STATIC RACE DETECTION TOOL FOR GOLANG

An Undergraduate Research Scholars Thesis

by

LORNA SANDERS

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

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Jeff Huang

May 2021

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I , Lorna Sanders, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Static Race Detection Tool For Golang

Lorna Sanders
Department of Computer Science
Texas A&M University
Research Faculty Advisor: Jeff Huang
Department of Computer Science
Texas A&M University

The built-in race detection tool works dynamically, so it finds races at runtime which can lead to false negatives. The static race detection tool on the other hand analyzes all code regardless of what happens on one particular runtime allowing it to find the races that the dynamic tool cannot.

The tool converts the code to SSA code for more easily analyzable data that does not need to be run dynamically. Then, the tool establishes an understanding of what instructions are read or write instructions, and analyzes what instructions have a happens before relation, meaning that one always happens before the other. If a happens before relation is found or the instructions are not either a write and a read instruction or a write and a write instruction, a data race between those two instructions is impossible and can be eliminated. With those baseline conditions, the tool could then be tested on real world data races to progressively make the tool more accurate.

In this real-world data race testing, the tool has been quite accurate in ensuring no false negatives occur. The goal of the tool is to have any misreported races be false positives rather than false negatives to improve on the dynamic tool. That goal has been ensured as testing has

gone on by reconfiguring the tool to meet that goal. The static tool is, as expected, slower than the dynamic tool due to the nature of static analysis compared to dynamic analysis. However, it brings a new benefit to the race detection tool world by having any errors be false positives while the dynamic tool in contrast gives false negatives. Programmers in Go can now use those two endpoints from the tools to more accurately and thoroughly find races knowing the respective benefits each tool can give separately and in conjunction with each other.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Jeff Huang, and my fellow students working on this tool Yiqing Zhao, Yahui Sun, Bozhen Liu, and Boren Zang for their guidance, support, and hard work throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

The static race detection tool described was also created, worked on, and improved upon with the work of Yiqing Zhao, Yahui Sun, Bozhen Liu, and Boren Zang.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Undergraduate research was supported by the Software & Hardware Foundation Program in the National Science Foundation with a Grant under Grant Number 1552935. Its contents are solely the responsibility of the author and do not necessarily represent the official views of the National Science Foundation

NOMENCLATURE

Concurrency	When a program is able to have 2 different things happen at once
Goroutine	A structure in the go language that allows concurrency(AKA a thread)
Data Race	2+ Goroutines race to modify or access a variable(type of Race Condition)
Race Condition	A general term for unfavorable results of concurrency
Runtime	When a program is actually used(or run)
Dynamic	When something is done at Runtime
Static	When something is done not at Runtime but could be done anytime

1. INTRODUCTION

1.1 Data Races

A data race is a very common problem in modern computer science. A data race occurs in a program that has concurrent threads which means at least two things are happening at one time. This practice allows programmers to make code that can be executed and run much faster since multiple things happen at once rather than one after the other. However, that benefit comes at a cost. When you are making multiple things happen at one time, there can be unexpected effects that would never happen in a sequential program (i.e., one that executes one after the other). When these undesirable and unexpected effects occur, they are called race conditions. These effects are very diverse and are collectively referred to as race conditions. Data races which are the focus of the current tool being discussed is a subset of that broad race condition category. As a subset of race conditions, data races are naturally also undesirable and unexpected effects from concurrency in a program. In Go, which is the language the tool analyzes, concurrency is achieved when at least two threads are happening at one time. This is much the same as other languages, but in Go, the structure to have threads is known as a Goroutine. Goroutines are the parts of the program that happen at one time. When a Goroutine is declared the program basically creates a split at that point to begin running anything in that new Goroutine at the same time as the Goroutine that originally created it and any other Goroutines that happen to have also been declared. From that first split point on we have concurrency. Data races become an issue when two Goroutines that are executing at the same time try to access the same variable and at least one of those Goroutines tries to change that variable [1]. This is a problem when left unhandled. Because the two Goroutines happen at the same time, no one can be sure which of those two accesses will be executed first. Because of this, we cannot not know whether the variable we are accessing is still the original value or if it has already been changed by another Goroutine happening at the same time. The goal of concurrency is a faster program, but not at the cost of unexpected results that could jeopardize the entire program. It does not matter how fast a program is if it does not always give the result that you were expecting or hoping for. Much like if you were web browsing and clicked on a new website. You will not care how fast the website loads if it is not the website you wanted to see. Instead, we want those instances of bad possible outcomes to be prevented and only have the parts of Goroutines happening together that won't hurt the final product. The simplest solution would be to just not have Goroutines if this could be an issue and so just have the program execute sequentially. However, this is unrealistic for how large, extensive, and time costing real world programs are already with Goroutines. Small programs that already don't take much time often don't show much of a difference in how long a program operates synchronously or concurrently at least to the user who probably won't be terribly concerned about a fraction of a second. But in real world programs that must accomplish massive amounts of code in a timely enough manner so as not to annoy the user, this can be much more difficult. Where before in those tiny programs the difference is miniscule, in a real program that time difference can be quite massive due to the size of the program. If the program is taking too long, it often does not matter how useful it is, because many users won't want to wait that long. Instead, rather than not use concurrency in problem programs, many programmers use different practices to ensure that the Goroutines happen concurrently, but the part of the Goroutine that accesses the variable is secure and must happen in a certain order with other Goroutine variable accesses for those affected programs. There are many structures, methods, and guidance's on how best to protect those vulnerable pieces, and they can be very effective when implemented properly. The problem is that often they are not implemented and even more often not implemented properly. Aside from inexperienced programmers, either for programming in general or Go specifically, who would obviously be expected to not yet have the experience to accurately protect vulnerable areas there is a less expected group of people that often experience this problem. Many people writing code today are experienced, but they work in groups of people writing massive programs many thousands of lines of code long if not more that is being continuously changed and updated by other people, who clearly did not write all of the original code, to keep the program functional. Because of this, real-world programs are extremely complex and simply impossible for any one person to accurately predict and find data races manually. Therefore, programmers will resort to using tools that can go above and beyond manually searching the code for data races, because these data race detection tools can analyze these massive complex programs automatically.

1.2 Previous Work

Prior to this static race detection tool, there was another tool in Golang that was used to detect data races. This tool is built into the Go language and works dynamically. It will henceforth be referred to as the dynamic or built-in tool. This tool works by analyzing for data races as the program is run, so it detects data races that occur at runtime [2]. The issue with this previous work is that it can only detect data races that occur at runtime.

In a programming language like Go, it is often the case that what happens at runtime is not necessarily what will happen the next time the tool is run. If-statements and their associated extensions, switches, and other structures in the language are so commonplace in Go that a real-world program is extremely likely to have them and often the result depends on conditions that are set at runtime as user input or data. Therefore, the path taken in the code that the dynamic tool is analyzing is not necessarily the path that will always occur. Because of that, it is entirely possible that another path in the code that was not chosen at the current runtime will be taken on a different run and contain data races. This would lead to a false negative. The program does contain a data race, but due to that race not presenting in this particular run of the code the dynamic tool cannot find it. This means that despite the dynamic tool being quite accurate in ensuring that races reported to the user are actually races it is still limited by the fact that those reported races are not all of the possible races.

This limitation presents an opportunity for an different tool that can ensure that all possible races are reported. This would allow programmers using the tool to know about any possible data race issues in their code. This is not possible using dynamic analysis, because dynamic analysis finds data races as the tested program runs and so can only find the data races in the current path in the code being executed at runtime. To truly improve a tool in such a way that this fault does not occur, a new tool cannot be solely based on dynamic analysis. Instead, static analysis allows a tool to analyze all possible paths rather than only the one executed at runtime. This attempt at a new perspective in data race detection tools in Go is the focus of this research and serves as a contrast to the dynamic tools limitations and benefits.

2. METHODS

2.1 Methods

This tool is meant to statically analyze Go programs for data races. That means that it needs a way to know what the individual instructions in the program are meant to do without actually running the program which would be dynamic analysis. That would defeat the purpose of using static analysis to finding data races, because the point is to use static analysis which allows for the entire program to be searched rather than the part run for this particular runtime. This knowledge can be achieved by using a helpful tool in Go that already exists called SSA. The static tool takes the program that will be analyzed and converts it into SSA code that can be easily parsed and analyzed. This is thanks to helpful structures in SSA that categorize each instruction based on what they are to give the most useful information about it. This allows the tool to know what type of instruction is being used and other helpful information without having to manually parse the original extremely complex code with only the base text to go on trying to replicate what SSA does in categorizing and cataloging the program into structures that can be analyzed.

Once the SSA code has been built, the static tool can analyze that code for data races. A data race can only happen when a read and a write operation or a write and a write operation occur concurrently and the order in which they happen is fluid. This is because as mentioned they must happen concurrently to be a data race since a data race is a race condition which is a result of concurrency. Also, although two read accesses could still access the variable, it wouldn’t really matter if they do in two different Goroutines. Because a read access would not change the variable, the second Goroutine could perform the access unaffected. A data race must include at least one write instruction in the two accesses for this reason since otherwise no ill effects would actually occur. Therefore, to search for data races the tool must ensure that a prospective race would have two accesses and at least one of those accesses is a write instruction. First the tool must know which instructions are write operations, which instructions are read operations, and which instructions are neither. This is firstly helped by the SSA code that was built. By knowing what types of instructions are being used and observing what types of instruction will result in a read, write, or neither as well as the conditions that might make one type of instruction more likely to be a certain group, the tool can be calibrated to categorize instructions based on those groups. These conditions that make instructions read or write instructions have been found by manually examining real world programs for instructions that would be read or write instructions and finding what information the SSA provides about that instruction that differentiate between when the instruction could be a write or read instruction and when it could not. For instance, a write could be a change to an int variable, but the first declaration of that variable is not a write instruction even though a value is assigned because the variable did not exist beforehand and so would not race. Therefore, the tool looks at those instructions and will not classify the first declaration as a write instruction. For an actual race to occur, both instructions must be either read or write instructions and at least one must be a write instruction. Therefore, some instructions that are neither read nor write instructions and are not necessary for other purposes can be left out of certain costly analyses like checking if the instructions are going to execute sequentially. These later checks are necessary to check if two instructions will be a data race, but for instructions that have already eliminated for not being the right accesses to lead to that, they can be skipped.

Just because two instructions are read and write instructions and at least one is a write, does not mean that those two instructions automatically qualify to possibly be racing with each other. There are more checks that must be done to ensure that the two instructions are even able to race, before checking if they actually are. The next necessary condition for a data race is concurrency. The two instructions must have been called in two different goroutines or they have no possibility of happening in different orders due to the goroutines happening concurrently, and so would not race. For this reason, the tool must keep track of what goroutine resulted in each of the instructions of note (e.g., the read or write instructions), so that it can ensure that all pairs of instructions being analyzed for a data race come from different goroutines. Additionally, for a data race to be possible, the variable being read or written to by the instructions must be the same for both instructions. That is essential for a race to occur, because otherwise the order that the two instructions happen in would not matter. Therefore, for a race to happen both instructions must affect the same value and occur in different goroutines, because that is what would lead to uncertain outcomes from a data race.

However, even after going through those checks, the tool still has to confirm if those instructions would actually race against each other. For that to occur, there must be some concurrency in the program that could lead to the two instructions happening in either order thus causing a data race as mentioned with needing two goroutines. However, there is another check that must occur to ensure that those instructions are actually running concurrently even if they are in different goroutines. Certain structures in Go will protect those sections that could cause a data race when done properly. They do this by ensuring that although the pair of instructions are in different goroutines, the actual pair of instructions have a definite order for which will happen first. Therefore, the tool needs to establish some information about what order the instructions could or definitely will happen. This is done by keeping track of certain instructions that may not be read or write instructions, but could affect the order of those read or write instructions. This is stored in a graph called the Happens Before graph that has stored a unique node for each instruction in the program of relevance. Then if one instruction is always going to happen before another like due to them being sequential instructions that occur in one single goroutine than an edge is made that shows that order. The simplest order of instructions is one after the other, but this is not always and even rarely the case. For one, certain structures in Go mean that not all instructions happen, but rather one group of instructions in a set of groups of instructions such as if-statements and its derivatives, switches, and selects is chosen among other sets of instructions to be executed based on some condition. These determinations are not possible to statically determine, so the tool cannot know which of those sets is actually executed, because it would defeat the purpose of static analysis. When the tool is making happens before relations, these structures must be handled differently. The beginning of each set of instructions must happen after the instructions before the structure begins. At the same time all instructions in the set would happen before the instructions after the structure ends, but none of the sets can be connected to each other since only one in any path would occur. Also we cannot know for sure which of the sets is chosen, so any instructions in those sets cannot be counted as definitely there or definitely not much like Schrodinger's cat. Instead, the tool must handle either case, because both are possible. The tool works by specifying in the Happens Before graph only those instructions that are guaranteed to always be in that order. Any instructions that have some uncertainty to order or sets of instructions that, like mentioned, either one or the other happen but not both are excluded. Therefore, if two instructions are not connected in this happens before relation, the tool can conclude that there is uncertainty in what order the instructions occur. In addition, if the two instructions originate from two separate goroutines, the tool can know that that uncertainty could be due to a data race. That in conjunction to the previous checks as well as checking for any structures in Go that correctly prevent a race allow the tool to determine if a data race is present.

Such structures like channel operations, waitgroups, and other safeguards are also analyzed to check not only if they are present, but if they actually prevent a race. The correctness of these safeguards is essential to check, because an ineffective safeguard will not actually prevent a data race. Oftentimes, a program will have a structure there, but may not have a properly implemented structure. Data races due to ineffective race preventing structures is fairly common, because most reasonable programmers in Go are aware of what causes races. So, these programmers often make an attempt to protect their program with race prevention structures but use those structures incorrectly in some way. These structures are not typical read or write instructions, but they are important to determining if a data race is present and must be analyzed.

All of these necessary checks that are made in the process of determining if a race is present are depicted in the flow chart below. If all of these checks are true, then the tool can report a data race. The process of these checks is depicted in Figure 2.1 shown below.

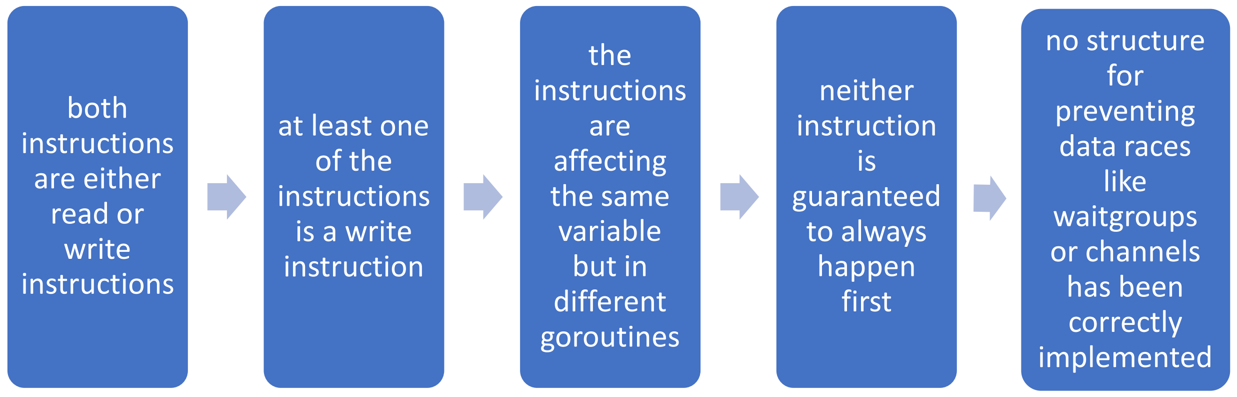

both instructions are either read or write instructions
at least one of the instructions is a write instruction
the instructions are affecting the same variable but in different goroutines
neither instruction is guaranteed to always happen first
no structure for preventing data races like waitgroups or channels has been correctly implemented

Figure 2.1: Flow Chart for Data Race Potential

Aside from practical methodology in terms of the actual process the tool uses for determining data races, there is also a methodology to testing and improving the tool as the research progressed. Initially when the tool was in the early stages, it was designed with simple test programs as a first model, and was simple as many first models are. The tool was initially tested on two sets of bench tests for real world programs that mimic real-world races in smaller programs that were called GoBench and Godel [3], [4], [5]. These were initially made by the original programmers to help debug the data races in the actual program in a smaller yet accurate portrayal that would be simpler to fix and/or to keep categories of common real world races along with their respective fixes for research and education purposes. These cases were ideal for initial testing, because they were both small in size and simple in complexity yet were based on real programming behavior and actual data races that occurred. These cases were used to analyze how different structures and their behaviors affect whether a data race occurs and develop an algorithm to detect these differences. However, these cases although helpful were not extensive enough to represent every possible scenario. A different set of test cases were manually created based on observation of different structures and experimentation to determine what could and couldn't cause a data race in situations that were not covered in the previous test sets. These edge cases allowed for shifting the testing model from not just reactive testing to races that have already been recorded to predictive testing of races that we had not yet encountered but needed to be considered. Together these test cases allowed the tool to better encompass any possible situations that could occur based on those proposed scenarios. After many smaller test cases as possible were used to implement and test initial functionality of the data race detector, the tool was ready for full real-world programs. Various popular open-source programs like Hugo, GRPC, TIDB, and others were incorporated into testing to find more scenarios that were previously unconsidered as well as transition the tool to analyzing the larger more complex programs. This was also done to allow more efficiency and speed focused improvements to be tested on programs large enough to find a significant difference in the tool's runtime.

2.2 Inherent Issues

Although this tool is meant to be an improvement on the previous dynamic tool that through using a different approach can resolve its key limitations, that by nature introduces some limitations that are inherent in this different approach. Just as the dynamic strategy has advantages and disadvantages, the static tool must also naturally have some advantages and disadvantages. Although the two tools are meant to have advantages and disadvantages that complement each other so as to provide a separate but worthwhile perspective, there are some key issues in the static approach regardless of the complementary nature of the two tools.

The dynamic tool has the advantage and disadvantage of being used as the program being tested is run. We have discussed the problems with this of other possible paths being ignored, but there is the distinct advantage of determinacy. Because the tool executes as the program runs, it can know what certain values are and not have to guestimate. A static tool does not have this advantage. Because the static tool is not run as the program executes, there is no way to know what values variables may have. The only way to account for this in analysis is to be over conservative in expecting data races. Many structures in Go that prevent data races in a program take variables as input such as waitgroups that takes in how many other Goroutines to wait to finish before continuing with the current Goroutine or channel buffers that take in how much can be in the channel before the current Goroutine is blocked to wait for some channel action in another Goroutine. For example, if a structure to prevent a data race such as a wait group or channel buffer has some value, the static tool has no way of knowing what that value is due to it not being run dynamically. As a result, the tool can only assume that that value is such that the structure cannot actually prevent a data race regardless of whether it actually will or not. This leads to false positives in the data races that are reported. Knowing whether protective structures are used properly is a key component of checking for data races, but it is hindered by this limitation of static analysis. There are possible methods to minimize this fault such as gathering context clues from the rest of the program as to whether that structure is working or not or if some other aspect prevents the possible data race which the tool does, but it cannot be completely eliminated.

The unpredictability of outcomes also has more subtle effects that the immediately obvious data race prevention structures. Yes, values are difficult or impossible to know for sure, but due to how static analysis works there is also the issue of not knowing what paths are taken. This is an advantage over the dynamic analysis for ensuring completeness of what is analyzed, but there are more unfortunate implications as well. Because we don't have a guarantee what path occurs, we also have no guarantee that what happens in any one path actually happens. For example, a simple structure like an if else that is common in many languages including Go is affected. Either one path happens or the other, but not both and not neither. Either the if path or the else path happens, but with static analysis we have no way of knowing which it was. Therefore, not only is the code within the if else section indeterminate, but often times everything after those sections is also indeterminate, because we have no idea what is actually executed before it that might have affected those later instructions. One common issue is having Lock and Unlock operations within those indeterminate if else sections, and when that is the case it can be difficult to determine what the state is after those sections are passed. Those lock operations are a common structure used to prevent data races, and they often happen in those indeterminate sections. Again, the only solution is to be overly conservative in assuming what happens in such a way that we will report races whether that may actually happen or not. Like previously stated, context clues and patterns can help to narrow down possibilities and minimize false positives, but this weakness is inevitable.

Another problem with static analysis is speed. The dynamic tool runs with the tested program and only has one possible path to worry about, and while that leaves some possible pieces out, it also ensures that the tool can run faster. To put it simply the dynamic tool does not have to analyze as much of the code as a static tool does. The larger a program gets the more stark that difference will be. Like the other problem there are certain possible fixes to try to make this better such as finding information in the analysis to allow the tool to take shortcuts because we know a data race could not happen or just in general making the tool run as fast and efficiently as possible as it analyzes the program. However, the magnitude of what and how the program is analyzed is inherently different which means the static tool will be inherently slower than the dynamic. That tradeoff for finding more possible data races is inevitable, but as long as the difference in time is close enough, that tradeoff can be considered a valuable asset. That is especially true for programs with data races in paths difficult or even impossible to have reported with the dynamic tool since speed does not matter if the race isn’t actually found.

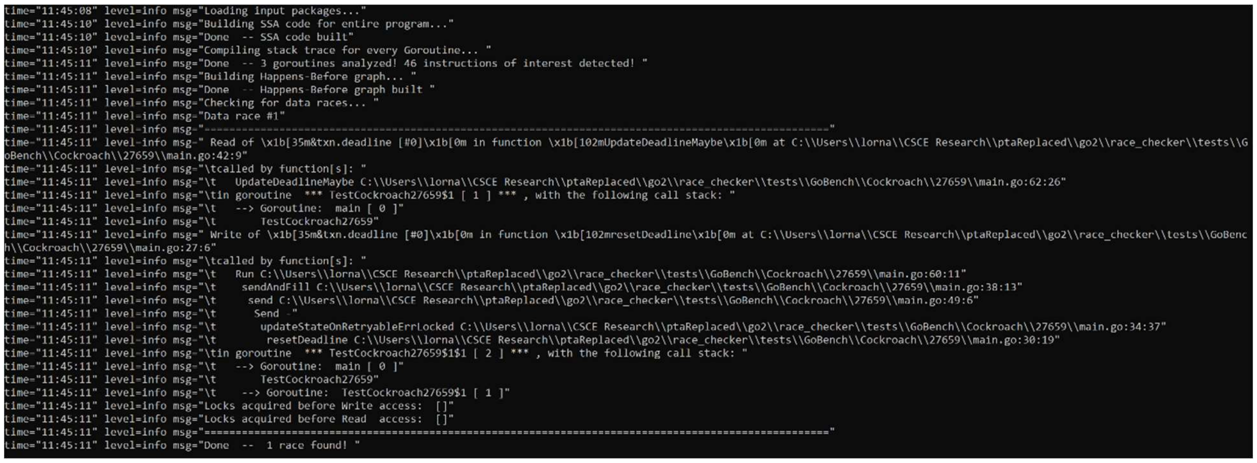
3. RESULTS

3.1 The Static Race Detection Tool

Unlike the built-in tool which needs a file for an entry point to be given, the static tool is run in the root of the project and will give a list of options for possible entry files to take that the user can choose and specify from. This is to allow the user a better ability to know what exactly is being checked and to systematically check their programs. If there is only one possible entry file, then it is run automatically.

Aside from that the tool will give you some information to tell you what step in the process it is on, as well as some information about the program itself. That information is how many Goroutines there where in the program as well as how many instructions of interest there were. Instructions of interest refers to instructions that are reading or writing to a variable which as pointed out is one of the first requirements for having a data race. Then it will either say 0 data races detected or it will tell you how many data races there were and will in sections for each data race give you information about that section's data race to help you debug your program. There will be two parts to each data race section that will represent each of the two instructions that are racing with each other. Each part will start with either 'Read of' or 'Write of' to show which type of instruction it was, then the variable name and the function it was in. Those can be tokens that are not easily decipherable in certain situations, but the last part of the line will be the file line number and character that the race occurred. On the following lines it will give the same information about the functions that called it and then tell you what Goroutine it was in.

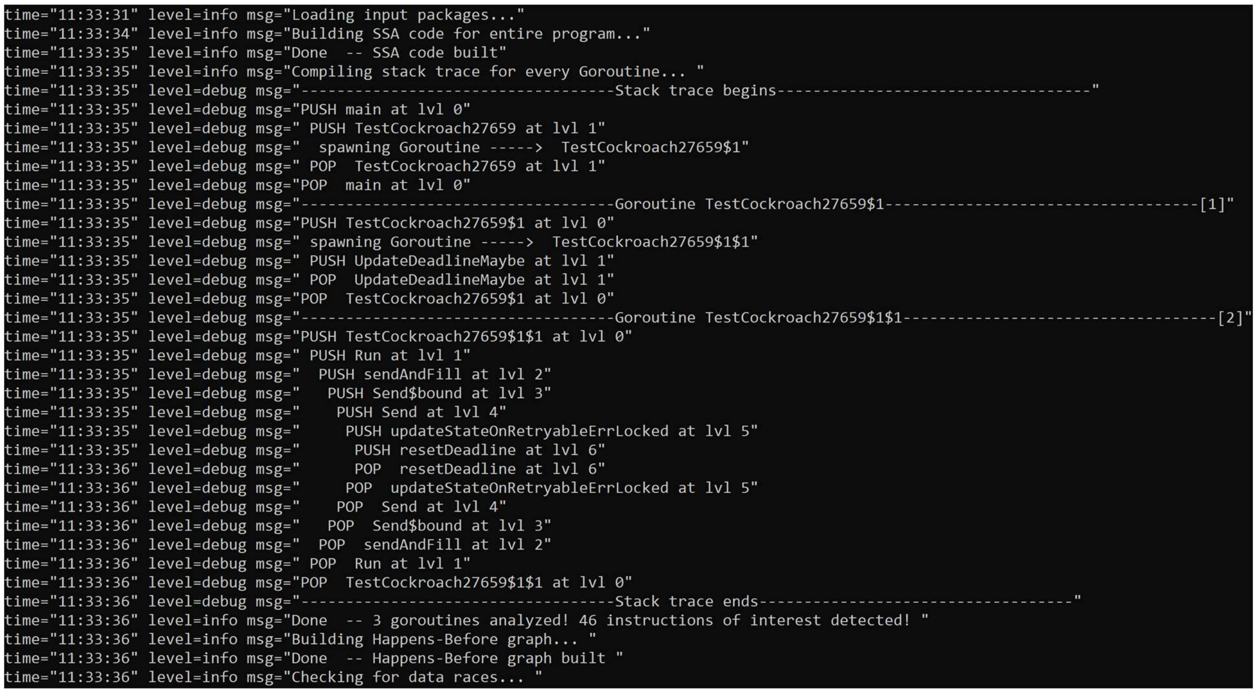
Below in Figure 3.1, there is the output of running the tool on a test case from GoBench [3], [4]. GoBench as mentioned is a set of test cases extracted from real world programs that contain real world data races [3], [4]. This is the result of test case Cockroach/27659. As shown, for the read instruction which is the first one presented, you can see it gives information about the instruction like that it occurs in the filepath of the main.go file on line forty-two at the ninth character of that line. Below that is the function and location that called it, and the Goroutine which is zero. This gives the user the ability to accurately find the instructions that cause the race and modify the code around the instructions to prevent it.



```
time="11:45:08" level=info msg="Loading input packages..."
time="11:45:10" level=info msg="Building SSA code for entire program..."
time="11:45:10" level=info msg="Done  -- SSA code built"
time="11:45:10" level=info msg="Compiling stack trace for every Goroutine... "
time="11:45:11" level=info msg="Done  -- 3 goroutines analyzed! 46 instructions of interest detected! "
time="11:45:11" level=info msg="Building Happens-Before graph... "
time="11:45:11" level=info msg="Done  -- Happens-Before graph built "
time="11:45:11" level=info msg="Checking for data races... "
time="11:45:11" level=info msg="Data race #1"
time="11:45:11" level=info msg="=============================================================================================="
time="11:45:11" level=info msg=" Read of \x1b[35m&txn.deadline [#0]\x1b[0m in function \x1b[102mUpdateDeadlineMaybe\x1b[0m at C:\\Users\\lorna\\CSCE Research\\ptaReplaced\\go2\\race_checker\\tests\\G
oBench\\Cockroach\\27659\\main.go:42:9"
time="11:45:11" level=info msg="\tcalled by function[s]: "
time="11:45:11" level=info msg="\t   UpdateDeadlineMaybe C:\\Users\\lorna\\CSCE Research\\ptaReplaced\\go2\\race_checker\\tests\\GoBench\\Cockroach\\27659\\main.go:62:26"
time="11:45:11" level=info msg="\tin goroutine  *** TestCockroach27659$1 [ 1 ] *** , with the following call stack: "
time="11:45:11" level=info msg="\t   --> Goroutine:  main [ 0 ]"
time="11:45:11" level=info msg="\t       TestCockroach27659"
time="11:45:11" level=info msg=" Write of \x1b[35m&txn.deadline [#0]\x1b[0m in function \x1b[102mresetDeadline\x1b[0m at C:\\Users\\lorna\\CSCE Research\\ptaReplaced\\go2\\race_checker\\tests\\GoBenc
h\\Cockroach\\27659\\main.go:27:6"
time="11:45:11" level=info msg="\tcalled by function[s]: "
time="11:45:11" level=info msg="\t   Run C:\\Users\\lorna\\CSCE Research\\ptaReplaced\\go2\\race_checker\\tests\\GoBench\\Cockroach\\27659\\main.go:60:11"
time="11:45:11" level=info msg="\t    sendAndFill C:\\Users\\lorna\\CSCE Research\\ptaReplaced\\go2\\race_checker\\tests\\GoBench\\Cockroach\\27659\\main.go:38:13"
time="11:45:11" level=info msg="\t     send C:\\Users\\lorna\\CSCE Research\\ptaReplaced\\go2\\race_checker\\tests\\GoBench\\Cockroach\\27659\\main.go:49:6"
time="11:45:11" level=info msg="\t      Send -"
time="11:45:11" level=info msg="\t       updateStateOnRetryableErrLocked C:\\Users\\lorna\\CSCE Research\\ptaReplaced\\go2\\race_checker\\tests\\GoBench\\Cockroach\\27659\\main.go:34:37"
time="11:45:11" level=info msg="\t        resetDeadline C:\\Users\\lorna\\CSCE Research\\ptaReplaced\\go2\\race_checker\\tests\\GoBench\\Cockroach\\27659\\main.go:30:19"
time="11:45:11" level=info msg="\tin goroutine  *** TestCockroach27659$1$1 [ 2 ] *** , with the following call stack: "
time="11:45:11" level=info msg="\t   --> Goroutine:  main [ 0 ]"
time="11:45:11" level=info msg="\t       TestCockroach27659"
time="11:45:11" level=info msg="\t    --> Goroutine:  TestCockroach27659$1 [ 1 ]"
time="11:45:11" level=info msg="Locks acquired before Write access:  []"
time="11:45:11" level=info msg="Locks acquired before Read  access:  []"
time="11:45:11" level=info msg="=============================================================================================="
time="11:45:11" level=info msg="Done  --  1 race found! "
```

Figure 3.1: Result of running the tool on a program

The user can also run the tool with a -debug flag that will further help the user debug their program by showing them a stack trace of the functions in the program to see what exactly led to the data race. Those lines that begin with the ‘level = info’ are always shown as seen in the figure above, but those with ‘level = debug’ are only added when the debug flag is used as shown in the figure below. This allows the user to easily distinguish between the two types of information. Figure 3.2 depicted below shows the beginning of the results, where the debug information is shown, with the end omitted from the figure due to being identical to the output shown in Figure 3.1.



```
time="11:33:31" level=info msg="Loading input packages..."
time="11:33:34" level=info msg="Building SSA code for entire program..."
time="11:33:35" level=info msg="Done  -- SSA code built"
time="11:33:35" level=info msg="Compiling stack trace for every Goroutine... "
time="11:33:35" level=debug msg="-----------------------------------Stack trace begins-----------------------------------"
time="11:33:35" level=debug msg="PUSH main at lvl 0"
time="11:33:35" level=debug msg=" PUSH TestCockroach27659 at lvl 1"
time="11:33:35" level=debug msg="  spawning Goroutine ----->  TestCockroach27659$1"
time="11:33:35" level=debug msg=" POP  TestCockroach27659 at lvl 1"
time="11:33:35" level=debug msg="POP  main at lvl 0"
time="11:33:35" level=debug msg="-----------------------------------Goroutine TestCockroach27659$1-----------------------------------[1]"
time="11:33:35" level=debug msg="PUSH TestCockroach27659$1 at lvl 0"
time="11:33:35" level=debug msg=" spawning Goroutine ----->  TestCockroach27659$1$1"
time="11:33:35" level=debug msg=" PUSH UpdateDeadlineMaybe at lvl 1"
time="11:33:35" level=debug msg=" POP  UpdateDeadlineMaybe at lvl 1"
time="11:33:35" level=debug msg="POP  TestCockroach27659$1 at lvl 0"
time="11:33:35" level=debug msg="-----------------------------------Goroutine TestCockroach27659$1$1-----------------------------------[2]"
time="11:33:35" level=debug msg="PUSH TestCockroach27659$1$1 at lvl 0"
time="11:33:35" level=debug msg=" PUSH Run at lvl 1"
time="11:33:35" level=debug msg="  PUSH sendAndFill at lvl 2"
time="11:33:35" level=debug msg="   PUSH Send$bound at lvl 3"
time="11:33:35" level=debug msg="    PUSH Send at lvl 4"
time="11:33:35" level=debug msg="     PUSH updateStateOnRetryableErrLocked at lvl 5"
time="11:33:35" level=debug msg="      PUSH resetDeadline at lvl 6"
time="11:33:36" level=debug msg="      POP  resetDeadline at lvl 6"
time="11:33:36" level=debug msg="     POP  updateStateOnRetryableErrLocked at lvl 5"
time="11:33:36" level=debug msg="    POP  Send at lvl 4"
time="11:33:36" level=debug msg="   POP  Send$bound at lvl 3"
time="11:33:36" level=debug msg="  POP  sendAndFill at lvl 2"
time="11:33:36" level=debug msg=" POP  Run at lvl 1"
time="11:33:36" level=debug msg="POP  TestCockroach27659$1$1 at lvl 0"
time="11:33:36" level=debug msg="-----------------------------------Stack trace ends-----------------------------------"
time="11:33:36" level=info msg="Done  -- 3 goroutines analyzed! 46 instructions of interest detected! "
time="11:33:36" level=info msg="Building Happens-Before graph... "
time="11:33:36" level=info msg="Done  -- Happens-Before graph built "
time="11:33:36" level=info msg="Checking for data races... "
```

Figure 3.2: Result of running the tool on a program with the debug flag

4. CONCLUSION

4.1 Future Work

Although the tool can already detect data races on real world programs using static analysis, as was the goal, there do remain some aspects of the tool that could be improved as is true with all such things.

For one, the tool could be made faster. The static tool uses static analysis meaning it must analyze all possible branches in a program being checked for data races unlike the dynamic tool which can only check the current branch. In contrast, the dynamic tool works by running the program and dynamically detecting data races. These factors ensure that the static tool is likely to remain somewhat slower than the dynamic, built-in tool due to the nature of how the two tools work. However, there is still room for faster analysis within this limitation. Ways of attempting this feat in the future could include simple code changes to order of analysis that would ensure that fewer instructions that take up time are executed, using greater amounts of pointer analysis with fewer unnecessary frills that would take up time, finding shortcuts to not have to execute certain commands or conduct analysis when conditions ensure the tool knows the outcome, and many more. The tool is a large program, and as such has a fair bit of room for increasing speed. For the initial implementation of the tool, accuracy was placed over speed, but now that accuracy is being achieved, speed has been reprioritized.

Aside from time for the tool to execute, improvements could be made on what can be analyzed. The built-in data race detection tool runs dynamically while the program itself runs. This means that the built-in tool can only check programs that can be run themselves. It cannot check libraries or other Go programs that do not run. This is a limitation that cannot be solved with dynamic analysis. Currently the static tool is designed to match the functionality of the built-in tool in terms of needing a main package to focus on in order to speed the process of finding the program to analyze within the project. This is part of the tool's process of finding the go files in a project. However, in static analysis it could be possible to solve the previous limitation of the built in tool and be able to analyze programs the built-in tool could not like the mentioned libraries that do not have a main package. To do this the tool would have to have further methods that would look for possible Go programs that are not the main package or called by it. This may cause a kind of clutter for users that do not want or need programs analyzed that aren't part of the main package, but it could also give users that do need this feature options that the dynamic tool could not.

Aside from those improvements, there is a rather obvious improvement that is doubtful to ever truly be complete. In making a tool such as this, to be more accurate means to better predict what could happen in the programs analyzed. This is monumentally difficult, because of the sheer number, diversity, and size of the programs being analyzed. Predicting edge cases that could occur in those programs that could affect how we report races is especially difficult due to the magnitude of possible situations. Accounting for all of those situations would be difficult to accomplish, and if done inefficiently could make the tool unrealistically slow. Speed and accuracy are a tradeoff ultimately, but that does not mean accuracy is not worth pursuing further. It only means that such pursuit's rewards must be weighed against the requisite time cost. All of this will of course be made easier as more and more programs are used with the tool, because that will introduce many edge cases as well as allow for a broader array of examples to analyze and categorize for making more efficient analysis.

As the tool is meant to analyze and work with real world programs that are continuously growing and evolving, the tool must naturally grow and evolve to accommodate the programs being analyzed. However, there are more ways for the tool to adapt than just to the programs. A real-world program being used by real people as the tool is meant to be will encounter many new situations and revelations. These will allow for growth not only from the programs being analyzed but also for growth in the scope within which the tool can be used as well as how effective the tool is in doing its job.

REFERENCES

[1] J. Regehr, “Race Condition vs. Data Race,” *Embedded in Academia*, 13-Mar-2011. [Online]. Available: https://blog.regehr.org/archives/490. [Accessed: 02-Feb-2021].

[2] D. Vyukov and A. Gerrand, “Introducing the Go Race Detector,” *The Go Blog*, 26-Jun-2013. [Online]. Available: https://blog.golang.org/race-detector. [Accessed: 25-Jan-2021].

[3] T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue, *GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs*. [Online]. Available: http://lujie.ac.cn/files/papers/GoBench.pdf. [Accessed: 11-Apr-2021].

[4] Timmyyuan, “GoBench,” *GitHub*, 10-Dec-2020. [Online]. Available: https://github.com/timmyyuan/gobench. [Accessed: 11-Apr-2021].

[5] JujuYuki, “Godel2,” *GitHub*, 20-Apr-2020. [Online]. Available: https://github.com/JujuYuki/godel2-benchmark. [Accessed: 11-Apr-2021].