# UNDERSTANDING AND EVALUATING DYNAMIC RACE DETECTION WITH GO

An Undergraduate Research Scholars Thesis

by

ANDREW H. CHIN

Submitted to the Undergraduate Research Scholars program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Jeff Huang

May 2020

Major: Computer Science

# TABLE OF CONTENTS

# ABSTRACT

Understanding and Evaluating Dynamic Race Detection with Go

Andrew H. Chin
Department of Computer Science & Engineering
Texas A&M University

Research Advisor: Dr. Jeff Huang
Department of Computer Science & Engineering
Texas A&M University

With the prevalence of concurrent software and the difficulties that come with properly synchronizing threads, it is easy to introduce data races into programs. A data race is a software bug which occurs when at least two threads simultaneously access shared memory, with at least one of them performing a write. Because of the indeterministic nature of thread scheduling, data races lead to undefined behavior which may only manifest in rare thread schedules. The difficulty lies in detecting hidden data races based on observed program executions.

Go (Golang) is a modern, open source programming language designed for simplicity, efficiency, and reliability. With the introduction of Goroutines and other primitives, Go enables quick development of highly concurrent software. However, this model makes it necessary to apply race detection to programs written in Go.

We present an in-depth study of the Go race detector, a dynamic analysis tool developed by Google. The Go race detector traces program events and provides detailed information regarding detected races. Furthermore, we evaluate the race detector on Kubernetes, the most popular open source project written in Go.

# ACKNOWLEDGMENTS

I would like to thank my faculty advisor, Dr. Jeff Huang, for being extremely patient, reviewing my thesis, and guiding me as I pursued my research. I would like to thank my colleague, PhD student Yahui Sun, for working very closely with me and also reviewing my thesis. I also would like to thank the Automated Software Engineering Research group for their hospitality during the past two years. Finally, I would like to thank my family and friends for their encouragement during difficult times and their unconditional support.

# NOMENCLATURE

| | |
|---|---|
| Go | Go programming language |
| TSAN | Thread Sanitizer |
| HB | Happens-Before relationship |
| RTL | Run-Time Library |
| LLVM | LLVM Compiler Infrastructure Project (not an acronym) |
| GitHub | A Platform used to host source code, perform version control, track issues, and collaborate on projects |
| AST | Abstract Syntax Tree |
| SSA | Static Single Assignment form |
| VM | Virtual Machine |

# CHAPTER I

# INTRODUCTION

In modern computing, concurrency is crucial to both the development and performance of software applications and systems. However, as the level of concurrency increases, so does the complexity. When the number of threads increases, the space of possible thread interleavings explodes, making it impossible to reason about. Because of this, it is easy to unknowingly introduce data races. A *data race* occurs when at least two threads access shared memory concurrently, with one of them performing a write operation. Data races are dangerous because they lead to undefined behavior. Because data races arise as a result of nondeterminism, it becomes necessary to create ways to automatically detect these kinds of bugs.

In 2009, Google introduced *ThreadSanitizer* [1, 2], a precise data race detector for programs written in C/C++. TSAN uses dynamic program analysis to partially order the events of a program execution over the *Happens-Before relation* [3]. If two threads access the same memory with one of them performing a write and are unordered by Happens-Before, then they are *concurrent*, causing TSAN to report a data race. TSAN consists of a compiler instrumentation library that generates information about program events and a state machine that detects data races at runtime. TSAN uses a heavily optimized [4] vector clock implementation to track memory accesses, making it the most well-known, fast (relatively) data race detector used in practice.

Also in 2009, Google released *Go* [5], an open-source programming language designed for the creation of simple, highly concurrent code. Go's core design consists of two features: goroutines and channels. A *goroutine* is a lightweight unit of execution, similar to threads in C++ or Java. A *channel* is a synchronization primitive that orders two goroutines through message

passing. Although the design principles are heavily catered towards concurrent software, a recent study [6] found that Go may potentially introduce more concurrency bugs than classical languages like C++. Thus, there is still a need for automated detection of concurrency bugs, including data races. TSAN has been officially ported to Go as the Go race detector. In the context of Go, a data race occurs when two goroutines access the same variable concurrently and at least one of them is a write.

This paper makes the following contributions:

- We present an in-depth study of the Go race detector internals, specifically targeting the compiler instrumentation. We study the optimizations implemented for reducing redundant instrumentation and compare them to the more mature optimizations implemented in LLVM TSAN. In summary, we suggest one redundancy pattern which may reduce the overall instrumentation by up to 33%.
- We present the first public (to our knowledge) empirical evaluation of the Go Race Detector on a real-world application, Kubernetes.
- We introduce and discuss a real data race reported during our evaluation of the data race detector.

# CHAPTER II

# MOTIVATING EXAMPLE

First, we present a small, motivating example showing a data race in Go, show in Figure 1. In this simple Go program, there are two functions and a global integer variable *x*, which is set to 0. The function *addOne* simply takes the variable *x* and adds 1 to it. In function *main*, we start two goroutines on lines 8 and 9, which will concurrently execute the *addOne* function. What will be the value of *x* in the print statement on line 12? Intuitively, it should be 2. However, it is possible for *x* to have a value of 1 at the end of the program. How is this possible?

```
1   var x int = 0
2
3   func addOne() {
4     x = x + 1
5   }
6
7   func main() {
8       go addOne()
9       go addOne()
10
11      // wait for both goroutines to finish
12      fmt.Printf("value of x: %d\n", x)
13  }
```

Figure 1. Simple Go program exemplifying a data race between two goroutines.

First, we must understand how the code $x=x+1$ is actually translated by a compiler. It is actually broken into multiple instructions: the value of x is read and stored into a temporary variable, the temporary variable is incremented by 1, and then *x* is set to the value of the temporary variable. As you can see from the left half of Figure 2, the goroutines executions may interleave.

This is a data race, as they concurrently access the variable *x*, with one of them performing a write operation.

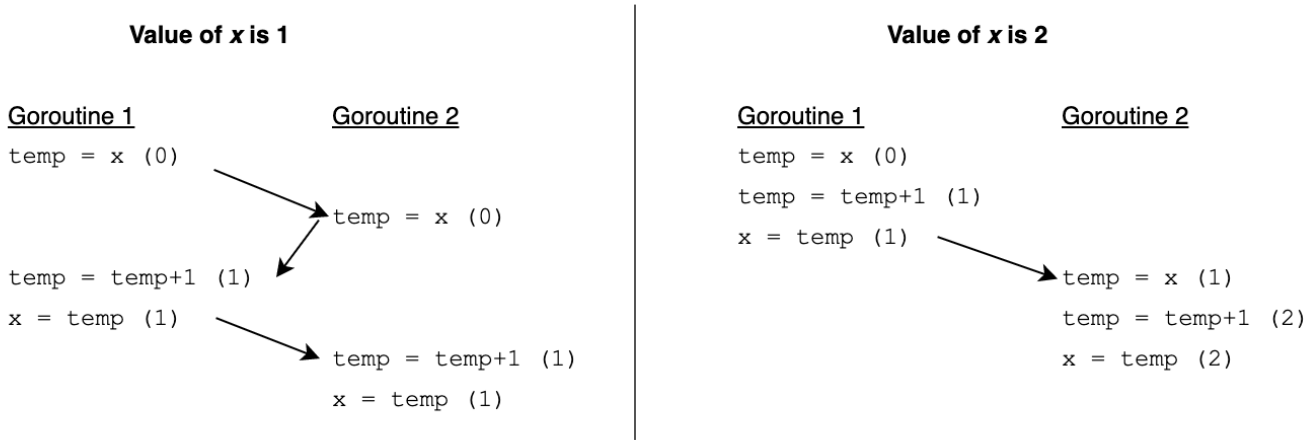


Figure 2: Go program execution which results in *x* being 1.

Data races are dangerous because they are elusive and may lead to undefined behavior. As shown in the example, a program may exhibit illogical behavior leading to hard-to-explain software or system failures. How might we resolve a data race? In this example, we can easily wrap function *addOne* with a mutex lock (Figure 3). This synchronization primitive prevents instruction interleaving as show in the left half of Figure 2 by enforcing atomicity of the three instructions corresponding to *x=x+1*.

```
func addOneThreadSafe() {
  mutex.lock()
  x = x + 1
  mutex.unlock
}
```

Figure 3: Mutex lock which prevents a data race on variable *x*.

# CHAPTER III

# UNDERSTANDING THE GO RACE DETECTOR

The *Go race detector* is a powerful data race detector which uses dynamic program analysis to verify the correctness of concurrent programs. It is integrated into the Go compiler, which is available at the Go GitHub repository [5]. The goal of this chapter is to deeply understand the internals of the race detector: the architecture and race detection instrumentation.

**Architecture**

Like TSAN, the Go race detector consists of a compiler instrumentation library and a runtime state machine. However, the runtime state machine is compiled from the original LLVM project [7] and linked to the Go runtime as an external library. Thus, the Go compiler only needs to reimplement the instrumentation library. The purpose of the instrumentation library is to provide the runtime state machine with instruction metadata required to track the ordering of events. For details on the state machine, see [8].

**The Go Compiler**

Before getting into race instrumentation, we present a primer on the Go compiler. The purpose of a compiler is to translate programs written in a high-level programming language (Go, C/C++, Java) into instructions which can be executed by a processor. Like most compilers, the Go compilation process is logically split into four phases (Figure 4):

1. Parsing
2. Type-checking and AST transformations
3. Generic SSA
4. Generating machine code

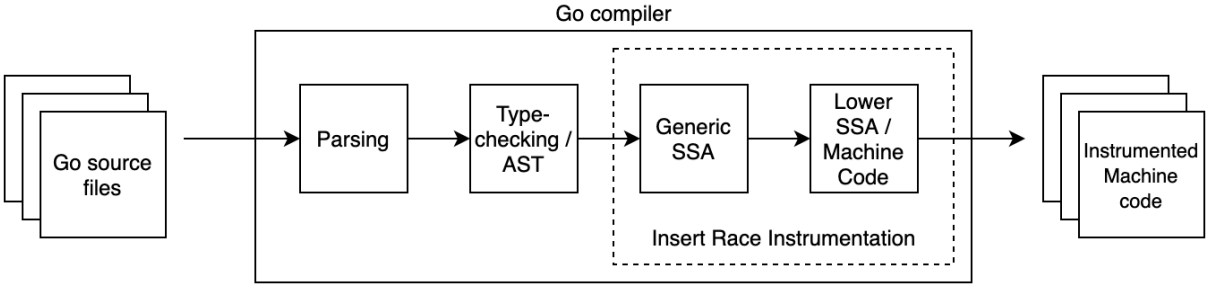


Figure 4: Race detection instrumentation in the Go compiler.

In the first phase, the source code files are tokenized, parsed, and converted into abstract syntax trees. Each syntax tree is an exact representation of its source file with expressions, declarations, and statements. In the second phase, the abstract syntax tree is type-checked and transformed. This includes performing the first round of dead code elimination, function call inlining, and escape analysis. In the third phase, the full abstract syntax tree is converted to generic SSA (Static Single Assignment) form, a lower-level intermediate representation which makes it easy to perform machine-independent optimization passes. Finally, in the fourth phase, the generic SSA is lowered into machine-specific SSA, optimized specifically for the current architecture. From this point, it is straightforward to translate optimized SSA into machine code.

For race detection, the compiler is modified to insert special race instrumentation functions during phases 3 and 4: Generic and Machine-Specific SSA.

**Instrumentation for Data Race Detection**

For data race detection, the Go compiler instruments Go source code by inserting additional function calls in order to record program events. By providing instruction metadata to the runtime state machine, the race detector can maintain the happens-before ordering over events and detect data races (Figure 5). Special race functions are inserted before the following events:

reads, writes, function entry and function exits, synchronization acquires, synchronization releases, process create and destroys, dynamic memory allocations and frees, and atomic operations. To accomplish this goal, the compiler inserts instrumentation in two ways: manual instrumentation at the source code level and compiler instrumentation at the SSA level.

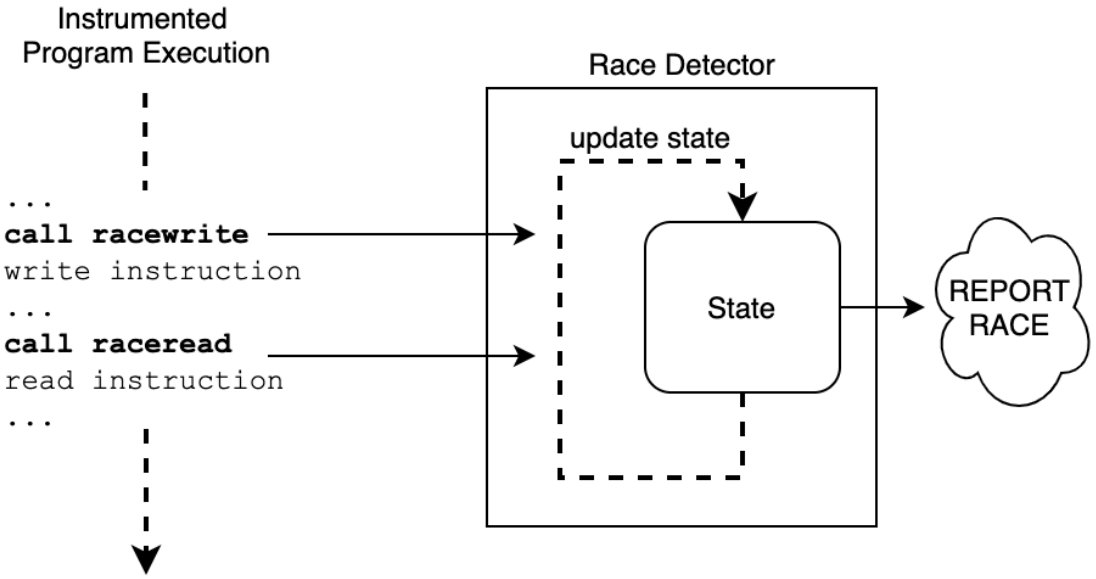


Figure 5: Visualization of the Go race detector at runtime.

*Manual Instrumentation*

Go source code can be manually instrumented by a developer using the *race* package. This package is a standard library of Go functions that can be added into code to influence the behavior of the data race detector. Additionally, the manual source code approach is used to provide race detection capabilities to built-in Go synchronization primitives and containers residing in the Go *runtime* package. Some objects that are instrumented include *Channels*, *Mutex*, *RWMutex*, *WaitGroups*, *Maps*, *Strings,* and architecture-specific *atomic instructions*.

*Compiler Instrumentation*

Whereas manual instrumentation is inserted by a developer at the source-code level and explicitly translated, compiler instrumentation is inserted by the compiler at specific program events. Compiler instrumentation occurs during the construction of the generic SSA, with a few race function optimization passes occurring as a normal SSA pass. During the building of the generic SSA, the compiler prepends *racewrite* and *raceread* functions before memory loads and stores (zero, move, storeType instructions), respectively. For composite loads and stores (arrays), there is *racereadrange* and *racewriterange.* These functions all pass the address and the length of the value being accessed to the runtime state machine.

**Optimizations**

In order to keep data race detection as lightweight as possible, instrumentation that can be proven as redundant should be removed. The Go compiler has implemented a number of optimizations. First, during instrumentation, functions are prepended and appended with *racefuncenter* and *racefuncexit*, respectively. These functions serve to restore stack traces on reports. Thus, in functions with no instrumented instructions, we can safely remove these two calls. This is implemented as an SSA pass. Second, when the compiler is instrumenting memory loads and stores, it can safely skip stack addresses, read-only fields (Itabs, string data, and closure fields), and addresses in the static read-only data segment, as they are guaranteed to be data race free. Third, certain instruction patterns are proven to be redundant. A read instruction before a write instruction is redundant if they are within the same basic block and there are no calls in between them. This is because a race on the read will also race with the write. Finally, redundant instrumentation optimization does not need to be too aggressive, as many redundant patterns will not survive

the classical compiler optimizations. For example, two reads from the same temp will be removed by common subexpression elimination and two writes to the same location will be removed by dead-store elimination.

**Discussion**

We compared the LLVM TSAN instrumentation library with the Go instrumentation library. Since the Go race detector is not as mature as TSAN, it is expected to be lagging behind in terms of development. In particular, we noticed one instruction pattern optimization present in TSAN which has not yet been implemented in the Go race detector [9]. Uncaptured pointers are guaranteed to be safe as they will not escape the current function. In this case, capturing a pointer means that no part of the value will be returned from the function nor stored in memory elsewhere. In TSAN, this optimization reduces the amount of instrumentation by up to 33% [9].

# CHAPTER IV

# EVALUATING THE GO RACE DETECTOR

Now that we understand the internals of the Go race detector, we seek to evaluate it on real-world large Go programs. The objectives of our experiments are to measure the performance overhead of software instrumented for races in combination with the Go race detector. We also aim to measure the size increase due to race instrumentation.

**Methodology**

We have selected Kubernetes [10] for our evaluation. Kubernetes is an ecosystem of tools and services used to coordinate a highly available cluster of computers that are connected to work as a single unit [11]. Kubernetes provides automated distribution and scheduling capabilities for containerized applications. Kubernetes also provides other features such as deployment, monitoring, and recovery. Kubernetes is production-grade, and is used by many large, successful companies, such as Adidas, eBay, The New York Times, etc. [12]. Additionally, since we are constrained to a single machine for testing, we use Minikube [13], a tool which runs a single-node Kubernetes cluster inside a VM. Minikube enables quick development and local testing of Kubernetes, and is also widely used by developers.

All of our experiments were conducted on a 6-core 2.6 GHz Intel Core i7 machine with 16 GB of memory running macOS 10.14.6. We used Go version 1.13.8, cloned Kubernetes from the master branch on April 2, 2020 and used Minikube version 1.7.2.

## Performance

*Minikube*

Figures 6 and 7 summarize the results of our performance experiments on Minikube. We measured the time required to execute Minikube commands with and without race instrumentation. For each command, we averaged the result of 10 independent runs. We see that on average, the Go race detector incurs 1-7x slowdown compared to the native application with no race detection.

| | Execution Time (seconds)* | | |
|---|---|---|---|
| Minikube Command | No Race Detection | Race Detection | Slowdown |
| start | 30.208 | 37.2419 | 1.23x |
| stop | 5.0994 | 6.1824 | 1.21x |
| delete | 0.3483 | 1.41422 | 4.06x |
| logs | 0.5321 | 1.7592 | 3.31x |
| pause | 0.4253 | 1.5233 | 3.58x |
| unpause | 0.4258 | 1.4848 | 3.49x |
| status | 0.1743 | 1.2466 | 7.15x |

Figure 6: Performance evaluation on Minikube.

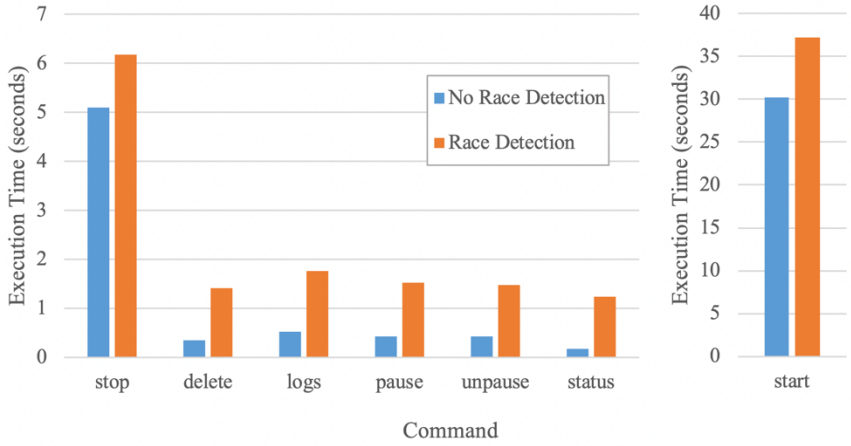


Figure 7: Performance evaluation on Minikube, visualized.

*Kubernetes*

For evaluating Kubernetes, we ran the tests available in the *k8s.io/client.go* repository [14]. These tests were selected because the *client-go* package uses many mutexes and channels, and most tests are concurrent. Figure 8 shows the distribution of tests according to their slow-down when compiled for data race detection.

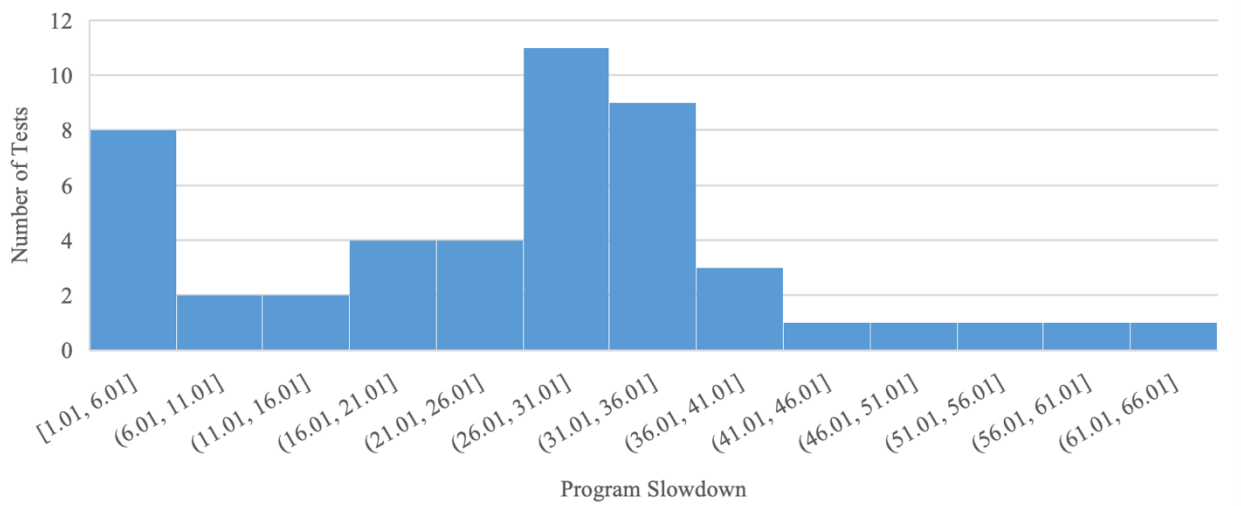


Figure 8: Distribution of Kubernetes client-go test slowdown when enabling the race detector.

We see that the majority of instrumented tests run 26-31x slower than the native test. This is quite more overhead than expected, as TSAN reports that typical program slowdown is 5-15x [15]. Figure 9 presents the full test results of our experiments on Kubernetes. Again, the execution time for each test is the average of 10 independent runs.

**Instrumentation**

We want to measure how much the size of a compiled binary increases when it is compiled for race detection. We compiled three tools in the Kubernetes family with and without race detection. These three tools are Minikube, Kops [16], and Kompose [17]. Statistics of the three programs and experiment results are shown in Figure 10.

| | Execution Time (seconds)* | | |
|---|---|---|---|
| **Test** | **No Race Detection** | **Race Detection** | **Slowdown** |
| k8s.io/client-go/discovery | 0.079 | 1.251 | 15.84x |
| k8s.io/client-go/discovery/cached/disk | 1.1584 | 2.1777 | 1.88x |
| k8s.io/client-go/discovery/cached/memory | 0.0402 | 1.0665 | 26.53x |
| k8s.io/client-go/discovery/fake | 0.0366 | 1.0613 | 29.00x |
| k8s.io/client-go/dynamic | 0.0584 | 1.1545 | 19.77x |
| k8s.io/client-go/dynamic/dynamicinformer | 0.343 | 1.3786 | 4.02x |
| k8s.io/client-go/dynamic/dynamiclister | 0.0313 | 1.0541 | 33.68x |
| k8s.io/client-go/dynamic/fake | 0.0349 | 1.0722 | 30.72x |
| k8s.io/client-go/examples/fake-client | 0.1425 | 1.178 | 8.27x |
| k8s.io/client-go/kubernetes test | 0.0362 | 1.0567 | 29.19x |
| k8s.io/client-go/listers/extensions/v1beta1 | 0.0333 | 1.0597 | 31.82x |
| k8s.io/client-go/metadata | 0.0455 | 1.0914 | 23.99x |
| k8s.io/client-go/metadata/fake | 0.0387 | 1.0701 | 27.65x |
| k8s.io/client-go/metadata/metadatainformer | 0.3441 | 1.377 | 4.00x |
| k8s.io/client-go/metadata/metadatalister | 0.0347 | 1.0572 | 30.47x |
| k8s.io/client-go/plugin/pkg/client/auth/azure | 0.0337 | 1.0461 | 31.04x |
| k8s.io/client-go/plugin/pkg/client/auth/exec | 2.2576 | 3.3269 | 1.47x |
| k8s.io/client-go/plugin/pkg/client/auth/gcp | 0.2408 | 8.3426 | 34.65x |
| k8s.io/client-go/plugin/pkg/client/auth/oidc | 0.0324 | 1.0413 | 32.14x |
| k8s.io/client-go/plugin/pkg/client/auth/openstack | 3.0387 | 4.0511 | 1.33x |
| k8s.io/client-go/rest | 0.082 | 1.2324 | 15.03x |
| k8s.io/client-go/rest/watch | 0.0538 | 1.1051 | 20.54x |
| k8s.io/client-go/restmapper | 0.0374 | 1.0583 | 28.30x |
| k8s.io/client-go/scale | 0.0747 | 1.2301 | 16.47x |
| k8s.io/client-go/testing | 0.0335 | 1.0501 | 31.35x |
| k8s.io/client-go/tools/auth | 0.0334 | 1.051 | 31.47x |
| k8s.io/client-go/tools/cache | 17.0474 | 17.2465 | 1.01x |
| k8s.io/client-go/tools/cache/testing | 0.0298 | 1.0368 | 34.79x |
| k8s.io/client-go/tools/clientcmd | 0.0689 | 1.1495 | 16.68x |
| k8s.io/client-go/tools/clientcmd/api | 0.0395 | 1.0426 | 26.39x |
| k8s.io/client-go/tools/events | 0.0363 | 1.0596 | 29.19x |
| k8s.io/client-go/tools/leaderelection | 0.0258 | 1.0903 | 42.26x |
| k8s.io/client-go/tools/pager | 0.0295 | 1.0626 | 36.02x |
| k8s.io/client-go/tools/portforward | 0.0359 | 1.0493 | 29.23x |
| k8s.io/client-go/tools/record | 0.0641 | 1.3933 | 21.74x |
| k8s.io/client-go/tools/reference | 0.0309 | 1.0412 | 33.70x |
| k8s.io/client-go/tools/remotecommand | 0.028 | 1.0457 | 37.35x |
| k8s.io/client-go/tools/watch | 12.2674 | 13.326 | 1.09x |
| k8s.io/client-go/transport | 0.036 | 1.7734 | 49.26x |
| k8s.io/client-go/util/cert | 0.0435 | 1.1256 | 25.88x |
| k8s.io/client-go/util/certificate | 0.0477 | 1.1273 | 23.63x |
| k8s.io/client-go/util/connrotation | 0.017 | 1.028 | 60.47x |
| k8s.io/client-go/util/flowcontrol | 5.1264 | 6.1397 | 1.20x |
| k8s.io/client-go/util/jsonpath | 0.016 | 1.034 | 64.63x |
| k8s.io/client-go/util/keyutil | 0.037 | 1.042 | 28.16x |
| k8s.io/client-go/util/retry | 0.0201 | 1.028 | 51.14x |
| k8s.io/client-go/util/testing | 0.0253 | 1.0364 | 40.96x |
| k8s.io/client-go/util/workqueue | 0.2119 | 1.8763 | 8.85x |

Figure 9: Performance evaluation on Kubernetes *client-go* tests.

| | | Binary Size (bytes) | | |
|---|---|---|---|---|
| **Application** | **LOC** | **No Instrumentation** | **Race Instrumentation** | **Size Increase** |
| minikube | 61K | 57932352 | 71558484 | 1.24x |
| kops | 1843K | 125116028 | 154970880 | 1.24x |
| kompose | 908K | 64176684 | 70045136 | 1.09x |

Figure 10: Comparison of applications with and without race instrumentation.

**Discussion**

Through our experiments, we have measured the performance and size overhead incurred by the Go race detector. While TSAN, the base of the Go race detector, boasts exceptional performance compared to other precise dynamic data race detectors, our data shows that it is much too heavy to run in production with program slowdown of up to 64x.

One important observation is that the performance overhead incurred by the Go race detector differs greatly between Minikube and the Kubernetes client-go tests. This is largely due to a large number of goroutines being created in the client-go tests as opposed to the more lightweight Minikube commands.

**Reported Race**

In addition to performance measurements, our test resulted in the report of one unique, real data race in Minikube (5 reports). This issue is related to a failing test, *TestTeePrefix*, which was reported in July of 2019 [18]. The function *teeSSH* runs an SSH command, streaming *stdout* and stderr to logs. Within this function, two goroutines are created to log and stream *stdout* and *stderror*, respectively. Underlying, *stdout* and *stderror* share a single fixed buffer, where the race occurs. We have shared our Go race detector reports and findings with the developers [19] and are working on a patch.

# CHAPTER V

# RELATED WORK

Our work focuses on redundant instrumentation, or rather, instrumentation which cannot result in a reported race. However, there are other definitions of redundancy when it comes to race detection. ReX [20], a dynamic analysis algorithm, removes multiple races reported on the same memory accesses, which they call redundant events. Bigfoot [21], a hybrid static and dynamic analysis data race detector, combines multiple race checks on a composite object into a single check. RedCard [22] is a static analysis technique which removes redundant race checks for memory accesses within the same "release-free" span. Finally, Thread Sharing Analysis [23] provides both a static and dynamic algorithm for determining if objects escape a thread with fine-grained access, which may help instrument only concurrently accessible objects.

# CHAPTER VI

# CONCLUSION

Through the project, we have gained a strong understanding of how the Go race detector is implemented, and how it has been optimized to be widely used in practice. We studied the Go race instrumentation library and the architecture of the runtime data race detector, and then compared the project to TSAN. By comparing the LLVM TSAN instrumentation library with Go's, we found an unhandled redundant instruction pattern which may significantly reduce the amount of instrumentation required for Go race detection, improving the performance of the race detector. We conducted the first public empirical evaluation of the Go race detector, using a variety of Kubernetes tools and tests to measure the performance and instrumentation overhead of dynamic race detection. Although we show that the Go race detector is not practical for production use, there is still work to be done. A more comprehensive study on the taxonomy of benchmark programs used would significantly increase our understanding of the nature of the race detector and how we can improve it. Lastly, we discussed a real reported race by the Go race detector which we have reported to the developers. We hope that our findings will provide motivation for future research and improvements to the performance and scalability of the Go race detector, making it a more practical tool.

# REFERENCES


[1] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71.

[2] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2011. Dynamic race detection with LLVM compiler. In *Proceedings of the Second international conference on Runtime verification (RV'11)*. Springer-Verlag, Berlin, Heidelberg, 110–114.

[3] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commuications of the ACM*, 21, 7 (July 1978), 558–565.

[4] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 121–133.

[5] Go. https://github.com/golang/go.

[6] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 865–878.

[7] LLVM. Downloads. https://releases.llvm.org/download.html.

[8] ThreadSanitizerCppManual. ThreadSanitizer. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[9] Issue 19054. Go. https://github.com/golang/go/issues/19054#issuecomment-279327160.

[10] Kubernetes. https://github.com/kubernetes/kubernetes.

[11] Kubernetes. Production-Grade Container Scheduling and Management. https://github.com/kubernetes/kubernetes.

[12] Kubernetes User Case Studies. Kubernetes. https://kubernetes.io/case-studies/.

[13] Minikube. https://github.com/kubernetes/minikube.

[14] Client-go. https://github.com/kubernetes/kubernetes/tree/master/staging/src/k8s.io/client-go.

[15] ThreadSanitizer. Clang 11 Documentation. https://clang.llvm.org/docs/ThreadSanitizer.html.

[16] Kops. https://github.com/kubernetes/kops.

[17] Kompose. https://github.com/kubernetes/kompose.

[18] Issue 4767. Minikube. https://github.com/kubernetes/minikube/issues/4767.

[19] Issue 7427. Minikube. https://github.com/kubernetes/minikube/issues/7427.

[20] Jeff Huang and Arun J. Rajagopalan. 2017. What''s the Optimal Performance of Precise Dynamic Race Detection?–A Redundancy Perspective. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[21] Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. 2017. BigFoot: static check placement for dynamic race detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 141–156.

[22] Flanagan, Cormac & Freund, Stephen. (2013). RedCard: Redundant Check Elimination for Dynamic Race Detectors. 255-280. 10.1007/978-3-642-39038-8_11.

[23] Jeff Huang. 2016. Scalable thread sharing analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1097–1108.