systems tend to be overly conservative. Some of the false dependencies can be alleviated by fine-grained dependency tracking [32], while semantic independence of operations is still hard to efficiently identify on the fly.

5.3 Effect of Conflict Analysis

To understand how conflict analysis can reduce the useless operations for Morpheus to explore, we profiled conflict analysis in two cases, `gproc-1` and `mnesia-2`, for the total number of operations, real conflicts, false negatives, and false positives, under different signature schemes. The profiling setting is the same as §5.2. For each case, we compared the signatures of "`{P,PC}`" with the naïve signatures of "`PC`", considering only static locations. Table 4 shows the data. Overall, the signatures of "`{P,PC}`" significantly reduced false positives from the naïve scheme to the same level of the real conflicts, and they combined are an order of magnitude smaller than the total number of operations. False negatives are extremely rare.

The overall improvement of conflict analysis to Morpheus is shown in the "CA Improvement" row of Table 3. Conflict analysis improves the performance of POS by up to 241.94% in `ra-1`, and the average improvement is 64.82%.

Table 4. Results of conflict analysis with different signature schemes on `gproc-1` and `mnesia-2`. "Hit-ratio" denotes the error-finding performance of Morpheus without and with conflict analysis. "FNs" denotes falsely ignored conflicting operations amortized by trials, and "FPs" denotes falsely explored non-conflicting operations amortized by trials.

Case	Operations	Conflicts	Hit-ratio
`gproc-1`	6593.39	325.59	0.0031
Scheme	FNs	FPs	
`PC`	0.09	1192.16	0.0064 (206%)
`{P,PC}`	0.54	526.27	0.0106 (342%)
`mnesia-2`	10018.80	628.25	0.0117
`PC`	0.18	4612.02	0.0174 (149%)
`{P,PC}`	1.13	1442.74	0.0253 (216%)

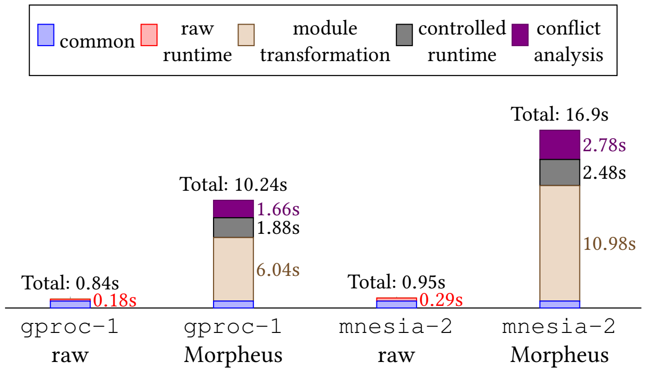


Figure 9. The real-time performance of Morpheus on `gproc-1` and `mnesia-2` over 100 trials. The numbers next to the bars show the runtime decomposition (except common).

5.4 The Real-time Performance of Morpheus

The runtime overhead of Morpheus comes from three aspects: (1) transforming the modules on the fly, (2) controlling and executing the operations, and (3) analyzing the trace after each trial. We evaluated the real-time performance on `gproc-1` and `mnesia-2` over 100 trials. We also measured the runtime of an empty test without Morpheus as the common time of a test. The results are shown in Figure 9. Overall, we observed nearly two orders of magnitude of overhead due to the heavy runtime manipulation, common in concurrency testing with complete interleaving control. Module transformation took the majority of the overhead, which we believe can be amortized by reusing the transformed modules across trials. We evaluated a mini-benchmark of sending messages in 1,000,000 round-trips, showing that Morpheus's runtime (94.02s) is 51.54% more efficient than Concuerror's (142.48s).

6 Related Work

Randomized concurrency testing. Burckhardt et al. proposed PCT [13] as the first randomized testing algorithm that leverages the simple root causes of realistic concurrency errors. Ozkan et al. later proposed PCTCT [42], adapting PCT into the context of general distributed systems. Concurrent to our work, Ozkan et al. recently proposed taPCT [43] that also identifies conflicting operations. Unlike Morpheus, it relies on discovering *all* conflicting operations in pre-processing, an assumption often unmet by real-world programs such as the ones we studied. Sen proposed RAPOS [45] that leverages partial-order semantics for randomized testing as a heuristics, but it does not provide probabilistic guarantees. Jepsen [26, 33] tests real-world distributed systems by exploring their behaviors under random network partitions. For Erlang programs, QuickCheck/PULSE [16] provides randomized concurrency testing with a simple strategy akin to random walk. Morpheus leverages the partial order sampling [57] in Morpheus to further exploit the small scope hypothesis.

Systematic concurrency testing. A number of systematic testing tools have been proposed to apply model checking on implementations of concurrent systems, including CHESS [38], dBug [47], MoDist [53], DeMeter [23], SAMC [27], and FlyMC [32]. Some of them are for Erlang programs, such as McErlang [20] and Concuerror [15]. They explore the concurrency of a system and verify the absence of errors by exhaustively checking all possible interleavings. The major limitation of systematic testing is its ineffectiveness to handle the immense interleaving spaces to find even simple errors. Partial-order reduction techniques [7, 8, 11, 19, 21] alleviate this problem by skipping equivalent interleavings. For real-world systems, however, the interleaving spaces after reduction are often still intractable for systematic testing.

Program analysis for finding concurrency errors. There is a large body of work detecting concurrency error using static analysis [39, 50], dynamic analysis [18, 31, 40], and the combination of two [29, 30]. None of them alone can be effectively applied to real-world distributed systems with both coverage and precision.

Other approaches and language support for concurrency testing. Coverage-driven concurrency testing [51, 56] leverages relaxed coverage metrics to discover rarely explored interleavings. Directed testing [44, 46] focuses on exploring specific types of interleavings to reveal targeted errors such as data races and atomicity violations. Some programming languages, for example, Mace [25] and P# [17] provide first-class support for building concurrent systems with high-level primitives, where our techniques can easily apply.

7 Conclusion

We have presented Morpheus, an effective concurrency testing approach for real-world distributed systems in Erlang. Morpheus leverages POS to exploit the simple root causes of concurrency errors, targets the high-level concurrency for systems written in Erlang to focus on the protocol-level errors, and introduces conflict analysis to further eliminate redundant explorations of reordering non-conflicting operations. Our evaluation showed that Morpheus effectively found protocol-level errors in popular real-world distributed systems in Erlang.

Acknowledgments

We thank the anonymous reviewers for their helpful feedbacks. This work was in part supported by NSF grant CNS-15-64055; ONR grants N00014-16-1-2263 and N00014-17-1-2788; a JP Morgan Faculty Research Award; a DiDi Faculty Research Award; a Google Cloud grant; and an Amazon Web Services grant.

References

[1] Amazon simpledb. https://aws.amazon.com/simpledb.

[2] Apache cassandra. http://cassandra.apache.org.

[3] Cassandra-6023. https://issues.apache.org/jira/browse/CASSANDRA-6023.

[4] Erlang programming language. https://erlang.org.

[5] The go programming language specification. https://golang.org/ref/spec.

[6] Why whatsapp only needs 50 engineers for its 900m users. https://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/.

[7] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 373–384, New York, NY, USA, 2014. ACM.

[8] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 526–543, Cham, 2017. Springer International Publishing.

[9] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 51–68, Berkeley, CA, USA, 2018. USENIX Association.

[10] Stavros Aronis, Scott Lystig Fritchie, and Konstantinos Sagonas. Testing and verifying chain repair methods for corfu using stateless model checking. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 227–242, Cham, 2017. Springer International Publishing.

[11] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–248, Cham, 2018. Springer International Publishing.

[12] Radu Banabic and George Candea. Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[13] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM.

[14] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core erlang 1.0.3 language specification. 2004.

[15] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 154–163, March 2013.

[16] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in erlang with quickcheck and pulse. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 149–160, New York, NY, USA, 2009. ACM.

[17] Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 154–164, New York, NY, USA, 2015. ACM.

[18] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[19] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[20] Lars-Åke Fredlund and Hans Svensson. Mcerlang: A model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 125–136, New York, NY, USA, 2007. ACM.

[21] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[22] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[23] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278, New York, NY, USA, 2011. ACM.

[24] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[25] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[26] Kyle Kingsbury. Jepsen, 2016-2019.

[27] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 399–414, Berkeley, CA, USA, 2014. USENIX Association.

[28] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 517–530, New York, NY, USA, 2016. ACM.

[29] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 677–691, New York, NY, USA, 2017. ACM.

[30] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. Fcatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 419–431, New York, NY, USA, 2018. ACM.

[31] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[32] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 20:1–20:16, New York, NY, USA, 2019. ACM.

[33] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.*, 2(POPL):46:1–46:24, December 2017.

[34] Dahlia Malkhi, Mahesh Balakrishnan, John D. Davis, Vijayan Prabhakaran, and Ted Wobber. From paxos to corfu: A flash-speed shared log. *SIGOPS Oper. Syst. Rev.*, 46(1):47–51, February 2012.

[35] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[36] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, OSDI '02, pages 75–88, Berkeley, CA, USA, 2002. USENIX Association.

[37] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[38] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[39] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[40] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[41] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[42] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.*, 2(OOPSLA):160:1–160:28, October 2018.

[43] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[44] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 25–36, New York, NY, USA, 2009. ACM.

[45] Koushik Sen. Effective random testing of concurrent programs. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 323–332, New York, NY, USA, 2007. ACM.

[46] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[47] Jiri Simsa, Randy Bryant, and Garth Gibson. dbug: Systematic evaluation of distributed systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[48] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using controlled schedulers: An empirical study. *ACM Trans. Parallel Comput.*, 2(4):23:1–23:37, February 2016.

[49] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 7–7, Berkeley, CA, USA, 2004. USENIX Association.

[50] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

[51] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 221–230, New York, NY, USA, 2011. ACM.

[52] Ulf Wiger. Github - locks, 2017.

[53] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[54] Junfeng Yang, Can Sar, and Dawson Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 10–10, Berkeley, CA, USA, 2006. USENIX Association.

[55] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.

[56] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 485–502, New York, NY, USA, 2012. ACM.

[57] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 317–335, Cham, 2018. Springer International Publishing.

A Artifact Appendix

A.1 Abstract

The artifact contains all experiment code and data for our evaluation (Section 5). The experiments were running on workstations running Ubuntu 16.04, with each of 64 GB of RAM and 2 Intel Xeon E-2640 CPUs. To run all our experiments, we provide Docker containers based on Ubuntu 18.04.

A.2 Artifact check-list (meta-information)

Obligatory. Use just a few informal keywords in all fields applicable to your artifacts and remove the rest. This information is needed to find appropriate reviewers and gradually unify artifact meta information in Digital Libraries.

- **Algorithm:** Partial order sampling with conflict analysis.
- **Program:** Customized benchmarks based on open-sourced software, included in the artifact.
- **Run-time environment:** Linux supporting Bash, Python 3 with `tabulate` module, and Docker.
- **Hardware:** `x86_64` architecture; 16 GB or more RAM recommended.
- **Metrics:** Execution time, error-detection performance (# of errors vs # of trials), complexity profiles of the benchmarks.
- **Output:** Raw logs - scripts included to generate tables from the logs.
- **Experiments:** Container based experiments using scripts/-makefiles. Runtime performance could vary based on hardware. Other results could also vary in a small range since the algorithms are randomized.
- **How much disk space required (approximately)?:** 15 GB minimal - may need more for copying data from containers. 50 GB recommended.
- **How much time is needed to prepare workflow (approximately)?:** 1~2 hour to build the docker containers.
- **How much time is needed to complete experiments (approximately)?:** ~5000 hours - can be largely parallelized. We used ~1 week with effectively ~30 CPUs.
- **Publicly available?:** Morpheus is open-sourced at https://github.com/xinhaoyuan/morpheus and https://github.com/xinhaoyuan/firedrill (for the POS implementation)
- **Code licenses (if publicly available)?:** Apache 2.0
- **Data licenses (if publicly available)?:** Apache 2.0
- **Archived (provide DOI)?:** http://doi.org/10.17605/OSF.IO/69H75

A.3 Description

A.3.1 How delivered

Experiment code. `morpheus-asplos-2020-artifact.tgz` in http://doi.org/10.17605/OSF.IO/69H75 contains:

- The specifications of the Docker containers for running the experiments.
- The Morpheus and Concuerror benchmark code used to evaluate our system.
- Modified Concuerror supporting randomized testing algorithms, i.e. random walk and POS.

Preparing the docker image would require a few hours and ~1 GB of storage. Running the experiments in the containers would produce uncompressed data. See below for their size.

Original data used by the paper. Available at http://doi.org/10.17605/OSF.IO/69H75 as `morpheus-asplos-2020-data.tgz`. There are ~13 GB of the uncompressed data.

A.3.2 Hardware dependencies

x86_64 architecture. 16 GB or more RAM recommended. Experiments can be largely parallelized.

A.3.3 Software dependencies

Linux with support of Bash, Python 3 with `tabulate` module, and Docker. `tabulate` can usually be installed by command

```
pip3 install tabulate
```

Our container specifications handle other software dependencies.

A.4 Installation

Download `morpheus-asplos-2020-artifact.tgz` from http://doi.org/10.17605/OSF.IO/69H75. Extracting it will create a directory `artifact-evaluation`. Enter the directory and run `build-docker-images.sh` to build two containers for Erlang/OTP with version 20 and 21, patched with Morpheus support. This results in Docker images `morpheus-benchmarks:otp-{20,21}` for the experiments under the two Erlang/OTP versions.

A.5 Experiment workflow

Our experiments run in Docker containers. Each experiment follows the workflow of three steps:

1. Create and run the docker containers corresponding to each experiment. The experiment is specified as jobs in `Makefile` in the containers. Thus the commands for the experiment are in the form of

   ```
   docker run --name [CONTAINER-NAME] make \
       [ARGS] -j [PARALLELISM]
   ```

   You may change `[PARALLELISM]` accordingly to the number of CPUs (cores) in your machines.
2. After the containers terminate, copy their output (logs) from the containers to the host using

   ```
   docker cp [CONTAINER-NAME]:[OUTPUT-DIR] \
       [HOST-DIR]
   ```
3. Use `artifact-evaluation/get_tables.py` script to generate tables from the data.

A.6 Evaluation and expected result

A.6.1 Comparison with Concuerror (Section 5.1)

Experiments here compare systematic concurrency testing implemented in Concuerror [15] with randomized testing algorithms, namely random walk and POS.

Use the following command to run the experiments in a container:

```
docker run --name exp-concuerror \
    morpheus-benchmarks:otp-20 make -C \
    concuerror-tests all -j 10
```

This would take ~15 CPU hours. When it is finished, copy the data from the container:

```
docker cp \
    exp-concuerror:/root/concuerror-tests/data \
    [HOST-DIR]
```

`[HOST-DIR]` could be e.g. `data/concuerror`. Please make sure the directory `data` exists before copying.

The part of the original data is in `data/concuerror` of the data archive.

A.6.2 Error-detection performance (Table 3)

Experiments here compare the error-finding performance of Morpheus with other randomized concurrency testing algorithms.

The experiments require two containers of different Erlang/OTP versions, using the following two commands to run them.

```
docker run --name exp-main-20 \
    morpheus-benchmarks:otp-20 make -C \
    morpheus-tests data-main-20 -j 10
```

and

```
docker run --name exp-main-21 \
    morpheus-benchmarks:otp-21 make -C \
    morpheus-tests data-main-21 -j 10
```

This is the most time-consuming part of the experiments - it would take up to 4900 CPU hours, assuming each trial taking up to 20 seconds. When they are finished, copy the data from the containers to local directory:

```
docker cp \
    [CONTAINER-NAME]:/root/morpheus-tests/data \
    [HOST-DIR]
```

`[CONTAINER-NAME]` is one of `exp-main-{20,21}`, and `[HOST-DIR]` could be, e.g. `data/main/otp-{20,21}`, respectively. Please make sure the directory `data/main` exists before copying.

The part of the original data is in `data/main` of the data archive.

A.6.3 Conflict analysis (Table 4)

Experiments here evaluate the performance of conflict analysis and its improvement to error-finding on two specific test cases.

Using the following command to run the experiments:

```
docker run --name exp-ca \
    morpheus-benchmarks:otp-20 make -C \
    morpheus-tests data-ca -j 10
```

This would take ~200 CPU hours to finish. When it is finished, copy the data from the containers to local directory:

```
docker cp exp-ca:/root/morpheus-tests/data \
    [HOST-DIR]
```

`[HOST-DIR]` could be, e.g. `data/ca`. Please make sure the directory `data` exists before copying.

The part of the original data is in `data/ca` of the data archive.

A.6.4 Runtime performance (Figure 9)

Experiments here analyze the runtime overhead of Morpheus compared with raw executions in Erlang.

Using the following command to run the experiments:

```
docker run --name exp-rt \
    morpheus-benchmarks:otp-20 make -C \
    morpheus-tests data-rt -j 10
```

This would take one to two CPU hours. When it is finished, copy the data from the containers to local directory:

```
docker cp exp-rt:/root/morpheus-tests/data \
    [HOST-DIR]
```

`[HOST-DIR]` could be, e.g. `data/rt`. Please make sure the directory `data` exists before copying.

The part of the original data is in `data/rt` of the data archive.

A.6.5 Expected results

Run `get_tables.py` on the reproduced data or original data (in `morpheus-asplos-2020-data.tgz`) to get the tables of the paper (modulo format changes). Note that if you copy data into directories other than `data/concuerror`, `data/main/...`, `data/ca`, and `data/rt`, you need to use arguments `--data-main`, `--data-ca`, etc. with `get_tables.py`. Use argument `--help` to see the help information of the arguments.

We include `expected_tables.txt` in the experiment archive for the expected results from the original data. For reproduction, runtime performance could vary based on hardware performance. Other results could also vary in a small range since the algorithms are randomized.

A.6.6 Running smaller scale experiments

It is possible to run the experiments in a smaller scale by modifying the following files in `artifact-evaluation/exp-files/`:

- `concuerror-tests/Makefile`
- `morpheus-tests/Makefile`

There are variables `REPEAT` and `instance_repeat`, controlling the number of trials to perform on each test case and algorithm. After modifications to the variable, you would need to rebuild the Docker image again using the shell script.

Note that if you change `instance_repeat`. You also need to pass `-r` argument in `get_tables.py` to 10 (the number of instances in `Makefile`) times the changed value, such as:

```
./get_tables.py -r 1000
```

if you set `instance_repeat` to 100.

A.7 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging