Dara The Explorer: Coverage Based Exploration for Model Checking of Distributed Systems in Go

by

Vaastav Anand

BSc. Computer Science, The University of British Columbia, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Dara The Explorer: Coverage Based Exploration for Model Checking of Distributed Systems in Go

submitted by **Vaastav Anand** in partial fulfillment of the requirements for the degree of **Master of Science**
in **Computer Science**.

Examining Committee:

Ivan Beschastnikh, Computer Science
Supervisor

Margo Seltzer, Computer Science
Supervisory Committee Member

Abstract

Distributed systems form the backbone of modern cloud systems. Failures in distributed systems can cause massive losses in the form of service outages and loss of revenue impacting real users of systems. Thus, it is imperative to find bugs in distributed systems before they are used in production systems.

However, debugging distributed systems continues to elude us. Use of abstract modelling languages such as TLA+, PlusCal, Coq, and SPIN that check the correctness of models of distributed systems have become popular in recent years but they require a considerable amount of developer effort and do not necessarily find all the bugs in the implementation of the system. Model checkers that explore all possible executions of the implementation of a distributed system suffer from state space explosion, rendering them impractical as they are inefficiently scalable. To alleviate this, we propose Dara, a model checker designed for Go systems that uses a novel coverage-based strategy for ordering exploration of paths in the state space of the system according to the amount of code covered across nodes. Dara can find and reproduce concurrency bugs in go systems.

Lay Summary

Distributed systems are the backbone of modern cloud systems. However, these systems are complex and hard to build. Failures in these systems can lead to loss of revenue in addition to service outages impacting users. Due to their complexity, debugging these systems has been a continuous challenge for developers. There has been recent interest in developing techniques for finding bugs in systems before they are placed in production. One such technique has been model checking, in which all potential executions of a system are explored to find bugs. However, due to the complexity of these systems, there are a large number of paths that need to be explored. In this work, we propose a novel technique for exploring these paths based on maximizing the amount of code that is being covered during the execution of these paths.

Preface

All work presented henceforth was conducted in the Systopia lab, formerly known as the Networks, Systems, and Security (NSS) Lab, in the Department of Computer Science at the University of British Columbia, Vancouver Campus. This thesis is an original, unpublished work by Vaastav Anand, written under the supervision of Ivan Beschastnikh. A short, preliminary version of the work placed second in the Mircosoft Student Research Competition at the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering in November 2018.

Table of Contents

List of Tables

List of Figures

Acknowledgments

This work was supported by the Huawei Innovation Research Program (HIRP), Project No: HO2018085171. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 2014-04870.

I would like to thank Mr. Stewart Grant for helping with the initial hacking of the go runtime. I would like to thank my advisor, Dr. Ivan Beschastnikh for his invaluable mentorship and advice. I would also like to thank Dr. Margo Seltzer and Dr. Jonathan Mace for being constant sources of advice and encouragement throughout my M.Sc. degree.

Finally, I thank my family, my mom, my grandmother, and my uncle for constantly encouraging me to chase my dreams and supporting me emotionally through this beautiful journey. I would not be able to be where I am without their constant love and support.

In loving memory of my late grandfather, Mr. Satish Kumar Anand.

Chapter 1

Introduction

Debugging distributed systems has continued to bedevil developers. Despite rigorous testing efforts , an increasing number of systems are in the news due to complex bugs sneaking into production systems. For example, in 2017 a bug in Amazon S3 caused 150 million dollars in damage for those companies that rely on Amazon AWS [8]. Such incidents are becoming increasingly common.

State-of-the-art techniques for ensuring the correctness of distributed systems are widespread, ranging from traditional testing to formal verification, using techniques such as interactive provers and model checking. Interactive provers mathematically prove systems correct by constructing machine-verifiable proofs based on deductive reasoning. These provers include Coq, Hal, or Isabelle, and typically come with powerful specification languages targeting verification. Projects such as Verdi, IronFleet, and Chapar [19, 29, 40] demonstrated the utility of these techniques by developing fully verified systems. These proofs however require significant expertise and effort.

Model Checking (MC) is a formal methods technique that exhaustively tests the system, rather than using deduction to prove it correct [10, 21, 26] Traditionally model checking finds bugs in abstract models of the system, but not in implementations. That is, model checking explores many program behaviours and checks if the program behaviours meet the user-provided specification of the system. However, traditional model checking is capable of finding bugs only in a given specification and has no way of finding bugs in real implementations.

A new line of work in "concrete model-checkers" (concrete MCs) [28, 31, 42] extends conventional model-checking to implementations, by searching for failures, crashes, and violations of user-defined properties. Concrete MCs directly use the system implementation as the model while interposing on the network as well as the operating system layers to control delivery of messages as well as scheduling of threads. Use of a concrete MC eliminates the need for a developer to create and maintain a separate, abstract, model of their system that decreases developer effort and eliminates bugs that developers may introduce as part of composing the model. Additionally, a concrete MC has no false positives: a bug found using a concrete MC is a bug in the real implementation of the system and thus can be reproduced.

However, these concrete MCs suffer from state space explosion. They are unable to efficiently explore the state space of the system. This is because the number of reachable states grows exponentially as new components are added to the system [4]. As a result, most model checkers limit their exploration of the state space of a system or a model to some bounded depth. Thus, it is imperative for the model checkers to explore paths in the state space that are more likely to produce bugs.

But, designing and implementing a concrete model checker is hard as one must control various sources of nondeterminism in the distributed system while providing the ability to control thread scheduling and checking user-defined properties about the system. Moreover, rise in popularity of languages such as Go for building large distributed systems has further complicated the design process of concrete MCs. Specifically, designing a concrete MC for Go is complicated by the fact that Go has its own runtime for managing and scheduling threads, handling system calls, synchronization whilst providing language-level interfaces to applications for managing concurrency. A concrete MC for Go must control all these features to be able to correctly explore the state space of the system and find bugs. To the best of our knowledge, there does not exist a concrete MC for Go.

We present Dara The Explorer, a concrete MC designed for Go, that uses a novel coverage-based exploration strategy to explore paths that are more likely to produce bugs. The key insight behind the coverage-based exploration strategy is that the paths that cover a higher percentage of code are much more likely to result in bugs as a larger fraction of code is being executed through these paths than would

be through regular execution of the system. Additionally, Dara allows users to specify properties of the system as standalone go functions that can be executed and checked during the exploration phase to find property violations in addition to crashes and failures. However, for ease of debugging, finding a presence of a crash or a violation is not enough as one must be able to reproduce the bugs found consistently. To achieve this, Dara exports a replay engine that can deterministically replay a buggy schedule recorded during the exploration of the state space. Dara also provides some auxiliary tools for inspecting and analysing the buggy schedules.

To summarize, our contributions are as follows:

- A novel coverage-based exploration strategy to explore the state space of an implementation of a distributed system.
- A concrete MC designed and implemented for distributed systems in Go.
- Specification of system properties as executable, standalone go functions
- A suite of auxiliary tools for deterministically reproducing and understanding the cause of the bugs.

Chapter 2

Related Work

Dara's concrete MC for Go shares similarities with prior concrete MCs but it also makes different design decisions specifically to manage the Go runtime. Dara also provides coverage-based exploration strategies for optimizing state space exploration whereas a lot of the prior work has focused on reducing the state space by finding different strategies for finding redundant or symmetrical paths in the state space. Ultimately, Dara is a debugging tool and has strong roots in verification of systems and debugging of distributed systems.

Model checking of distributed systems. Model checking and model-based testing have been widely used in the past for finding bugs and property violations in actual implementations of distributed systems [2, 9, 17, 18, 23, 27, 28, 31, 32, 41, 42]. State-of-the-art model checkers, such as MODIST [42] and Chess [32], typically focus on testing, unmodified, distributed and concurrent systems, an approach that leads to massive state space explosion. DEMETER [18], built on top of MODIST, reduces the state space explosion when exploring unmodified distributed systems. DEMETER'S exploration algorithm separates out the system's global state from the local states of the system's components. It explores individual components in isolation, and dynamically extracts interface behaviour between components to perform global executions. Guerraoui & Yabandeh, introduce a local approach to model checking distributed systems where they remove the network state from the global state and focus on the remaining state, which is the required part for checking invariants [17]. Dara does not explore individual components in

isolation but rather explores different paths in the system based on the coverage of the whole system. Dara does maintain a separate local state for each node in the form of variable-value mappings for property checking and coverage information for each node. This local state is collated together from the various nodes to form the global state of the program which is then used for property checking and path exploration.

SAMC [28] offers a way to incorporate application-specific information in state-space reduction policies alleviating redundant interleavings of messages, crashes and reboots. FlyMC [31] exploits communication and state space symmetry to explore a larger number of states in shorter wall clock time. FlyMC specifically leverages various classes of node roles to explore only one representative interleaving for different nodes in that node class. Dara's coverage-based exploration strategies incorporate the overall application structure by incorporating code coverage across the nodes which is similar to the application-specific information incorporated by SAMC. The difference is that Dara's strategy focuses on capturing application structure whereas SAMC focuses on capturing information regarding ordering of crashes and messages that will help in reducing redundant states in the state space. Currently, Dara's coverage-based exploration does not take advantage of node roles in reducing the state space of the system. We believe that incorporating node role information in Dara's coverage-based exploration will increase Dara's efficacy.

MACEMC [23] is a model checker for distributed systems that finds possible liveness bugs. It combines bounded exhaustive search with random walks to distinguish between live, and possibly dead states. The same strategy was later used within MACEPC to identify possible latent performance bugs [24]. MACEMC and MACEPC can test systems written only in MACE, so they can not be used with real implementations of systems whereas Dara can be applied to any distributed system written in Go. However, checking liveness properties and performance properties remains a part of Dara's future work.

Verification of distributed systems. Formal methods and theorem provers have been successfully used to formally verify correctness of distributed protocols and systems [35, 44]. Amazon uses TLA+ [25] to verify its systems [33]. 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. There is still a

significant gap between the specification and implementation of systems leaving the system prone to implementation level bugs.

Recently, attempts have been made to generate verified implementations of distributed systems from specifications [19, 29, 40]. With Verdi [40], a developer provides an implementation of the system and proves the system correct in Coq using a simplified environment, which is then transformed into an equivalent implementation in a more robust and realistic environment using Verdi's verified transformers. IronFleet [19] requires developers to provide a Dafny specification and code. IronFleet verifies that the specification upholds all invariants and the implementation meets the specification. These frameworks require significant developer effort, while Dara requires minimal effort from the developer. Dara leverages the end-to-end tests already written by the developers to generate an input schedule to use as a starting point for exploration.

Bakst *et al* describe the canonical sequentialization of a system as a small set of representative executions of the system [3]. Their approach verifies the canonical sequentialization of a system instead of verifying the system. Dara meanwhile checks properties and finds crashes in implementations of systems.

Debugging distributed systems. Numerous debugging tools assist developers in finding bugs in distribution systems. Aspirator [43] finds bugs caused due to improper error handling in distributed systems. Jepsen [22] is a black box testing library powered with fault injection and generative testing to find bugs in systems that rarely manifest in the usual executions of the systems. DEMi [38] minimizes the length of bug traces and executions. Pip [37] allows users to specify expected behaviour and compare against actual behaviour to find structural and performance bugs. Dara provides a way for the developers to specify properties of the system as go functions which are then checked at runtime. Currently, Dara does not inject random faults but that is something we are looking at as future work.

Record and Replay tools such as D3S [30] and Friday [14] record a single execution of a distributed system and capture all nondeterministic events so that the execution can be replayed exactly. Dara's replay engine embodies this approach to allow developers to replay buggy execution traces recorded during the exploration of the state space of the system.

Fuzz Testing Fuzz testing is a technique for finding bugs by generating random

inputs in an application. Despite the great success of fuzz testing in finding bugs for single-threaded programs, there have been few efforts in developing fuzzing techniques for multi-threaded programs or distributed systems. MUZZ [6] is a greybox fuzzing tool for finding input-dependent and interleaving-dependent paths that result in crashes and concurrency bugs. MUZZ generates different inputs and different thread interleavings for finding concurrency bugs. Dara's approach is similar to that of MUZZ as Dara's coverage-based exploration strategies help guide the explorer in exploring different thread executions and interleavings in the system. However, Dara does not modify the original input whilst providing a dedicated property checker to users for specifying arbitrary properties about the system.

Chapter 3

Background

Dara is a concrete MC for Go that introduces novel coverage-based strategies for efficient exploration of a system's state space. However, before we describe the detailed design of Dara, we first introduce some key background information. Section 3.1 describes *model checking* and introduces the notion of concrete model checkers, the type of model checkers that work directly on implementations of systems instead of user-defined abstractions of systems. Section 3.2 provides an overview of the Go runtime and how certain language-specific features of Go are implemented in the Go runtime.

3.1 Model Checking

Model Checking is a verification technique to check whether a user-provided **model** of a system meets or satisfies a given correctness **specification** of the system. The model of a system could be a finite state machine where each state describes a state the system can be in, and the transition relationship between the states describes the legal sequence of steps that must happen for the system to transition from one given state to another. The correctness specification of the system is provided as a set of properties usually written in propositional temporal logic such as Linear Temporal Logic (LTL) [36]. Model checking verifies that a model of a system meets its specification by performing an exhaustive search over the state space of the system's model and checking whether the properties provided by the

specification are satisfied on every possible execution of the model. However, model checking is more useful than simply providing a binary answer of whether a model satisfies the specification or not. If a model does not satisfy the specification, the model checker provides a **counterexample**, that is a sequence of steps that the model checker executed to end up in a state where the specification was violated.

However, traditional model checkers have two key disadvantages: (i) the developer must learn a Domain Specific Language (DSL) to construct a model of the system, and if the system changes, then the model must be manually updated to reflect the change; (ii) the model may admit false positives, since it is not the actual system: a bug in the model may not correspond to a bug in the implementation. Also, the model checker may have false negatives (*i.e.* it may miss bugs that are present in the implementation). To eliminate false positives, false negatives, and to reduce developer effort, researchers developed concrete model checkers that work directly on the implementation of a system and do not require an additional abstract model of the system.

3.1.1 Concrete Model Checkers

Concrete MCs use the system's real implementation as the model. They have two primary advantages: the developer does not need to develop a separate, abstract model of their system (decreases developer effort and eliminates bugs that developers may introduce as part of composing the model), and by using the implementation directly, a concrete MC has no false positives: a bug found using a concrete MC is a bug in a concrete implementation and can always be reproduced. Concrete MCs have two key disadvantages: a very large (concrete) state space of the system under analysis and engineering complexity — few concrete checkers exist and the first concrete MC for distributed systems, MODIST [42], is relatively young.

For a concrete MC to be successful in model checking, it must at least successfully fulfill the following four requirements:

- Have control over scheduling of actions for every single execution unit (thread, goroutine, process) so that the concrete MC can have precise control over the thread interleavings and schedule the threads in an order desired by the concrete MC.

- Have control over the sending and delivery of messages over the network and via other inter-process communication channels so that the concrete MC can control and order the sends and deliveries with respect to each other to explore potentially different executions of the system.
- Have some way of collecting data and evaluating global properties for a distributed system so that the concrete MC can evaluate and find violations of user-defined properties about systems.
- Have control over the environment including the file system and environment variables so that the concrete MC can restore the environment before restarting the execution of the program to explore a new path in the state space of the system.

3.2 Go Runtime Background

We next detail features that are specific to the Go runtime and the language, which plays a significant role in the design and implementation of a concrete MC for Go.

GoRoutines The go runtime does application level thread management instead of kernel-level thread management. This allows Go programs to launch arbitrarily many "threads" in go. Each "thread" in go is called a goroutine. Unlike kernel threads, goroutines are scheduled cooperatively so there is no goroutine pre-emption unless the thread is going to sleep or has made a blocking call. The stack of a goroutine starts small and grows/shrinks during execution. New goroutines are created by using the keyword "go foo()" where "foo" is the function to be executed by the newly created goroutine. This keyword is just syntactic sugar and it gets compiled into a function call to the "newproc" runtime function which creates the new goroutine and places it on the ready queue.

Runtime Initialization When a go binary is executed, the first function that gets called is the runtime.init function which in turn calls the runtime.main function. By default this function is run in the goroutine with ID 1. The runtime main function is responsible for initializing the runtime as well as setting up main.main, the user defined main function. In addition to this main goroutine, the runtime creates two

other goroutines. The goroutine with ID 2 enables the garbage collection whereas the goroutine with ID 3 runs the finalizer code.

GoRoutine Scheduling The Go runtime is responsible for managing goroutines and allocating them on kernel level threads for execution. The Go runtime follows a M:N goroutine/threading model, with $M \geq N$, such that M goroutines are scheduled cooperatively on N threads. The value of the environment variable, *GOMAXPROCS*, which is by default the number of CPU cores, determines the maximum number of OS threads that can be running concurrently at any given time. Threads blocked on a system call do not count towards the total number of threads running.

Sleeping GoRoutines To put itself to sleep, a goroutine makes a function call to time.Sleep, which traps into the runtime, where the goroutine installs a timer for itself and puts itself onto waiting mode after which a different goroutine, if any, is chosen to be scheduled. When a timer is installed for the first time for any goroutine in the runtime, the runtime creates a new goroutine which executes the the timer process called timerproc. If there already exists a timerproc, then the runtime does not create another timerproc. The runtime then installs the newly created timer on a blocked queue managed by the timerproc. This timerproc is responsible for firing off the timers from this queue that have been installed by other goroutines. Just like the garbage collector and finalizer goroutines, the timerproc goroutine is a runtime internal goroutine.

Channel-based Communication Channels are pipes that connect concurrent goroutines allowing the goroutines to send and/or receive values to and from another goroutine. Channels between goroutines could either be one-way or two-way and they could be buffered or unbuffered. To send a value into a channel, a goroutine uses the "channel <-" syntax and to receive a value from a channel, a goroutine uses the "<- channel" syntax. Both of these compile into function calls in the go runtime where the runtime removes the values from a channel's send queue to the channel's receive queue. By default, sends and receives block until both the sender and receiver are ready which allows for strong synchronization between goroutines.

Select statements The select statement lets a goroutine wait on multiple communication operations. It blocks until one of its cases can run, then it executes that case. If multiple cases are ready then it chooses a case at random to execute. The select statement is also implemented in the go runtime where it is implemented by

```
1 ...
2 resultChannel := operation()
3 select {
4     case result := <-resultChannel:
5         log.Println("Operation finished with result", res)
6     case <- time.After(5 * time.Second):
7         log.Println("Operation timed out")
8 }
9 ...
```

Figure 3.1: Example of timeout code in Go

the function "selectgo" which returns the index of the case that needs to be executed based on which case unblocks first.

Timeouts Timeouts are an integral part of distributed systems. Since Go was primarily designed to be a language for building distributed systems, Go has its own idiomatic way of specifying timeouts using channels and "select". To ensure that an operation has a bounded execution time, the developer wraps the operation around with a select statement such that one of the cases of the "select" statement is the operation returning a result via a channel and the other case is a timer installed to fire after a specific amount of time which acts as the time bound. To fire off the timer, the go runtime installs a timer to go off at the scheduled time and returns a channel to the application to which a notification will be sent via a message. Figure 3.1 shows a snippet of code that demonstrates the Go way of specifying timeouts in Go. In this example, the "time.After" function call returns a channel that receives a message from the go runtime after 5 seconds indicating a timeout.

Chapter 4

Design

4.1 Overview

We present an overview of how a user interacts with Dara in the context of discovering a Go non-determinstic bug caused by the non-deterministic behaviour of select [39].

Figure 4.1a shows the snippet of code that contains the non-deterministic bug which we discuss in the context of Dara. In the code snippet, the loop at line 4 executes a heavy function *f()* at line 7 and then waits to receive a stop message on the stop channel, *stopCh*, at line 10. However, if the message does not arrive before the ticker installed at line 3 fires at line 12, then the function *f()* is executed again. Thus, the loop at line 4 executes until it receives a stop message on *stopCh* at line 9 and executes a return from the function *foo* at line 11. However, if the stop channel, *stopCh*, receives the message at the same time as the ticker ticks, then there is no guarantee which case will be chosen by the *select* to be executed. In such scenarios, *select* will nondeterministically choose the case to be executed. If *select* chooses case 2, *i.e.* the timer case, then *f()* will be executed unnecessarily one extra time than intended by the user.

The user wants to ensure that the function f does not get executed when a message has been delivered on the *stopCh* channel. Thus, the user writes a property file as shown in Figure 4.1b which checks the number of deliveries on the stop channel, *stopCh*, is equal to zero. The user then manually instruments the source

```
1  func foo() {
2  +    runtime.ReportBlockCoverage("block1")
3       ticker := time.NewTicker()
4       for {
5  +        runtime.ReportBlockCoverage("block2")
6  +        runtime.DaraLog("LogID1", "Recvs", runtime.NumDeliveries(stopCh))
7           f()
8           select {
9           case <-stopCh:
10 +            runtime.ReportBlockCoverage("block3")
11              return
12          case <-ticker:
13 +            runtime.ReportBlockCoverage("block4")
14        }
15      }
16 }
```

(a)

```
1  //NoRecvs
2  //Recvs
3  func NoRecvs(Recvs int) bool {
4      return Recvs == 0
5  }
```

(b)

Figure 4.1: (a) Code with a non-deterministic bug due to select. The lines in blue are added by the instrumenter for reporting coverage information and the line in red is added by the user to report values of variables needed by the property checker to check properties provided by the user in (b), the user-provided property file.

code to capture the value of the number of successful deliveries of messages to the stop channel, *stopCh*, just before the function *f* is executed. However, the user wants the property to hold just before the function *f* is executed and not after the execution has returned from function *foo*. Thus, the user adds manual instrumentation at line 6, shown in red, in Figure 4.1a to capture the value of number of successful deliveries on the stop channel using Dara's special runtime function, *runtime.NumDeliveries* to update the value of the *Recvs* variable that is used by the property checker for checking the specified property. This function returns the total number of successful deliveries on a given channel regardless of whether the application might have read the message from the channel or not. So, if the channel has successfully received a message but the application has not read the message then the function will return

the value 1 instead of 0 as a message would have been successfully received on the channel. Note that the user does not add any manual instrumentation just before the return at line 11 to update the value of the *Recvs* variable in the property as that would update the value of *Recvs* to be 1 and cause the property checker to find a false positive. Not updating the *Recvs* variable ensures that the value of *Recvs* used by the Property Checker is the value logged just before the function f is executed. The property checker checks the properties for every logging statement that updates values of the variables.

The user then uses Dara's command line application, the *Overlord*, to instrument the source code with coverage information. The *instrumenter* adds function calls to Dara's runtime function, *runtime.ReportBlockCoverage*, at the start of every basic block in the code. The added function calls by the instrumenter are shown in blue in Figure 4.1a at lines 2, 5, 10, and 13. After instrumenting the source code, the user uses the Overlord to record an execution of the system using a test driver program to exercise the system. As part of the recorded execution, the basic blocks covered by each goroutine is also recorded and stored in the saved schedule of the execution. This information is used by the explorer for guiding the exploration process.

The user then uses the overlord once again to start the exploration process. The user provides a configuration file that contains the location of the recorded schedule, the test driver program, and configuration options for the explorer such as the exploration strategy, the maximum number of paths to be explored in the state space, and other configuration options. The explorer then explores various interleavings of goroutines and timer events to find paths which contain a violation of the property provided by the user. In this case, the explorer finds one path where the property is violated.

4.2 Concrete Model Checker

Figure 4.2 shows Dara's Concrete MC design. The blue nodes are user-visible and the orange nodes are internal to Dara and not exposed to the user. The overlord serves as the starting point for the users for working with Dara. The user instruments the source code of the distributed application using the Instrumenter via the overlord followed by launching the Dara's model checker via the overlord. Each node in

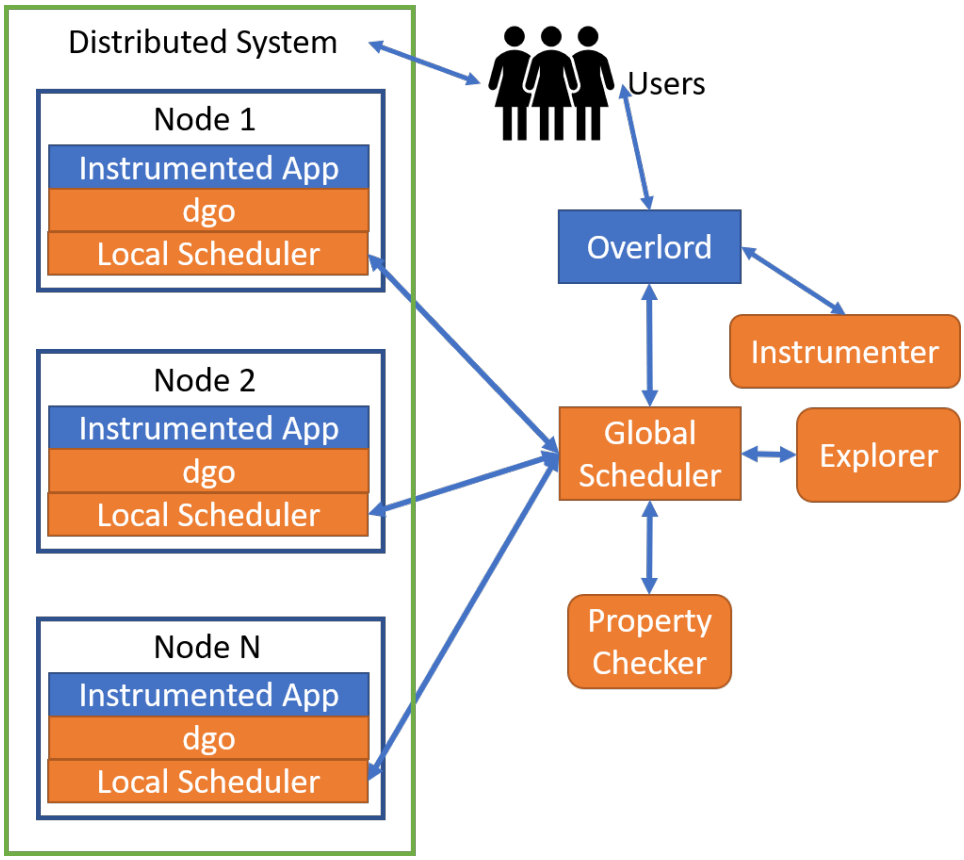


Figure 4.2: The architecture of Dara's concrete model checker

the distributed system under test runs an instrumented version of the source code. The application code is managed by our modified go runtime, called **dgo** which captures system calls, coverage information, and user-defined log statements for property checking. At each node, Dara installs a local scheduler, which controls the actions executed by the node via controlling the scheduling of goroutines and firing off timers. Each local scheduler communicates the data collected by dgo and the state and status of every goroutine at the node to the global scheduler. The global scheduler relays the collected information to the property checker for finding violations and to the explorer for deciding the next step of the execution. The Global Scheduler relays the action decided to be executed by the explorer to the local scheduler of the node where the execution must be executed. If the explorer

decides to finish exploration or to start exploring a new path, the global scheduler communicates with the overlord to kill the current execution of the system or to restart the execution of the system.

4.2.1 Controlling Nondeterminism

Model Checking real systems requires complete exploration of the state space of the system to make claims about the system's correctness. Any source of non-determinism can prevent bugs from manifesting or from being deterministically replayable, making our model checker unsound. Furthermore, failing to explore all non-deterministic events leads to incomplete checking. Here we tabulate sources of non-determinism and how our concrete model checker deals with them.

Ideally we achieve a deterministic execution by controlling every instruction so that any deviation of instruction or value could be monitored and considered as a new state. However, instruction level deterministic execution is slow and is generally not required. Thus, we follow a pragmatic approach focused on eliminating *sufficient* non-determinism from the system to guarantee that we can consistently find the same bug during the replay of a counterexample trace. There are various sources and types of non-determinism in a distributed system, each of which are specified below in detail.

Concurrency and Thread Interleaving: Concurrency can give rise to race conditions, which can lead to non-deterministic behaviour in the system. Similarly, thread interleavings can cause the system to execute in an incorrect manner giving rise to heisenbugs. To eliminate the non-deterministic effects of concurrency and thread interleaving, global events are executed in a serialized fashion with only a single goroutine executing at a given point of time across the system.

Random Library: Randomness in the system can be a huge stumbling block for deterministic execution as effected variables may effect the control flow of the system. Non-determinism due to the usage of the random library is eliminated by modifying Go's rand library such that the seed value is recorded during the system's initial execution and then reused during exploration. This ensures that the random values used during execution are the same as those used during all the runs.

Time Library Similar to the use of the random library, the use of the time library

can cause the system to take different execution paths than the one intended by the schedule. During exploration, the global scheduler maintains a virtual clock which controls whether timer events or goroutines that were put to sleep can be enabled so that the explorer can choose these actions to be executed. Since our virtual clock behaves like a logical clock without relying on real time, it also serves as an optimization during exploration for executing paths where timeouts or sleeps need to be executed. The use of a virtual clock ensures that all the time-dependent actions such as timer events and scheduling of a sleeping thread follow the strict ordering dictated by the real time of these actions. Moreover, Dara interposes on the time functions in the standard library to provide the application with a real time that is consistent with the other time related actions at that node.

Environment Changes in the environment during exploration can cause the system to execute different paths than the ones intended by the explore. To prevent the environment from having any uncontrolled impact on the exploration, each path in the system must be explored under the same environment. However, capturing the full system environment is out of scope for Dara. Instead, Dara requires the user to provide a clean up script that is executed before every restart of the system to restore the environment to what it should be during the execution of the system.

Network Randomness To deal with non-determinism due to the network such as reordered messages, message drop, or random delays, Dara runs every single node on the same machine as the global scheduler to minimize the amount of randomness that the network could possibly inject during the execution of the program. Dara indirectly controls different send and receive orderings of messages by controlling the scheduling of goroutines executing the respective send and receive code for messages. Dara also explores message timeouts by firing the timers at the receiving node before a message can be received by the particular node. However, Dara does not detect re-ordering of messages on the wire nor does it explore schedules where messages can be arbitrarily dropped.

4.2.2 Overlord

The overlord serves as the command centre for the user. The overlord interacts with Dara's instrumenter as well as the global scheduler. The overlord expects a

configuration file as well as a command (instrument, explore, replay, record) from the user as input that provides the location of the source code and the location of any custom build and run scripts, in the local file system, that is required for building and running the system. At the behest of the user, the overlord can instrument the system using the instrumenter or set up the model checker to either record an execution, replay a previously recorded schedule, or explore the state space of the system to find bugs. The overlord also maintains a communication link with the global scheduler to carry out system restarts or shutdown.

4.2.3 Instrumenter

Dara's instrumenter is a static analysis tool that rewrites the Abstract Syntax Tree (AST) of the source code of the system for reporting coverage to **dgo**. Dara's Instrumenter instruments the source code to report coverage information to **dgo** as to which basic blocks were executed by the goroutine. A *basic-block* is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit [1, 20]. The instrumenter walks the AST of every file in the source code and adds the relevant function call at the start of each basic block to report coverage information to **dgo**. The instrumenter instruments multiple files at a time and annotates the source code such that coverage information can be collected and correctly reported to dgo. As a side effect of walking through the AST of every file in the source code, the instrumenter knows about every single basic block in the source code. The instrumenter generates a "blocks" file which contains the list of every block in the source code. This file can be used by our auxiliary tools for generating comprehensive coverage reports.

Reporting Coverage to dgo To report coverage to **dgo**, at the start of every block in the source code, a function call to the function "runtime.ReportBlockCoverage" is added with the argument to the function call being the unique ID of the block. This function call traps into **dgo** where the runtime maintains a data structure mapping the block ID to a count of the number of times the block was executed. The function call serves as a proxy for block execution. This data structure is then forwarded by the local scheduler to the global scheduler. When the runtime schedules a new goroutine, the runtime wipes out the data structure so that the data structure only

contains coverage information from the newly selected goroutine's execution.

4.2.4 Global Scheduler

The global scheduler is the central piece of Dara as it connects to the various local schedulers, the overlord, property checker, and the explorer. It collates and collects the information from all the local schedulers and forwards the information to the property checker for property checking and the explorer for generating the next action. Once the explorer has selected the next action, the global scheduler contacts and commands the local scheduler for the node at which the action must be scheduled. Once the exploration is over or requires a restart, the global scheduler contacts the overlord for restarting or shutting down the system.

Virtual Clock The Global Scheduler implements a virtual clock for each node to ensure that time-related actions are consistent in a node. For *e.g.*, a goroutine that was put to "sleep" should not be scheduled before a timer fires if the timer would go off before the goroutine wakes up in real time. The virtual clock ensures that the actions at a node do not violate the real time order of actions. To ensure this, the global scheduler maintains a virtual clock for every node such that the clock is only fast-forwarded if the explorer decides to execute a previously sleeping goroutine or fire off a timer. In such cases, the global scheduler fast forwards the virtual clock precisely to the time the action would be available. However, to provide the application a consistent view of the clocks, Dara must also control the Go standard library time functions so that they report a time that does not violate the ordering constraints placed by other actions that the explorer might have already scheduler in the past. This means that once a time-related action by the explorer has been executed, any subsequent calls into the time library in the future must return a time greater than the real time at which the action would have executed. However, since we do not want to set off any timers without the explicit permission of the scheduler, any future timers place a hard upper bound on the time that can be returned to the application. Thus, the time system calls return a time between a specific time window that is initially set by the global scheduler and modified by the local scheduler to prevent the monotonicity of time whilst preserving the ordering of actions in Dara.

4.2.5 Local Scheduler

Dara installs a local scheduler at every node in the distributed system under test. The local scheduler is responsible for executing the actions specified by the global scheduler. The actions include scheduling a particular goroutine or firing off a timer. Once the action is completed, the local scheduler contacts the global scheduler with the information collected by dgo and waits for the next action to be performed. A timer action is considered to be completed once a notification has been sent to the channel corresponding to the installed timer. A goroutine schedule action is considered to be completed once the goroutine executes its instructions and traps back into the runtime through a network call, a sleep call, or another way to give up its quantum of execution.

4.2.6 Explorer

Dara's explorer uses the coverage information collected by global schedulers from the various local schedulers to determine the next action to be taken by the system. At any given point of time, the explorer requires the system to execute at most one action. This means that at any given instance, at most one goroutine is scheduled to be executed across all the local schedulers and that all but one local schedulers are blocked waiting to receive instructions from the global scheduler. The explorer decides which action the system executes. The explorer can ask the system to execute one of two actions - firing off a specific timer or scheduling a specific goroutine. To start the exploration, the explorer requires a seed input that acts as a starting point for the exploration. This seed input is a recorded schedule of actions from the execution of the system using an end-to-end test or a user-defined workload. The coverage information collected as part of this seed input is used by the explorer to select the first path that needs to be explored. Once the explorer finishes exploring a path, it informs the global scheduler to restart the distributed system so that it can start executing a new path in the state space. This process continues until a user-defined fixed number of paths have been explored. The explorer explores only different states generated by one particular workload. Dara does not modify the contents of the workload.

CounterExample Schedule The presence of a counterexample serves as ev-

idence that there is a bug in the system. Although useful, this is not enough information for the developer to reproduce or debug the system. Thus, to provide a developer with a starting point for debugging, whenever the explorer encounters a bug in the form of a crash or a property failure, the explorer saves a schedule of actions that led to the discovery of the bugs. The schedule contains a list of scheduling and timer actions in the order in which they were executed by the explorer. Additionally, detailed system call event information is also included to provide the developer with some context about what the system was doing when the bug was detected. The counterexample schedule can then be reproduced or analysed by our auxiliary tools for further debugging.

4.2.7 Property Checker

Property Specification The user provides a property checker file that contains the list of properties that need to be checked by Dara during the exploration. The properties are provided as standalone go functions that act upon the variables in the source code. The functions are compiled into plugins which are then executed during exploration with values of variables captured by dgo.

Evaluation Context During the exploration phase, the property checker maintains a context under which to evaluate the properties. The context is a mapping between variable names and their values. When evaluating the properties, the property checker looks for values of the variables in this mapping. If any variable for a property is not present in the mapping then the property is not checked as it probably suggests that the variable has gone out of scope. The users must ensure that the context is up-to-date by reporting the values of variables right after they have been updated or removing the variables from the context when variables go out of scope. The context can only be modified using two API calls exposed by dgo which we discuss below.

Execution Semantics Ideally, the property checker should check properties after the execution of every single instruction. However, that would be too slow and is not ideal. Instead, Dara makes the pragmatic choice of checking properties every time a local scheduler contacts the global scheduler for the next action. We know, that at any given point at most one goroutine can be executing, so, at most one local

scheduler is executing. However, when the executing local scheduler contacts the global scheduler for the next action, there is no goroutine executing at any local scheduler. So after processing the information received from the local scheduler and before scheduling another action at one of the local schedulers, the context is up-to-date with no stale values. Thus, property checking can take place with the correct values and any violation of the property found is a real violation of the property specification. During the execution, if a goroutine updates the value of the variable multiple times then the global scheduler checks the properties with each value of the variable.

Variable Capture Through dgo, the property checker provides two API calls that the user can use to update the context that the property checker is using for evaluating the properties. Note that for correct property checking the user must instrument the application with these API calls so that the properties are evaluated with the correct context. These API calls have no relationship with the instrumentation done by the instrumenter for reporting coverage information. We make a design choice of not instrumenting the source code automatically to report variable information for three reasons: (i) users will find it more confusing to specify properties of the variables especially as there might be variables with the same name across the source code; (ii) using manual instrumentation, users can filter out which variables are necessary and only log the values of the necessary variables; (iii) using manual instrumentation, users can decide how often to report values for variables, allowing the users more flexibility in specifying functions that should only be executed at specific points in the execution of the program. Thus, the user is directly responsible for maintaining the context of the property checker and ensuring it does not contain stale information. The two API calls are as follows:

- **runtime.DaraLog(logID string, varNames string, varValues...)** : A variadic function that reports the values of a list of variables to dgo. The "varNames" argument expects a string of comma-separated variable names. The variadic argument "varValues" reports the values for the variables in the order specified by "varNames".
- **runtime.DaraDeleteLogVar(logID string, varNames string)** : Function that tells the property checker to remove specified variables from its execution

context. The "varNames" argument specifies the list of variables that need to be deleted from the context as a comma-separated string.

Exposing data to applications The property checker also exports API calls to the application so that the application can use values for reporting that will not usually be available to the application. Currently, three such API calls are provided. They are as follows:

- **runtime.NumSendings(ch chan) int** : The function takes a channel as an argument and returns the number of successful sends that have been completed on that channel.
- **runtime.NumDeliveries(ch chan) int** : The function takes a channel as an argument and return the number of successful receives that have been completed on that channel.
- **runtime.GetDaraProcID() int** : The function returns the internal Dara process ID for the node to the application. This is especially useful for making the variable names for the property checker unique as multiple nodes might be using the same variables.

4.3 Exploration Strategies

Dara provides various exploration strategies that can be used by the explorer for exploring the state space. Dara provides a random path exploration strategy in which every path is chosen at random for exploration and is completely independent of the previously explored paths. We refer to this strategy as the *Random* strategy in future sections. Dara provides three novel code coverage-based exploration strategies that optimize the order in which paths are explored.

4.3.1 Distributed Code Coverage

In Dara, we provide novel coverage-based strategies for exploration. The key idea behind the strategies is to use the coverage information for exploring the paths in the state space of the system such that the paths that are more likely to contain a

bug are explored first. We believe that paths that execute different sections of code have a higher likelihood of finding bugs as such paths will execute code that is not well-tested or executed as part of the normal execution of the system. Thus, our exploration strategies explore the paths first that execute a large fraction of code. This allows the explorer to give priority to paths that are more likely to execute code that has not been executed in any of the prior paths. Prior work in combating state space explosion has primarily focused on reducing the state space to be explored by eliminating symmetrical paths [13], by incorporating application-specific protocol information to remove redundant paths [28], and by exploiting communication and node-role symmetry to explore only one path from an equivalent set of path [31]. To the best of our knowledge, there exists no prior work which uses coverage information for ordering paths during exploration. We provide a formal description of three coverage-based strategies below.

A distributed system is defined as a system that consists of a set of nodes, N, connected by the network. The source code of the system is comprised of a set of source code files, F, which define the execution behaviour of the system.

We choose to define coverage at the granularity of a *basic block*, where a basic block represents a code block in the source code (*e.g.* body of an if statement). A *basic-block* is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit [1, 20] The set of all files in the source code, F, is made up of B unique basic blocks where each block is uniquely identified by the fully qualified file path and the starting and ending line numbers of the block in the file. We use block coverage instead of statement coverage as the cost of measuring coverage at a statement level is too high and can be deduced from block coverage.

We provide three different functions for measuring code coverage. Any of these can be used as the optimization objective for state space exploration by the explorer. Let the set of blocks covered by the system across all nodes across all runs be denoted by, B', then the optimization functions are defined as follows:

- *Unique:* $S_{ucount}(B') = \|B'\|$; maximizes the total number of unique blocks covered.
- *Frequency:* $S_{fcount}(B') = \sum_{b \in B'} \frac{1}{Count(b)}$, where the *Count* function returns the

number of times a block was executed. This function maximizes the covered blocks but discounts blocks by how often they are executed. The more a block is executed the less important it is for that block to be covered in the future.

- *NodeFrequency:* $S_{ncount}(B') = \sum_{b \in B'} \frac{Nodes(b)}{Count(b)}$, where the *Count* function returns the number of times a block was executed and the $Nodes$ function returns the number of nodes on which the block was executed. This function maximizes the covered blocks but discounts blocks by how often they are executed but allowing for the block to be executed by more nodes in the system. Thus the more diversely a block is covered the less important it is to be covered in the future.

The explorer uses these optimization functions to guide the exploration of the state space. The explorer maintains a history of action-coverage pairs from previous path explorations that maps an action to the coverage information generated from the execution of the action. The explorer builds the action history by analyzing the coverage information recorded in the seed schedule. As the exploration progresses and executes different paths, the explorer updates the coverage history of each action. The explorer uses an action's coverage history to decide the potential impact the action will have on the coverage of the system. When choosing the next action to be taken by the system, the explorer analyses the set of available actions and decides to execute the action that will maximize the total coverage across all the explored paths according to the objective function chosen by the user. The explorer uses the action's prior coverage history to calculate the potential impact the execution of the action will have on the coverage of the system. If there is no action that would lead to an increase in the total coverage, then a previously unseen action is selected. If no such action exists, then an action is selected that will lead to an increase in the coverage of the current path but will not result in a previously seen path. If no such action exists, the coverage finally defaults to choosing a random action.

Chapter 5

Implementation

5.1 Dara - Concrete MC

We now present details of Dara's implementation. Table 5.1 shows the number of lines of code for each component. The Dara codebase is written entirely in Go. The codebase is available at https://github.com/DARA-Project/GoDist-Scheduler.

Dara's current implementation sets up all the local schedulers, i.e. all the different nodes of a distributed system, as well as the Global Scheduler on a single machine. This is done for the sake of simplicity and to avoid any uncontrolled bugs caused due to network failures or partitions that may occur if all the nodes were distributed across different machines. The Global Scheduler and the local schedulers communicate via shared memory as shown in Figure 5.1.

5.1.1 Modified Go Runtime

As described in Chapter 4, a concrete model checker must be able to control execution of events in a system. For Dara to control events, we modified the Go runtime to interpose on all system calls. We also modified the runtime to control timers installed by the application. We refer to our modified Go runtime as **dgo**. Moreover, **dgo** also collects code coverage information of an application through the *runtime.ReportBlockCoverage* function call described in Section 4.2.3. We chose to interpose on the system calls in the Go runtime as the Go runtime is the last layer before the system call traps into the operating system. Thus, interposing on system

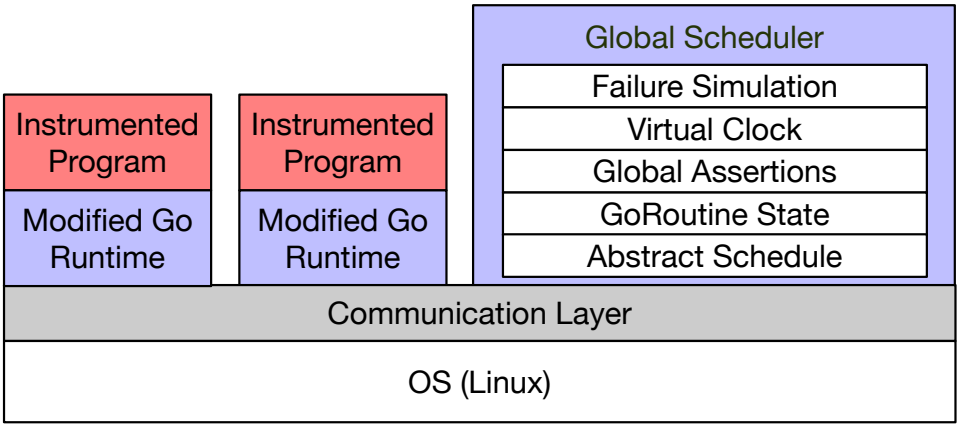


Figure 5.1: Global and Local Scheduler interface

Component	LoC
Overlord	804
Global Scheduler	1202
Modified Go Runtime	1597
Local Scheduler	1167
Explorer	671
Property Checker	195
Instrumenter	508
Virtual Clock	150
Auxiliary Tools	332
Total	6626

Table 5.1: Lines of Code (LoC) breakdown by component for Dara. Note that the total lines of the replay engine are included as part of the global scheduler instead of auxiliary tools.

calls in the Go runtime provides Dara the opportunity to control the execution and completion of system calls without any modifications to the underlying operating system. This allows Dara to be portable and runnable on any machine that installs **dgo**. The complexity in terms of lines of code for interposing on various go runtime libraries is shown in Table 5.2. The modified Go runtime is built on top of go version 1.10.4 and is available at https://github.com/DARA-Project/GoDist.

Category	# of Functions	# of LoC
Network	10	178
Sync	6	104
Time	3	30
Syscall + OS	50	749
Panic	2	10
Other	NA	526
Total	71	1597

Table 5.2: *Interposition complexity*. This table shows the lines of code for interposing on various system calls and time-related events in the go runtime.

5.1.2 System Event Capture

Dara captures ten different kinds of events detailed in Table 5.3. Every recorded schedule for a local scheduler must start with an "INIT" event and must end with an "END" event or a "CRASH" event. The "END" event represents a successful execution of the system whereas the "CRASH" event represents a crash event in the system that maybe caused due to a panic or due to receiving an external kill signal. The "INIT" event serves as the starting point for a local scheduler and an "END" or a "CRASH" event serves as the ending point for a local scheduler in an execution of a given path. Scheduling decisions made by the runtime are recorded as "SCHEDULE" events and new goroutine creations are recorded as "THREAD" events. "SLEEP" and "TIMER" are two events that capture time-related operations such as a goroutine being put to sleep and a new timer being installed. The global scheduler uses these four events to maintain the correct state for every goroutine in the node. Dara captures "SYSCALL" events that capture the system call and the arguments and return values with which the system call was executed. This is not particularly useful for exploration but it is useful for post-exploration analysis of counterexample schedules by the user. Finally, Dara also captures two user-defined events called "LOG" and and "DELETELOGVAR" which update the variables in the property checker context under which the properties are evaluated. The global scheduler uses these events to update the context for the property checker. We chose these ten different events as these events encapsulate enough information about the

Event	Description
Init	Initialization of go runtime/local scheduler
System Call	The execution of a system call, including calls to time and random libraries
Thread Create	Corresponds to the creation of a new goroutine.
Thread Schedule	Corresponds to the scheduling of a certain goroutine
Sleep	Corresponds to a goroutine asking to go to sleep
Timer	Corresponds to a new timer being installed by the goroutine
Log	User-defined log event which provides variable-value mappings for property checking
DeleteLogVar	User-defined event for removing a list of variables from the property checker's context after they have gone out of scope
Crash	System crash event
End	End of execution of a local scheduler.

Table 5.3: List of all system events

state of a node that is needed by the explorer to decide the next action, the property checker to correctly evaluate properties, and for the global scheduler to construct a schedule for replay.

5.1.3 Overlord

Overlord is the command hub for the user. The user uses the overlord to instrument the system, explore the system, record an execution of the system with a given driver program, or replay a counterexample schedule recorded by Dara during exploration. The Overlord is currently implemented as a command line application that expects two inputs as command line arguments. The first argument is the command that the user wants to execute - this is one of (instrument, explore, record, replay). The second argument is a configuration file that contains arguments for various inputs needed by Dara such as max depth of a path during exploration, maximum number of paths to be explored, paths to the build and run scripts for the system. The Overlord also sets up the environment and manages the environment for Dara's Global Scheduler to communicate with the different Local Schedulers. The Overlord sets up the shared memory for the global scheduler

and the local schedulers communicate with each other. Overlord also runs an RPC server to serve meta commands issued by the Global Scheduler such as restarting the execution of the system so that the explorer can start exploring a different path in the system. We chose to implement the communication between the Overlord and the Global Scheduler through RPC as it allows us to distribute the Overlord and Global Scheduler across different nodes in the future.

5.1.4 Local Scheduler

Each local scheduler is responsible for scheduling of actions for one node in the distributed system. Scheduling includes running/blocking goroutines and firing off timer events. The local scheduler performs these actions based on the commands provided by the global scheduler. Thus, to correctly control goroutine scheduling for a given node, the local scheduler lives inside the "runtime/proc.go" file where it interposes on the default go scheduler to execute actions as commanded by the global scheduler. Since the Go runtime creates and manages its own threads in the form of goroutines, interposition in the goroutine management code is required. This is because, without interposing on the goroutine management code it would be impossible for Dara to schedule threads and control the interleaving of the goroutines. Each local scheduler also forwards the information collected by **dgo** which includes the coverage information, the system call event information, and information about the state of every goroutine. To prevent multiple goroutines running at the same time, for each local scheduler the environment variable "GOMAXPROCS" is set to 1 which ensures that only one OS thread is executing for Dara at any given point of time. Moreover, the local scheduler that is not currently executing any action specified by the explorer blocks itself waiting for instructions from the global scheduler.

System Call Events

The local scheduler captures 62 different system calls that are made using **dgo** and reports the execution of these system calls to the global scheduler. We manually analyzed the Go runtime version 1.10.4 to find the system calls and found these 62 system calls. We believe that this list is exhaustive. We chose to interpose on the

lowest level of implementation of each system call before the system call calls a generic syscall execution function implemented in assembly. We did this so that we could capture the system calls being executed by all the high-level user-facing functions provided by the standard library. The list of all system calls captured by are listed in Table 5.4.

5.1.5 Global Scheduler

The Global Scheduler acts as the global coordinator for coordinating the execution of actions by the local schedulers. It digests the information forwarded by the local scheduler and prepares the information for the explorer and the property checker. At the behest of the explorer, the global scheduler commands the local scheduler to execute a specific action as desired by the explorer. However, if the explorer wants to finish executing the current path and backtrack to a different path, the global scheduler contacts the overlord to restart the system. The global scheduler also maintains the state of all the goroutines across all the nodes and keeps track of all the timers that have been installed by the node at every local scheduler. This is to ensure that the explorer always chooses an enabled action for execution. The global scheduler runs an RPC client for the RPC server provided by the overlord. RPC is chosen as the medium of communication betwene overlord and the global scheduler to support distribution of the global scheduler and the overlord across multiple nodes. The client makes RPC calls at the server to either restart the system for a new exploration or to kill the execution to bring halt to the exploration. The global scheduler communicates with every local scheduler over shared memory. When not executing an action, the local schedulers poll over a lock in shared memory which prevents the local scheduler from executing any action without the permission of the global scheduler or the explorer. Each local scheduler has its own lock which is shared with the global scheduler. We chose a lock based system because we could not figure how to set up blocking and notifications inside the Go runtime code. After the global scheduler has installed an action to be executed at a local scheduler, the global scheduler waits for the local scheduler to update the shared memory with the result of the scheduled action.

Virtual Clock The Global Scheduler implements a virtual clock for each node to

READ	READDIRNAMES	MKDIR
WRITE	WAIT4	CHDIR
OPEN	KILL	UNSETENV
CLOSE	GETUID	GETENV
STAT	GETEUID	SETENV
FSTAT	GETGID	CLEARENV
LSTAT	GETEGID	ENVIRON
LSEEK	GETGROUPS	TIMENOW
PREAD64	EXIT	READLINK
PWRITE64	RENAME	CHMOD
GETPAGESIZE	TRUNCATE	FCHMOD
EXECUTABLE	UNLINK	CHOWN
GETPID	RMDIR	LCHOWN
GETPPID	LINK	FCHOWN
GETWD	SYMLINK	FTRUNCATE
READDIR	PIPE2	FSYNC

UTIMES
FCHDIR
SETDEADLINE
SETREADDEADLINE
SETWRITEDEADLINE
NET-READ
NET-WRITE
NET-CLOSE
NET-SETDEADLINE
NET-SETREADDEADLINE
NET-SETWRITEDEADLINE
NET-SETREADBUFFER
NET-SETWRITEBUFFER
SOCKET
LISTEN-TCP
SLEEP

Table 5.4: List of all system calls captured and subsequently reported to the global scheduler by the local scheduler with the modified go runtime version 1.10.4

ensure that time-related actions are consistent in a node. The virtual clock ensures that the actions at a node do not violate the real time order of actions. The global scheduler only increments the virtual clock if the explorer wishes to schedule a previously sleeping goroutine or to fire off a timer. The **dgo** runtime provides virtual clock support as it ticks the virtual clock by one tick for every function call made by the application to the time library to check the current time. This ensures that the time of a virtual clock is monotonically increasing from the application's perspective.

5.1.6 Explorer

The explorer is exploring the state space of the distributed system according to a pre-selected exploration strategy. Dara's explorer currently provides four different exploration strategies - Bounded-Depth Random Path, and three coverage strategies each of which maximizes a coverage score function described in Section 4.3.1. For the coverage-based strategies, the explorer interfaces with the Global Scheduler to obtain up-to-date coverage information about the nodes and selects the next action that has the highest potential for increasing the coverage score in this run. This ensures that the system is executing previously unexecuted blocks to explore new pathways in the system's source code. The explorer selects the path to be executed according to one of the coverage-based strategies defined in Section 4.3.1. Even though we provide three coverage-based strategies that focus on executing diverse code in the system, we believe that these are not the only coverage-based strategies that will be useful in finding bugs. There are possibly other strategies that might be better than our strategies at finding a certain class of bugs. We leave the implementation of other coverage-based strategies as future work.

Based on the exploration strategy selected by the user, the explorer decides the next action to be executed which is then executed by the Global Scheduler. The explorer finishes a path that it is exploring if it reaches the maximum depth of the path as specified by the user or if it finds a property violation or a crash. It then asks the global scheduler to restart the system so that the explorer can explore a new path. When the explorer has explored the maximum number of paths specified by the user the explorer asks the global scheduler to halt the system and to save schedules with

bugs which can be replayed by the users for further analysis and debugging.

5.1.7 Property Checker

Property Specification & Data Collection Users specify properties of the system as go functions over the variables in the programs. Figure 5.2a provides an annotated example of a property file for the code snippet in Figure 5.2b. Each property consists of three parts: ① The name of the property; ② The name of variables, each on a new line, which are names given by the developer for real variables in the source code; and ③ the body of a function. The variable names specified in the property specification are used by Dara to form the context for tracking and using the right values from the runtime. For a property to be a valid property, it must return a Boolean value indicating whether the condition specified by the body of the property is true or not. For correctness, each variable name across all the properties should be used to track only one single real variable in the source code. The same variable name can be used for the same variable in multiple properties. Currently, Dara is not capable of tracking arbitrary variables so the user has to report the values of variables to the Dara using the "runtime.DaraLog()" statement as shown in Figure 5.2b. For completeness, the new values of the variables should be reported after every update to the variable. Thus, the user is directly responsible for maintaining the state of the property checker and ensuring that the property checker does not contain stale information. For example, ④ would result in a property failure since the values of *varA* and *varB* do not match but ⑤ will not result in a property failure as both *varA* and *varB* have the same value. In this example, *a* is given the name *varA* and *b* is given the name *varB* by the user. Once the variables go out of scope, Dara can be notified that variables have gone out of scope using the function "runtime.DaraDeleteLogVar".

Property Execution The property file is provided as input at the start of the exploration where each property in the property file is parsed and compiled into a plugin that can be loaded and executed at runtime with arbitrary values. The bodies of the functions are compiled into plugins using the go-eek evaluation library [34] To execute these plugins with the appropriate context, the Global Scheduler maintains a mapping of the variable names to their values which is then passed to the property

```
1  package property
2
3  //EqualVarAVarB ①
4  //varA ②
5  //varB
6  func equalVarAVarB(A, B int) bool {
7      return A == B ③
8  }
```

(a)

```
1  func foo() {
2  var a, b int
3  a, b = bar(5)   // a = 4, b = 6
4  runtime.DaraLog("LogID1", "varA,varB", a
       ↪ ,b) ④
5  a, b = bar(10) // a = 5, b = 5
6  runtime.DaraLog("LogID2", "varA,varB", a
       ↪ ,b) ⑤
7  }
```

(b)

Figure 5.2: Depiction of how (a) properties of a system are specified, and, (b) how the values for property checking are captured through the source code.

checker so that it can execute the properties with the correct values of variables. If all the variables required by the property are not present in the context then the checking of the property is skipped. Otherwise, a property failure is recorded if the return value of executing the property under the correct context is false.

5.1.8 Instrumenter

Dara's instrumenter is implemented as a repurposed version of the go coverage tool [16] for rewriting go source code. An example of a file generated by the instrumenter is shown in Figure 5.3. In the example, each basic block contains a function call into **dgo** containing the unique block ID that specifies which block is about to be executed.

Unique Block IDs For accurate coverage scores, it is imperative that correct information about the blocks is recorded in **dgo**. This requires that each basic code block must have a unique ID amongst all the basic code blocks in the source code of the system under exploration. To achieve this, each basic block is assigned a three-part ID of the form "path-to-file:startLine:endLine" where the first part is the full path to the file where the block resides, the second part is the starting line number of the block, and the third part is the ending line number of the block. This guarantees the uniqueness of block IDs. To avoid computing block IDs during runtime, the instrumenter embeds the unique block ID statically as an argument in the function call to the ReportBlockCoverage function.

```
1  package main
2
3  import "runtime"
4
5  import (
6          //"fmt"
7          "log"
8          "os"
9  )
10
11 func normal() {
12         runtime.ReportBlockCoverage("../examples/SimpleInstrument/file_read.go:10:12")
13         f, err := os.Open("file.txt")
14         if err != nil {
15                 runtime.ReportBlockCoverage("../examples/SimpleInstrument/file_read.go:12:14")
16                 log.Fatal(err)
17         }
18         runtime.ReportBlockCoverage("../examples/SimpleInstrument/file_read.go:16:18")
19
20         b1 := make([]byte, 20)
21         _, err = f.Read(b1)
22         if err != nil {
23                 runtime.ReportBlockCoverage("../examples/SimpleInstrument/file_read.go:18:20")
24                 log.Fatal(err)
25         }
26         runtime.ReportBlockCoverage("../examples/SimpleInstrument/file_read.go:21:21")
27         f.Close()
28 }
29
30 func main() {
31         runtime.ReportBlockCoverage("../examples/SimpleInstrument/file_read.go:24:26")
32         normal()
33 }
```

Figure 5.3: Instrumented version of a file called file_read.go with function calls to the **dgo** for reporting coverage information

5.2 Auxiliary Tools

Understanding the bug traces can be hard. Thus, to aid in the understanding of the bug traces, we make use of existing tools such as ShiViz [5] and provide some of our own tools.

5.2.1 Replay Engine

We provide a Replay Engine to deterministically replay buggy executions that were recorded during the exploration phase. This gives the developers an opportunity

```
2020/08/07 19:09:45 SCHEDULE LENGTH : 32
2020/08/07 19:09:45 LOG   Events : 1
2020/08/07 19:09:45 END   Events : 1
2020/08/07 19:09:45 INIT   Events : 1
2020/08/07 19:09:45 SCHEDULE   Events : 9
2020/08/07 19:09:45 THREAD   Events : 2
2020/08/07 19:09:45 SYSCALL   Events : 18
```

Figure 5.4: Output of the schedule info tool for a recorded schedule

to reproduce bugs that the explorer found and perform root cause analysis. The Replay Engine leverages the modified go runtime and controls the local schedulers for scheduling specific goroutines. However, during the replay, if the action to be replayed is not available, then the Replay Engine reports a failure. However, we are yet to encounter this failure in practice. The Replay Engine is implemented as 200 lines of go code on top of the modified go runtime and local scheduler framework with the overlord setting up the communication layers between the replay engine and the local schedulers just like it does for the explorer.

5.2.2 Schedule Info Tool

Our suite of auxiliary tools includes a schedule info tool that provides the user with some basic information about a recorded schedule. Currently, the tool prints out the number of events in the schedule as well as breakdown of events by the type of the event. Sample output from running the schedule info tool on a recorded schedule is shown in Figure 5.4.

5.2.3 Coverage Report Tool

We provide a coverage report generator tool which takes the list of all basic blocks in the source code (produced by the instrumenter) and a list of recorded schedules whose coverage needs to be analysed. Based on these schedules, the tool generates report showing the total number of blocks in the code, the number of blocks covered in the execution of the schedules, the frequency with which each block was covered in the schedule, and the number of blocks that were left uncovered in the execution of the schedules. Figure 5.5 shows the coverage report of

```
Total # of blocks in source code: 19
Total # of blocks covered: 9
Total # of blocks uncovered: 10
Covered Block     Frequency
../examples/ProducerConsumer/ProducerConsumer.go:77:92    1
../examples/ProducerConsumer/ProducerConsumer.go:22:24    1
../examples/ProducerConsumer/ProducerConsumer.go:44:47    5
../examples/ProducerConsumer/ProducerConsumer.go:37:39    1
../examples/ProducerConsumer/ProducerConsumer.go:52:66    1
../examples/ProducerConsumer/ProducerConsumer.go:24:27    5
../examples/ProducerConsumer/ProducerConsumer.go:42:44    1
../examples/ProducerConsumer/ProducerConsumer.go:48:49    1
../examples/ProducerConsumer/ProducerConsumer.go:17:19    1
Uncovered Blocks:
../examples/ProducerConsumer/ProducerConsumer.go:74:74
../examples/ProducerConsumer/ProducerConsumer.go:68:70
../examples/ProducerConsumer/ProducerConsumer.go:71:71
../examples/ProducerConsumer/ProducerConsumer.go:71:73
../examples/ProducerConsumer/ProducerConsumer.go:97:98
../examples/ProducerConsumer/ProducerConsumer.go:98:100
../examples/ProducerConsumer/ProducerConsumer.go:101:101
../examples/ProducerConsumer/ProducerConsumer.go:94:96
../examples/ProducerConsumer/ProducerConsumer.go:66:68
../examples/ProducerConsumer/ProducerConsumer.go:92:94
```

Figure 5.5: Output of the coverage report tool for a given schedule

```
Total number of property checks: 9
Index: 3 Property Failures:0
Index: 11 Property Failures:0
Index: 12 Property Failures:0
Index: 16 Property Failures:0
Index: 19 Property Failures:0
Index: 22 Property Failures:0
Index: 25 Property Failures:0
Index: 28 Property Failures:0
Index: 31 Property Failures:1: EqualSendingsDeliveries map[NumDeliveries:4
    ↪ NumSendings:5]
```

Figure 5.6: Output of the propchecker report tool for a given schedule

one schedule for a simple producer consumer program. From the output, it is clear which blocks were more frequently covered than the others while which blocks were left uncovered.

5.2.4 PropChecker Report Tool

We also provide a propchecker report tool which reports the total number of property checks that happened in a given schedule and how many of those property checks resulted in failure. Moreover, for each time the property check took place, the event index at which it took place is specified as well as detailed information regarding the property failures that took place at that event. Figure 5.6 shows the output from the propchecker report tool for a given schedule. Notice that out of the nine property checks, there was only one instance where a property failure was found. The name of the property that failed along with the variable values that caused the property to fail are also listed for better understanding of the failure by users.

5.2.5 ShiViz Converter Tool

ShiViz [5] is an online visualization tool that is used to generate interactive communication graphs from distributed system execution logs. The ShiViz converter tool converts an execution schedule recorded by Dara into a ShiViz-compatible trace. The converter tool generates a ShiViz-compatible log such that the coverage information for every goroutine's execution is embedded in the log. The log also embeds information about property failures by including the exact position in the scheduler where the property failure was captured and the variable values which led to the property failure. An annotated snapshot of the ShiViz visualization of the generated log from the recorded schedule is shown in Figure 5.7. In the visualization, each unique goroutine ① is given its own vertical lane representing the execution of the goroutine. The circles on the lane represent events that were recorded during the execution of the goroutine. The links ② between goroutines represent the switching of execution from one goroutine to another which helps the user track when the execution switched from one goroutine to another. In addition to event specific information, certain events are also tagged with detailed information about the amount and frequency of code blocks ③ the goroutine covered. Certain events are also tagged with property failure information ④ indicating the event is the place where the property check failed. This allows the user to backtrack through the execution and potentially find the cause of the property violation.

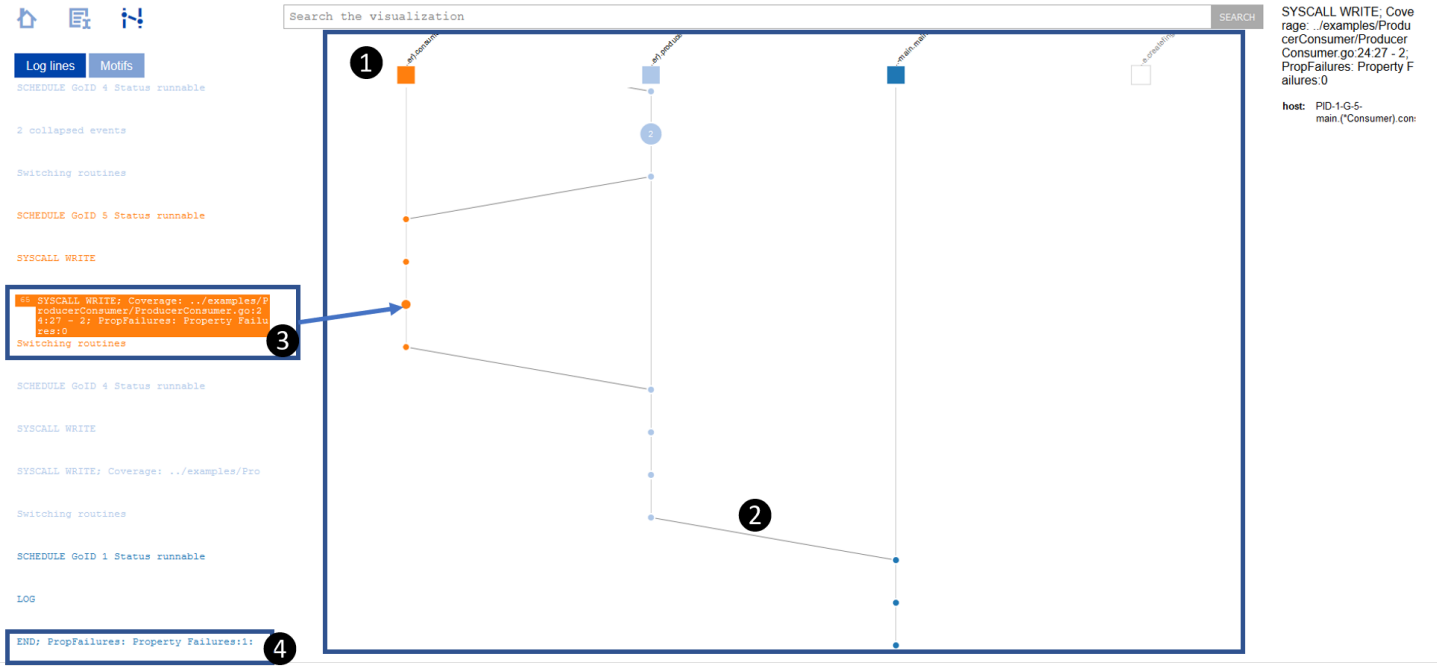


Figure 5.7: Shiviz visualization of a recorded schedule

Chapter 6

Evaluation

We focus our evaluation on measuring the efficacy of our coverage-based exploration strategy, measuring how effective Dara is at finding and catching bugs in Go-based systems, and measuring the performance of the various components of Dara. We address the following six questions regarding our system:

1. Can Dara find and replay Go-specific bugs?
2. Can Dara find bugs in systems?
3. What is the performance cost of property checking?
4. What is the performance cost of instrumenting the system to capture coverage information?
5. What is the performance cost of modifying the go runtime?
6. What is the performance cost of scheduling actions at nodes?

6.1 Experimental Setup

Experiment Machine We run all our experiments with Dara on an Intel i7-core 3.1GHz processor machine with 32GB of RAM.

Strategy Comparison To evaluate the efficacy of Dara's coverage-based strategies, we compare them to the random path, called *Random*, exploration strategy. As

part of this strategy, Dara explores actions at random with no constraint preventing Dara from executing previously seen paths or subpaths. This ensures that the *Random* strategy is truly random without incorporating information collected during prior runs.

6.2 Can Dara find and replay Go concurrency bugs

Methodology To assess whether Dara is able to find and catch Go concurrency bugs, we select three programs all of which contain a non-deterministic bug, i.e., a bug that manifests only intermittently. Two of the selected programs follow the data race bug pattern described in a 2019 study of real-world concurrency bugs [39], while the other is a synchronization bug caused by improper use of channels.

6.2.1 Data Race Crash bug

Figure 6.1 shows a snippet of code with a global unprotected map variable being shared across initializer, reader, and writer goroutines. The initializer routine initializes the map; the reader goroutine reads values from the map; and the writer writes values to the map. Upon completion, each goroutine writes to a shared channel with the main goroutine indicating completion. The main goroutine blocks until it receives a notification of completion from each goroutine. The expected behaviour is for the initializer goroutine to initialize the map first and then the reader and writer goroutines to read from and write to the map concurrently. However, there is a nondeterministic bug in this code that leads the program to crash due to a data race on the shared global map variable. In this bug, the writer goroutine could potentially attempt to write to the map before it has been initialized which will lead to a crash due to an attempt to write to a nil map. Note that if reading from a nil map is a safe operation in go and does not lead to crash.

Dara can capture these crashes as it explores all potential interleavings of the goroutines and generates a schedule in which the writer goroutine attempts to write to a nil map causing a crash. We ran Dara's explorer with three different exploration strategies - *Random*, *Unique*, and *Frequency* and explored 10 different paths with each strategy. Each strategy found the bug in the first three paths it explored.

```
1  var m map[string]string
2  var comm chan int
3
4  func initmap() {
5      m = make(map[string]string)
6      comm <- 1
7  }
8
9  func reader() {
10     for i := 0; i <= 5; i++ {
11         for k, v := range m {
12             fmt.Println(k,v)
13         }
14     }
15     comm <- 2
16 }
17
18 func writer() {
19     for i := 0; i <= 5; i++ {
20         m[string(i)] = string(i)
21     }
22     comm <- 3
23 }
24
25 func main() {
26     // Initializes the communication channel between the main
27     // goroutine and the children goroutines
28     comm = make(chan int, 3)
29     // Launches a goroutine that initializes the shared map
30     go initmap()
31     // Launches a goroutine that reads from the shared map
32     go reader()
33     // Launches a goroutine that writes to the shared map
34     go writer()
35     <- comm
36     <- comm
37     <- comm
38 }
```

Figure 6.1: A data race on an unprotected global shared variable leads to a crash.

```
1    func main() {
2        var wg sync.WaitGroup
3        wg.Add(5)
4        for i := 0; i < 5; i++ {
5            go func() {
6                fmt.Println(i)
7                wg.Done()
8    +           runtime.DaraLog("VarCheck","i",i)
9            }()
10       }
11       wg.Wait()
12   }
```

(a)

```
1  //VarCheckProperty
2  //i
3  func VarCheckProperty(Val int) bool {
4      return Val < 5
5  }
```

(b)

Figure 6.2: (a) Code with a data race bug on the loop variable between the parent goroutine and child goroutine; (b) property file used by Dara to find this bug

6.2.2 Data Race Property Violation

Figure 6.2a shows a snippet of code which has a bug caused by a data race. The local loop variable *i* is shared between the parent goroutine and the child goroutines it creates at line 5. The developer intends each child goroutine to use a distinct *i* value. However, the values of *i* are non-deterministic in the program. For example, if children goroutines begin after the whole loop of the parent goroutine is completed, then the value of *i* will be equal to 5 for all goroutines. Thus the output of the program will be "55555" instead of some anagram of "01234". The buggy program will produce the correct result only if each child goroutine uses the value of *i* before the parent goroutine updates the value.

Dara can catch such bugs by having the user specify a property that checks whether the shared variable takes desired values only during the various execution paths during the exploration. For this specific bug, the user can instrument their code with just one additional line of code, as shown in line 8 (coloured blue) of

Figure 6.2a, which reports the value of *i* used by the goroutine during the execution. Then, the user simply writes a property on the value of variable *i*, specifying that the value of *i* must always be less than 5 as shown in Figure 6.2b. We ran Dara's explorer with three different exploration strategies - *Random*, *Unique*, and *Frequency* and explored 20 different paths with each strategy. All three strategies were able to find the bug within the first paths they explored. *Random* strategy performed the best for this program as it found the bug in the first path it explored and was able to find the bug in 16 out of the 20 paths it explored. With the *Unique* strategy, Dara found the bug in 13 of the 20 different explored paths. With the *Frequency* strategy, Dara was able to find the bug in 12 of the 20 different explored paths.

6.2.3 Channel synchronization bug

Figure 6.3 shows a snippet of code of a concurrent Producer-Consumer system. In this system, the main goroutine creates a producer goroutine and a consumer goroutine with the producer goroutine programmed to generate five elements which it will send to the consumer goroutine via a shared channel. After the producer goroutine finishes producing five elements, it notifies the main goroutine about the end of production via a message on a shared channel. The expected behaviour for this system is for the producer to produce five elements and the consumer to consume five elements. However, this code has a nondeterministic bug. The bug manifests in the form of the producer correctly producing five elements but the consumer consuming only four elements, because once the production is finished, the main goroutine becomes unblocked as it was only blocked waiting to read from the shared channel between itself and the producer goroutine. Once unblocked, the main goroutine can exit and cause the program to terminate without letting the consumer complete consumption of the last element.

Dara can catch such bugs by simply checking whether the number of successful sends and deliveries on the shared channel are equal. In Figure 6.3, line 49 reports the number of successful sends and deliveries on the shared channel between the producer and consumer goroutine to Dara. Figure 6.4 shows the corresponding property that Dara checks during exploration to find a schedule where the property is violated. We ran Dara's explorer with three different exploration strategies - *Random*,

```
1     type Consumer struct {
2         msgs *chan int
3     }
4
5     // consume reads the msgs channel
6     func (c *Consumer) consume() {
7         fmt.Println("[consume]: Started")
8         for {
9             msg := <-*c.msgs
10            fmt.Println("[consume]: Received:", msg)
11        }
12    }
13
14    // Producer definition
15    type Producer struct {
16        msgs *chan int
17        done *chan bool
18    }
19
20    // produce creates and sends the message through msgs channel
21    func (p *Producer) produce(max int) {
22        fmt.Println("[produce]: Started")
23        for i := 0; i < max; i++ {
24            fmt.Println("[produce]: Sending ", i)
25            *p.msgs <- i
26        }
27        *p.done <- true // signal when done
28        fmt.Println("[produce]: Done")
29    }
30
31    func main() {
32
33        // get the maximum number of messages from flags
34        max := flag.Int("n", 5, "defines the number of messages")
35
36        flag.Parse()
37
38        var msgs = make(chan int)  // channel to send messages
39        var done = make(chan bool) // channel to notify end of production
40
41        // Start a goroutine for Produce.produce
42        go NewProducer(&msgs, &done).produce(*max)
43
44        // Start a goroutine for Consumer.consume
45        go NewConsumer(&msgs).consume()
46
47        <-done
48
49  +         runtime.DaraLog("FinalCheck", "NumSendings,NumDeliveries", runtime.NumSendings(msgs),
        runtime.NumDeliveries(msgs))
50    }
```

Figure 6.3: Snippet of code for a concurrent Producer Consumer system with an unexpected behaviour bug caused due to lack of synchronization between the producer, consumer, and main goroutines

```
1 //EqualSendingsDeliveries
2 //NumSendings
3 //NumDeliveries
4 func EqualSendingsDeliveries(Sends, Dels int) bool {
5     return Sends == Dels
6 }
```

Figure 6.4: The property file for the producer consumer code in Figure 6.3

Unique, and *Frequency* and explored 50 different paths with each strategy. We chose to explore only 50 paths since we believed that would be enough paths to cover every single block in the program at least once and explore enough orderings of exploration of these blocks to find the injected bug. With the *Unique* strategy, Dara was able to find the bug in the second path it explored. Out of the 50 paths, Dara found the bug in 29 different paths with the *Unique* strategy. With the *Frequency* strategy, Dara was able to find the bug in the first path it explored but out of the 50 paths, it found the bug in only 20 different paths. *Random* strategy performed the worst for this program as it found a bug only in the fifth path it explored and was able to find the bug in only 15 out of the 50 paths it explored.

6.2.4 Summary

Dara correctly finds the concurrency bug in all of the three programs and generates a counterexample schedule that can be replayed using our replay engine. This allows the developers to analyze the counterexample and reproduce the bug deterministically for root cause analysis. One of the main challenges faced by the authors who conducted a study of real-world concurrency bugs in Go [39] was that it was hard and time consuming to reproduce concurrency bugs. We believe that with a tool like Dara, such studies will be much easier to conduct as it would allow for easier reproduction of concurrency bugs. We also believe that there is no one exploration strategy that will work best for all programs but a combination of various strategies will be effective in unearthing different kinds of bugs.

6.3 Can Dara find bugs in systems

To evaluate if Dara can find bugs in systems, we use implementation of a popular concurrent algorithm, Dining Philosophers [11] and inject bugs in its implementation. We then use Dara's explorer to explore the state space to find the injected bug(s). We compare our coverage-based exploration strategies with one baselines for exploration - random path.

6.3.1 Dining Philosophers

```
1  package main
2
3  import (
4          "fmt"
5          "math/rand"
6          "time"
7  )
8
9  type Philosopher struct {
10         name            string
11         chopstick       chan bool
12         neighbor        *Philosopher
13 }
14
15 func makePhilosopher(name string, neighbor *Philosopher) *Philosopher {
16         phil := &Philosopher{name, make(chan bool, 1), neighbor}
17         phil.chopstick <- true
18         return phil
19 }
20
21 func (phil *Philosopher) think() {
22         fmt.Printf("%v is thinking.\n", phil.name)
23         time.Sleep(time.Duration(rand.Int63n(1e9)))
24 }
25
26 func (phil *Philosopher) eat() {
27         fmt.Printf("%v is eating.\n", phil.name)
28         time.Sleep(time.Duration(rand.Int63n(1e9)))
29 }
30
31 func (phil *Philosopher) getChopsticks() {
32         timeout := make(chan bool)
33         go func() {
```

```
34                  time.Sleep(1e9)
35                  timeout <- true
36          }()
37          <-phil.chopstick
38          fmt.Printf("%v got his chopstick.\n", phil.name)
39          select {
40          case <-phil.neighbor.chopstick:
41                  fmt.Printf("%v got %v's chopstick.\n", phil.name, phil.neighbor.
                        ↪ name)
42                  fmt.Printf("%v has two chopsticks.\n", phil.name)
43                  return
44          case <-timeout:
45                  phil.chopstick <- true
46                  phil.think()
47                  phil.getChopsticks()
48          }
49  }
50
51  func (phil *Philosopher) returnChopsticks() {
52          phil.chopstick <- true
53          phil.neighbor.chopstick <- true
54  }
55
56  func (phil *Philosopher) dine(announce chan *Philosopher) {
57          phil.think()
58          phil.getChopsticks()
59          phil.eat()
60          phil.returnChopsticks()
61          announce <- phil
62  }
63
64  func main() {
65          names := []string{"A", "B", "C", "D", "E", "F"}
66          philosophers := make([]*Philosopher, len(names))
67          var phil *Philosopher
68          for i, name := range names {
69                  phil = makePhilosopher(name, phil)
70                  philosophers[i] = phil
71          }
72          philosophers[0].neighbor = phil
73          fmt.Printf("There are %v philosophers sitting at a table.\n", len(
                ↪ philosophers))
74          fmt.Println("They each have one chopstick, and must borrow from their
                ↪ neighbor to eat.")
75          announce := make(chan *Philosopher)
76          for _, phil := range philosophers {
77                  go phil.dine(announce)
```

```
78          }
79          for i := 0; i < len(names); i++ {
80                  phil := <-announce
81                  fmt.Printf("%v is done dining.\n", phil.name)
82          go func() {
83              // Bug: No guarantee that the close will happen
84              // before a philosopher is able to return the chopstick
85              time.Sleep(time.Second)
86              close(phil.chopstick)
87          }()
88          }
89  }
```

Listing 6.1: Buggy implementation of dining philosophers; the buggy lines are shown in red

System Description We use an in-house concurrent implementation of a solution to the dining philosophers problem. Each philosopher alternates between eating and thinking. To start eating, a philosopher must grab two chopsticks - their own chopstick and their neighbour's chopstick. The philosopher must grab their own chopstick first before grabbing their neighbour's chopstick. If the philosopher is unable to grab both chopsticks in a fixed amount of time, they return both chopsticks and return to thinking before trying again. Once a philosopher finishes eating, the philosopher finishes dining and exits the table. In our implementation, shown in Listing 6.1, each philosopher is represented by their own goroutine. Each philosopher maintains their own channel to which they send and receive their chopstick. A philosopher also has access to the chopstick channel of one of its neighbours. To acquire a chopstick, the philosopher must successfully read from the channel and to release a chopstick, the philosopher must send the chopstick to the channel.

Bug Injected In Listing 6.1, we add a bug at lines 82-87, where after a philosopher has finished dining, a new goroutine is launched which sleeps for one second and then closes the chopstick channel for that philosopher so that no one can return a chopstick to that philosopher after all the philosophers have finished dining but still allowing for another philosopher to take a chopstick. The bug is nondeterministic as the channel might get closed before a philosopher who was using the chopstick has had a chance to complete their meal. Since the injected bug is a crash bug, we

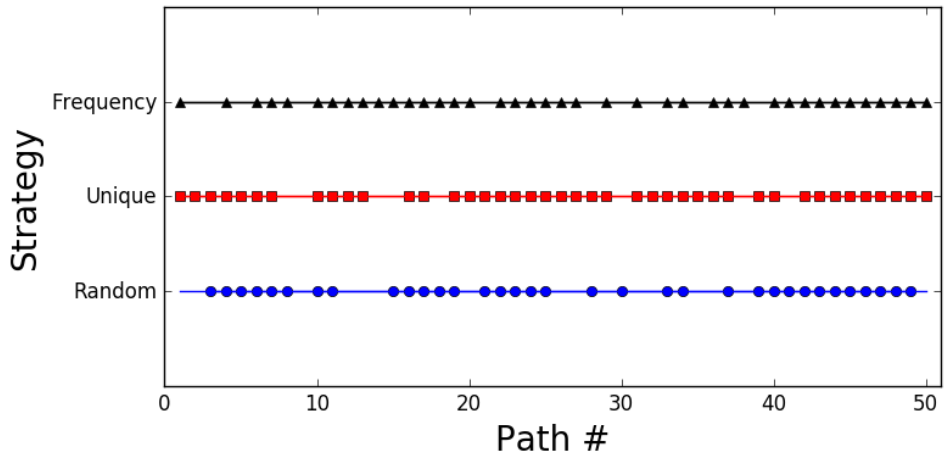


Figure 6.5: On which paths during the exploration did Dara report a bug for each of the three search strategies; the path number on which Dara found a bug is indicated by a mark

do not need to provide a property file for enforcing properties during exploration. We could not enforce the liveness property that all philosophers must eventually eat at least once since Dara's property checker currently does not allow users to specify temporal properties.

Results We ran Dara's explorer with three different exploration strategies - *Random*, *Unique*, and *Frequency* and explored 50 different paths with each strategy. Note that the specific paths explored are a function of the exploration strategy, so a strategy's quality is expressed in how quickly it finds the bugs; *e.g.*, strategies that find bugs in paths 1-3 are better than those that find bugs in paths 4-6 Figure 6.5 shows the path numbers where the explorer found bugs with each strategy. With the *Unique* strategy, Dara found the bug in the first path and out of the 50 paths it explored, it was able to find the bug in 42 paths. Additionally, Dara found the bug in the first 7 paths it explored. Similar to the *Unique* strategy, with the *Frequency* strategy, Dara found the bug in the first path and out of the 50 paths it explored, it was able to find the bug in 40 paths. With the *Random* strategy, Dara finds the bug in the third path and in 34 out of 50 different paths explored. But, Dara found the bug in only six of the first ten paths it explored with the *Random* strategy. This suggests that the coverage-based strategies explore more buggy paths than *Random*

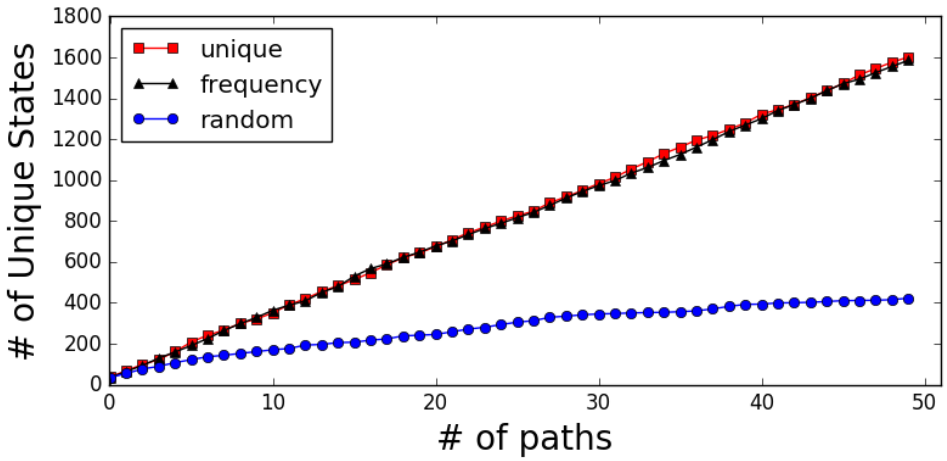


Figure 6.6: Number of states explored by each exploration strategy as a function of the number of paths

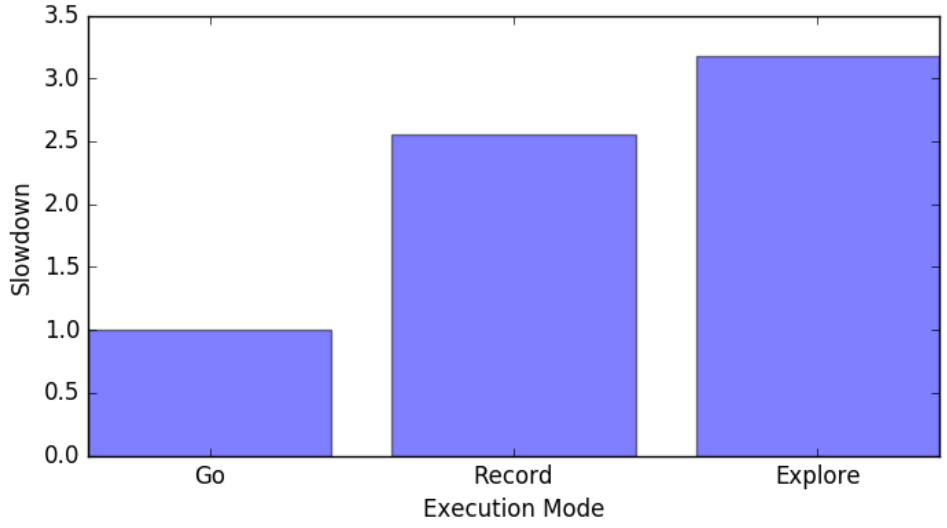


Figure 6.7: Performance Overhead for Dining Philosophers during exploration

strategy and can find the bugs faster than the *Random* strategy.

Figure 6.6 shows the number of unique states covered by the explorer for each strategy. We define a unique state in terms of the uniqueness of basic blocks and the frequency with which they are covered. For any two given states A and B, we consider the system in state A and the system in state B to be in the same state if the system has executed an identical of basic blocks and it has executed each block the same number of times. For dining philosophers, the coverage based strategies reach a higher number of unique states as compared to the number of unique states reached by the *Random* strategy in the same number of runs.

Figure 6.7 shows the performance overhead of running the Dining Philosophers program with Dara's explorer as compared to running the program with an unmodified Go runtime. For this program, Dara's explorer only experiences a 3.2x slowdown for each path explored in the program. The major bottleneck is the cost of scheduling the chosen actions by the explorer due to the high communication cost between the global and local scheduler. As this example has multiple calls to "time.Sleep", Dara is able to take advantage of the virtual clock to optimize the execution by fast forwarding the clock of the application during exploration. Thus, the amount of slowdown or speedup during exploration is dependent on virtual clock optimizations.

6.4 Performance cost of property checking

Methodology To measure the cost of property checking, we run the property checker in an isolated environment (without Dara's explorer/global scheduler/local scheduler) and emulate the values for the different variables required by the context. We run the property checker with four different property files containing 1,2,5, and 10 properties respectively. We measure three different things - (i) build time - time taken for the property checker to parse the input file and build the properties as executable plugins; (ii) load time - time taken for the property checker to load the built property plugins into memory; (iii) check time - the time taken for the property checker to check a property with a given context. We execute the build procedure and the load procedure only once while executing the checking procedure 100000 times. Between each execution, the context for the property checker is

System	Commit Hash	Total Blocks	Execution Time (in s)
Kubernetes	35c8fece86	426308	49.674
etcd	a621d807f0	52111	5.367
CockroachDB	8ee22d0f8d	25393	3.349
BoltDB	fd01fc79c5	2588	0.816

Table 6.1: Instrumentation execution time for popular real go systems

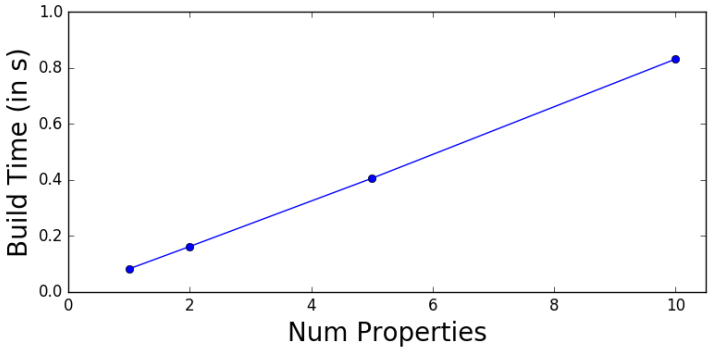
modified such that all of the variables have their values updated. This ensures that the property checker is not reaping any caching benefits.

Results Figure 6.8 shows the results of running our experiment described above. From the figures, it is clear the increasing the number of properties increases the buildtime, loadtime, and the checking time linearly. However, even for ten properties, it takes less than 1 second to build the plugins for the properties, takes under 6 milliseconds to load the plugins for all the properties, and takes about 80 microseconds for checking all the properties for a given context on average. Dara only has to build and load the plugins once for the whole exploration process. **Thus, the property checker is efficient as it only takes nanoseconds to evaluate whether a property is violated or not under a given context.**

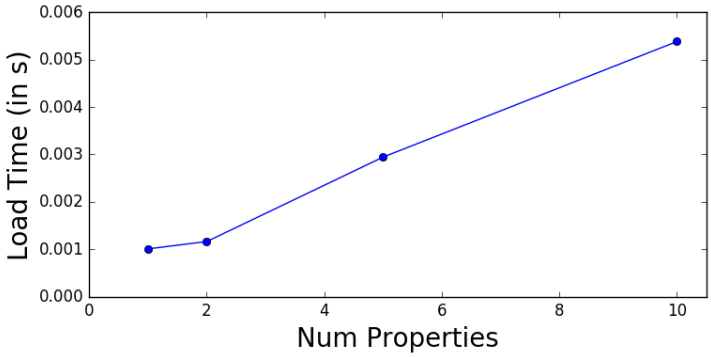
6.5 Performance cost of instrumenting the system

Methodology To measure the cost of instrumenting a system, we select four representative, real-world software systems in Go, including one container systems (Kubernetes), one distributed key-value store system (etcd), and two databases (CockroachDB and BoltDB). We select these applications for 2 reasons: (i) these applications are open-source projects that are popular on Github and have gained traction and wide usage in real datacenter environments [39]. and (ii) all of these applications have been targets of a prior study for understanding concurrency bugs in real-world go systems [39].

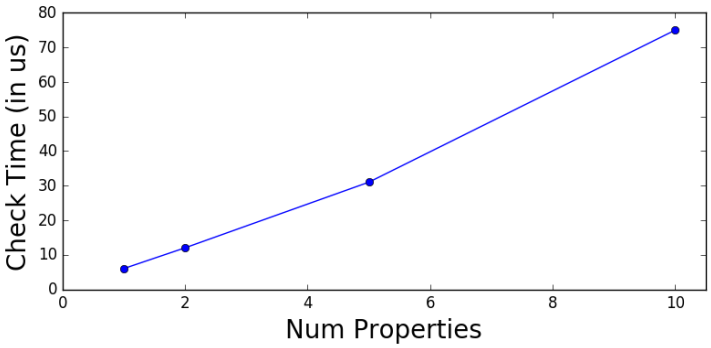
Results Table 6.1 shows the results for running Dara's instrumenter on the popular go systems selected above. Kubernetes is the system with the largest codebase with over 400K unique code blocks. However, despite the large size, it takes Dara's



(a)



(b)



(c)

Figure 6.8: (a) Increase in build time with increase in number of properties in the property file; (b) increase in load time with increase in number of properties in the property file; and (c) increase in checking time with increase in total number of properties

instrumenter less than a minute to instrument the full codebase. Both etcd and CockroachDB are medium-sized systems and it takes the instrumenter less than 6 seconds to instrument each of them whilst for the tiny BoltDB system it takes the instrumenter less than a second to instrument the full system. Dara's instrumenter is efficient and can instrument large codebases under a minute.

6.6 Performance cost of modifying the go runtime

6.6.1 Cost of Intercepting System Calls

To measure the cost of system call interception, we make use of go's benchmarking facility provided by the "testing" package. We write a benchmarking function wrapper for each system call where the system call is executed thousands of time and the execution time is measured. We measure only the cost of interception without including the cost of writing the system call information to the shared memory between the local schedulers and the global scheduler. We disable the writing to shared memory for this experiment. The cost of writing to shared memory is measured separately in the next section. First we run the benchmarking functions with an uninstrumented go (version 1.10.4) and then with our modified go runtime, dgo. The results of running the benchmarking are shown in Figure 6.9 ,Figure 6.10, Figure 6.11, Figure 6.12. For the benchmarks that show a decrease in execution time for a system call for dgo as compared to go, this is because of network related timing nondeterminism during the benchmarking process. The biggest slowdown is for GetPageSize and Executable system calls as they have slowed down from 2.8ns and 0.28ns to 54ns each. This is due to the fact that the interception code takes around 40-50ns to run. **The median interception cost across all system calls is 16%.**

6.6.2 Cost of Writing Events to Shared Memory

Dara's local schedulers and the global scheduler communicate with each other by writing information and commands to shared memory. The local scheduler must perform a write to shared memory every time an event, described in Table 5.3, occurs in the system. To measure the cost of writing event information to shared

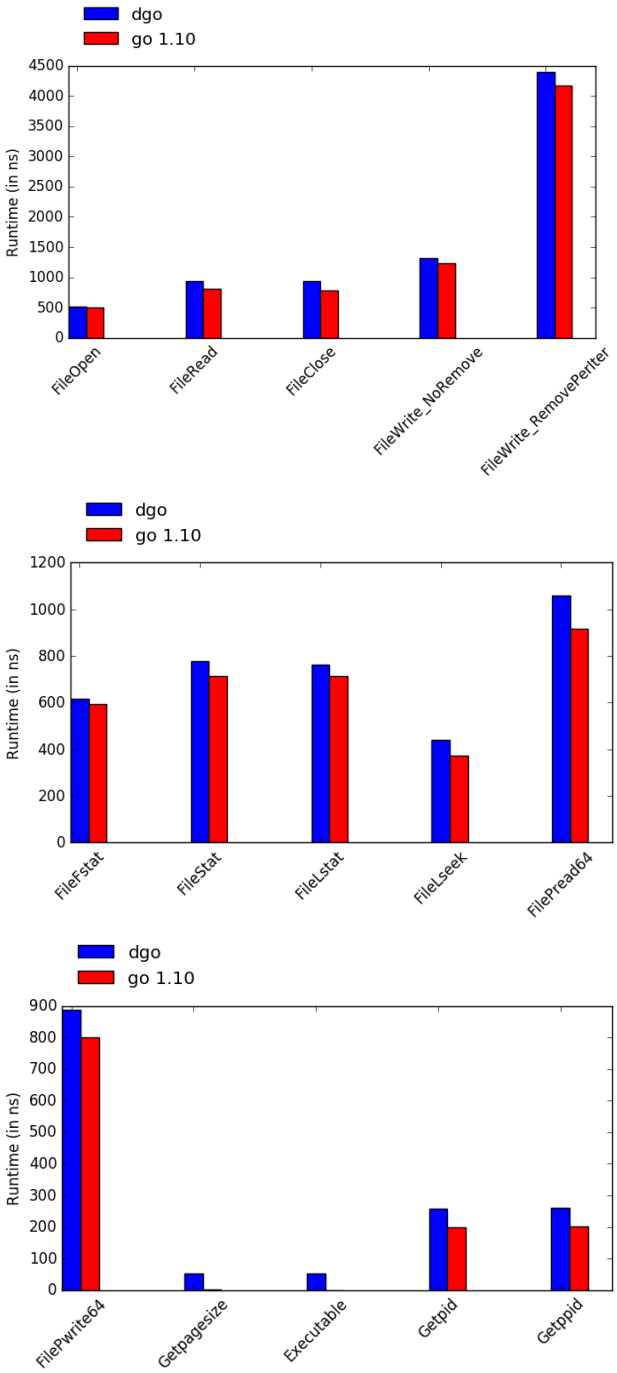


Figure 6.9: System Call performance comparison between go1.10.4 and dgo - Part 1

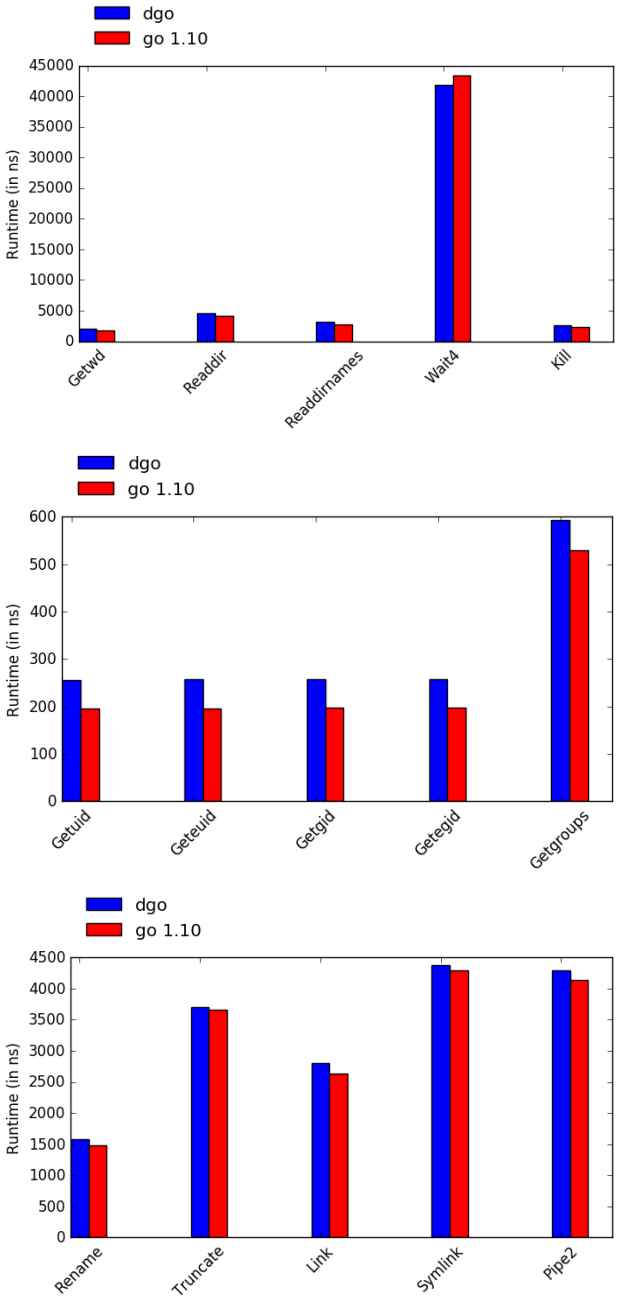


Figure 6.10: System Call performance comparison between go1.10.4 and dgo - Part 2

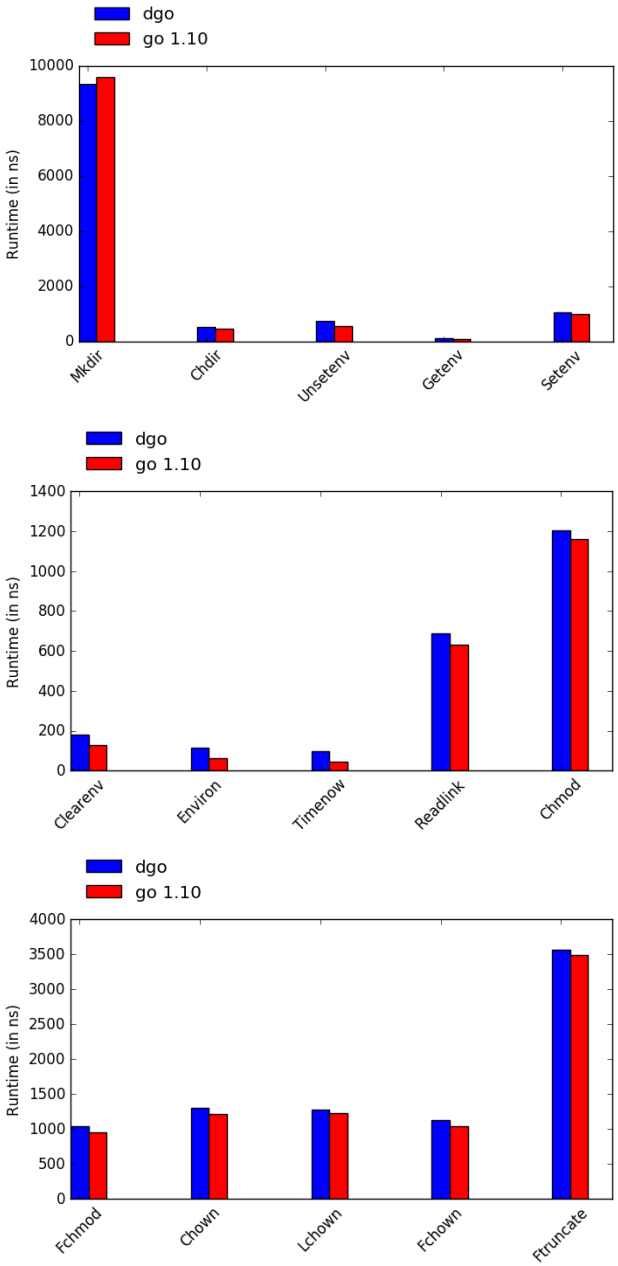


Figure 6.11: System Call performance comparison between go1.10.4 and dgo - Part 3

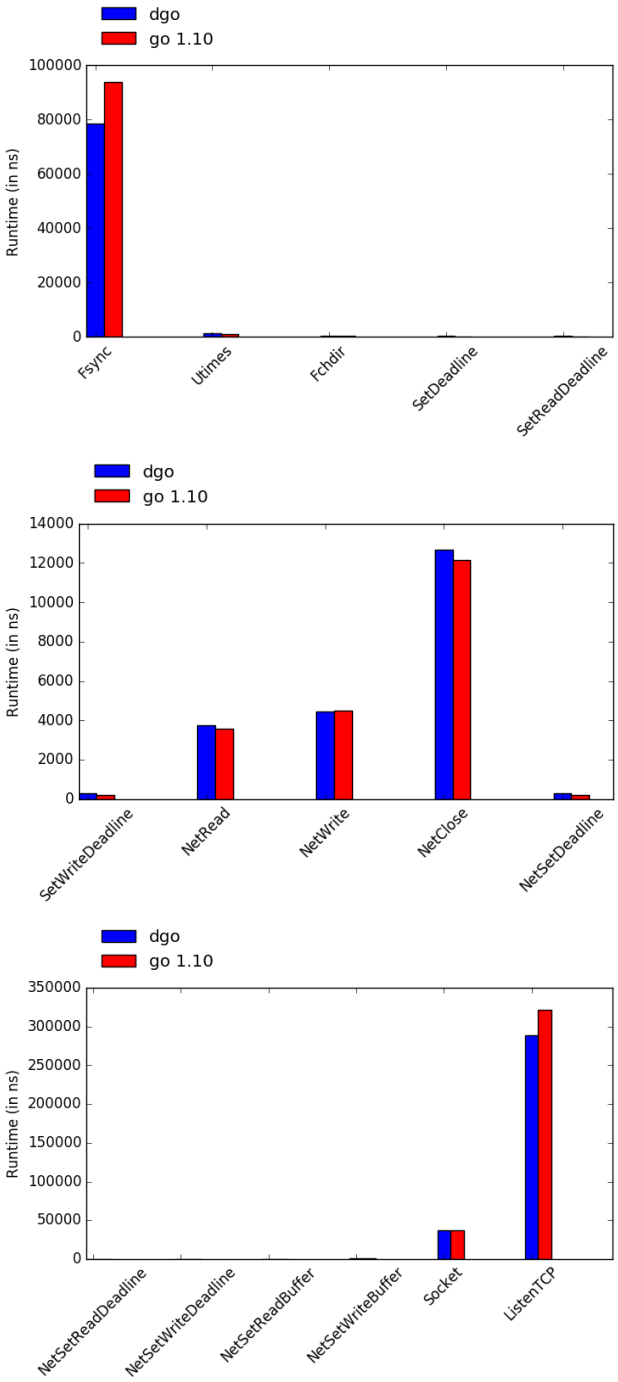


Figure 6.12: System Call performance comparison between go1.10.4 and dgo - Part 4

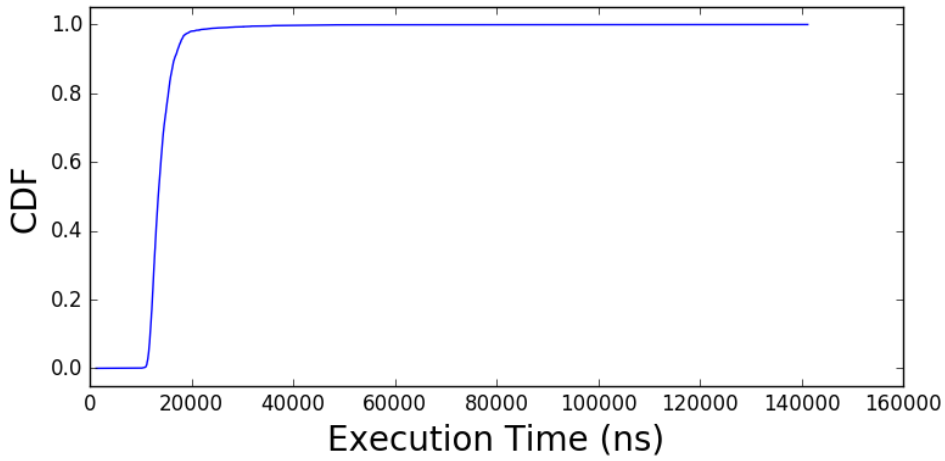


Figure 6.13: CDF of time taken to write record event information to shared memory

memory, we select an application that makes eight different system calls in its execution. We use system calls as a way to measure this cost as every write to shared memory for any given event must go through the same interace. We execute the program with Dara (recording only executions, no exploration) 1000 times to capture the amount of time taken to write the event information to shared memory.

Figure 6.13 shows the readings we observed from the 1000 different executions of the application. On average, writing the event information to shared memory takes 14.161 microseconds with a standard deviation of 3.98 microseconds. This is slow but is required the events must be reported for the global scheduler.

6.6.3 Cost of Recording Coverage Information

To measure the cost of recording coverage information in the runtime, we select an application that executes four different basic blocks in its execution and execute it with Dara (only recording executions, no exploration) 1000 times to capture the amount of time taken to write coverage information to shared memory.

Figure 6.14 shows the readings we observed from the 1000 different executions of the application. On average, recording the coverage information takes 369 nanoseconds with a standard deviation of 354 nanoseconds. The high standard

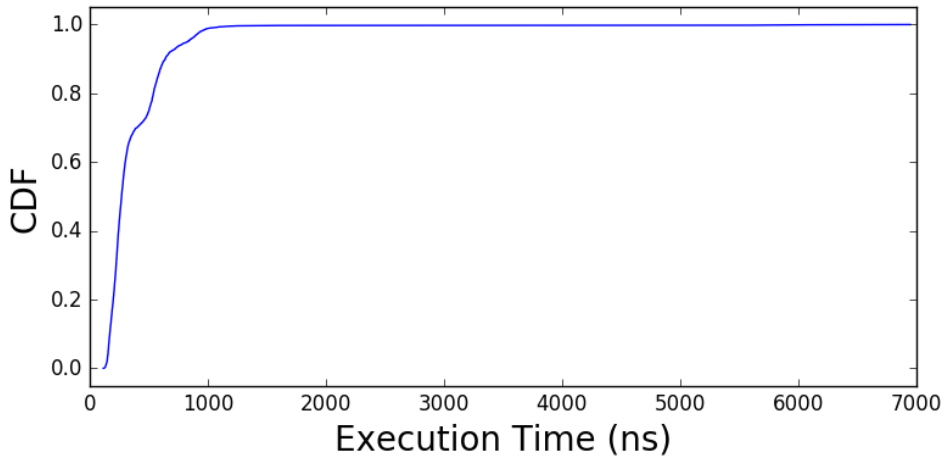


Figure 6.14: CDF of time taken to write record coverage information

deviation is due to some updates for coverage taking around 3-4 microseconds. We are unsure as to why some updates take up to 3-4 microseconds for completion but we suspect it might be due to the operating system scheduling a different process for execution and subsequently switching back to the Go process during the period in which coverage information is being updated. **Thus, the cost of reporting coverage of individual blocks to the go runtime does not induce a high overhead in the system's overall runtime.**

6.7 Performance cost of scheduling actions

To measure the cost of scheduling actions by Dara, we choose an application in which the number of scheduled actions can be configured. We run this application three different ways - (i) with unmodified go runtime, (ii) with Dara's recorder where the goroutine scheduling is simply observed and recorded but not controlled, and (iii) with Dara's explorer where the goroutine scheduling is controlled. For reproducibility and fair comparison, we choose to execute the same actions that were taken in the record mode, however the local scheduler must contact the global scheduler to find the next action. We measure the runtime for the application's execution for all three ways. Figure 6.15 shows that the application runtime grows

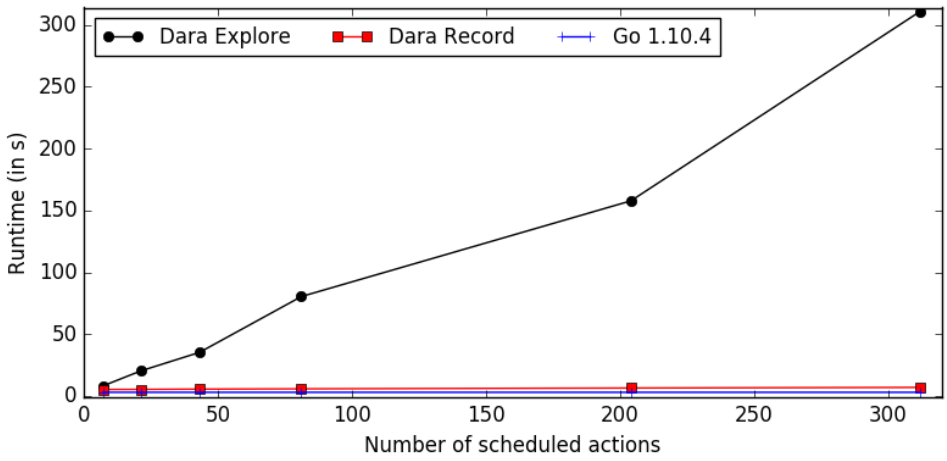


Figure 6.15: Application runtime grows linearly with the number of scheduled actions by Dara

linearly with increase in number of scheduled actions when Dara is in charge of making scheduling actions and that the local scheduler has to contact the global scheduler for actions. **Scheduling actions at local schedulers is the bottleneck for exploration as the communication between local schedulers and the global scheduler is expensive because it requires polling on shared memory by the local schedulers.**

To measure the cost of choosing actions by Dara, we ran exploration of up to 50 paths on Dining Philosophers for each exploration strategy and measured the time take by the explorer to choose the action that is to be executed next for every chosen action during the 50 paths. Figure 6.16 shows the CDF of the cost of choosing the action for different exploration strategies. Random exploration strategy takes the shortest time to decide the next action as the explorer does not need to consult prior information for choosing actions. Coverage-based strategies take nearly double the amount of time taken by Random strategy to choose the action because for every action, the coverage based strategies must consult prior information to decide the best possible action. Despite this, the explorer takes less than 10 microseconds for choosing the next action with coverage based exploration strategies.

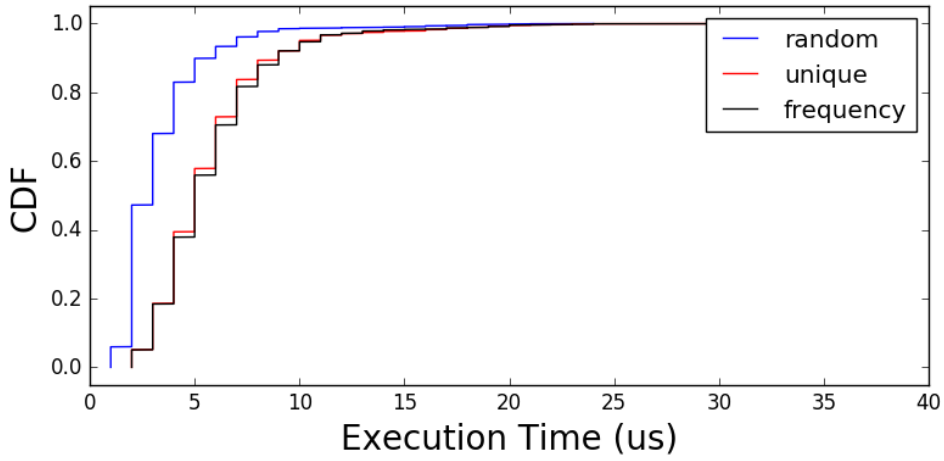


Figure 6.16: Cost of choosing the next action with different exploration strategies for exploration in Dining Philosophers

Chapter 7

Discussion

7.1 Limitations

Scalability One of the major drawbacks of the current implementation of Dara is that it can run only a limited number of nodes at a time as it runs every single node of the distributed system on the same machine. As there is a limit to the number of total nodes that can be run on a single machine, Dara can not scale to systems with thousands of nodes which are pervasive in large-scale cloud systems. However, this can be rectified by distributing the nodes across multiple machines and having the local schedulers communicate with the global scheduler through an intermediary agent at each node where the agent communicates with the local scheduler over shared memory and with the global scheduler over the network.

Property Checking Allowing users to write properties as go functions is a trade-off between the amount of effort required to specify the property and the expressive power of the properties that can be specified. Dara's current property checker has no notion of temporal properties as the current property specification interface provides no way of incorporating control flow information of the program. Moreover, the current property checker requires the user to instrument their program to report values of variables to Dara which adds to the developer effort required to use Dara.

Usability Analysis We would have liked to have performed a user study to understand the Usability of Dara as well as the auxiliary tools to be used for understanding buggy execution traces but due to time constraints we were unable to

perform a thorough usability analysis of Dara. This remains an important part of our future work.

Comparison with state-of-the-art The current evaluation of Dara does not include a comparison of Dara's exploration strategies with the current state-of-the-art exploration strategies. This is because the current exploration strategies are not open-source and need to be implemented in Go to carry out a comparative experiment. Due to time constraints, I was unable to complete this experiment.

7.2 Future Work

Environment Exploration Currently, Dara controls coverage of code through exploration of different schedules of actions. However, certain code blocks can be exercised and executed only if something has gone wrong in the environment or due to a misconfiguration issue or due to random faults. Dara does not control exploration of code coverage that may be caused due to environmental failures but we believe that once Dara incorporates this, it will find more bugs.

Stop-the-world distributed debugger The world is in a dire need of a stop-the-world debugger for Go based distributed systems. Dara in its current form falls short of being a stop-the-world debugger as it still is not able to expose stack and variable information the way tools like GDB expose such information to the user. With that being said, Dara has all the necessary prerequisites to serve as the basis for a stop-the-world debugger. Currently, it can control the execution and scheduling of any given goroutine on any node and the sending/delivery of messages across channels or the network. It also has access to the stack information for every goroutine but it is not exposed to the users of Dara. The one major piece missing from Dara is the access to debugging symbols for reading values of variables at runtime. However, this information is available in the binaries generated by go as DWARF debugging information [7] and is already in use by debuggers such as gdb and delve [15].

Expressive Property Interface The current property specification interface does not provide a way for the users to specify temporal properties. Moreover, to be as powerful as abstract MCs, the property specification must support Linear Temporal Logic (LTL) [12, 36] property checking where the users can specify properties about the system using LTL formulae. Moreover, the Global Scheduler has access

to performance information for every single node that is not currently exposed to the application via dgo but can be used in specifying performance properties. The property interface can be further developed to allow users to specify security properties about the system.

Composing Exploration Strategies There has been a plethora of work done in mitigating state space explosion in model checkers [13, 28, 31]. Currently, Dara's exploration strategy does not incorporate these into its coverage-based exploration strategies. We believe that combining Dara's coverage based strategies with existing strategies for combating state-space explosion is a fruitful future endeavour.

Combining Abstract and Concrete MCs Dara's exploration strategy doesn't completely eliminate the state space explosion problem. It just provides a better way of exploring the state space so that more bugs can be unearthed in fewer runs of the system. However, model checkers that work on abstract models are much faster and efficient at exploring the state space. We believe that there is definitely some merit in combining the speed of abstract model checkers with concrete model checkers like Dara for efficient bug hunting.

Chapter 8

Conclusion

In this thesis, we have presented Dara, a concrete MC specifically designed for Go. We discussed the design choices that we made to implement a concrete MC for Go such as interposing on the Go runtime to control goroutine creation and scheduling, interposing on Go's time library to control the firing of timers and to provide a consistent view of time to the application during exploration, controlling the various sources of nondeterminism in Go systems, and the design of APIs exposed to users for capturing variable values for property checking. Dara is also powered with three novel coverage-based exploration strategies for searching for crashes and violations of user-defined properties throughout the state space of a distributed system. The key idea behind Dara's exploration strategy is to explore those paths first which maximize the code coverage of the system. These are not the only possible coverage-based exploration strategies that are possible. We believe that there exist many such coverage-based strategies, each of which would potentially be useful in finding a different class of bugs. Ultimately, we believe that there is no one strategy that will work best for all possible programs and systems but a mixture of various strategies would be able to discover different kinds of bugs. Since finding a bug is not enough for a user to reproduce or find the root cause, we also provide a suite of auxiliary tools with Dara including a deterministic replay engine for replaying and understanding buggy execution traces.

Bibliography

[1] Basic block. https://en.wikipedia.org/wiki/Basic_block. Accessed: 2020-08-17. → pages 19, 25

[2] C. Artho, Q. Gros, G. Rousset, K. Banzai, L. Ma, T. Kitamura, M. Hagiya, Y. Tanabe, and M. Yamamoto. Model-based api testing of apache zookeeper. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 288–298. IEEE, 2017. → page 4

[3] A. Bakst, K. v. Gleissenthall, R. G. Kıcı, and R. Jhala. Verifying distributed programs via canonical sequentialization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):110, 2017. → page 6

[4] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013. → page 2

[5] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst. Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–38, 2020. → pages 37, 40

[6] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu. Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. *arXiv preprint arXiv:2007.15943*, 2020. → page 7

[7] D. D. I. F. Committee et al. Dwarf debugging information format, version 4, 2010. → page 67

[8] J. Condliffe. Amazon's $150 million typo is a lightning rod for a big cloud problem. *MIT Technology Review*, Mar. 2017. https://www.technologyreview.com/s/603784/, accessed Aug. 21 2018. → page 1

[9] P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal, R. Mudduluru, S. Qadeer, et al. Uncovering bugs in distributed storage systems during testing (not in production!). In *FAST*, pages 249–262, 2016. → page 4

[10] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *PLDI 2013*. → page 1

[11] E. W. Dijkstra. Two starvation free solutions to a general exclusion problem. *Unpublished Tech. Note EWD*, 625, 1978. → page 49

[12] E. A. Emerson. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter Temporal and Modal Logic, pages 995–1072. J. van Leeuwen, ed., North-Holland Pub. Co./MIT Press, 1990. → page 67

[13] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices*, 40(1):110–121, 2005. → pages 25, 68

[14] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global Comprehension for Distributed Replay. In *Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA, 2007. → page 6

[15] go delve. Delve: A debugger for the go programming language, 2020. → page 67

[16] Golang. Gocoverage tool. https://golang.org/cmd/cover/. → page 36

[17] R. Guerraoui and M. Yabandeh. Model checking a networked system without the network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 225–238, Berkeley, CA, USA, 2011. USENIX Association. → page 4

[18] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. 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. → page 4

[19] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving Practical Distributed Systems Correct. In *SOSP 2015*. → pages 1, 6

[20] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. → pages 19, 25

[21] G. J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. → page 1

[22] jepsen io. Jepsen. https://github.com/jepsen-io/jepsen. → page 6

[23] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: finding liveness bugs in systems code. In *Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA, 2007. → pages 4, 5

[24] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 17–26, New York, NY, USA, 2010. ACM. → page 5

[25] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994. → page 5

[26] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and verifying systems with TLA+. In *EW 2002*. → page 1

[27] F. Laroussinie and K. G. Larsen. Cmc: A tool for compositional model-checking of real-time systems. In *Formal Description Techniques and Protocol Specification, Testing and Verification*, pages 439–456. Springer, 1998. → page 4

[28] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *OSDI*, pages 399–414, 2014. → pages 2, 4, 5, 25, 68

[29] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: Certified causally consistent distributed key-value stores. *SIGPLAN 2016*. → pages 1, 6

[30] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3S: Debugging Deployed Distributed Systems. In *Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, USA, 2008. → page 6

[31] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, et al. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019. → pages 2, 4, 5, 25, 68

[32] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, pages 267–280, 2008. → page 4

[33] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, Mar. 2015. → page 5

[34] novalagung. go-eek. https://github.com/novalagung/go-eek, 2020. → page 35

[35] A. Panda, M. Sagiv, and S. Shenker. Verification in the age of microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 30–36. ACM, 2017. → page 5

[36] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977. → pages 8, 67

[37] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, page 9. USENIX Association, 2006. → page 6

[38] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing Faulty Executions of Distributed Systems. In *NSDI*, 2016. → page 6

[39] T. Tu, X. Liu, L. Song, and Y. Zhang. Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 865–878, 2019. → pages 13, 43, 48, 55

[40] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI 2015*. → pages 1, 6

[41] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *The 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, 2009. → page 4

[42] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *NSDI 2009*. USENIX Association. → pages 2, 4, 9

[43] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *OSDI*, pages 249–265, 2014. → page 6

[44] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea. A formally verified nat. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 141–154. ACM, 2017. → page 5