DYNAMATIC: A RACE DETECTION TOOL COMBINING STATIC AND DYNAMIC ANALYSIS

An Undergraduate Research Scholars Thesis

by

MATTHEW DAVIS[1], DYLAN THERIOT[2]

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: Dr. Jeff Huang

May 2021

Majors: Computer Science[1]
Computer Science[2]

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.

We, Matthew Davis, Dylan Theriot, 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

Page

ABSTRACT 1

DEDICATION 3

ACKNOWLEDGEMENTS 4

NOMENCLATURE 5

1. INTRODUCTION 6

    1.1 Background Information 6
    1.2 Current Research 10
    1.3 Dynamatic Research 11

2. METHODS 13

    2.1 The Dynamatic Tool 13
    2.2 Modifying HPCRace and Thread Sanitizer 15
    2.3 Dynamatically Analyzing The Results 16
    2.4 Dynamatic Analysis Example 17

3. RESULTS 20

    3.1 Data Race Benchmark Testing 20

4. CONCLUSION 22

    4.1 Final Remarks 22
    4.2 Future Work 23

REFERENCES 24

ABSTRACT

Dynamatic: An OpenMP Race Detection Tool Combining Static and Dynamic Analysis

Matthew Davis[1], Dylan Theriot[2]
Department of Engineering: Computer Science[1]
Department of Engineering: Computer Science[2]
Texas A&M University

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

Data races are a type of bug in concurrent programming which can result in unexpected program behavior. When multiple threads modify the same memory location in parallel, a data race occurs. Detecting these races is a difficult problem that becomes unrealistic for a programmer to perform at a large scale. Thus, automated data race detection has a large importance on fixing and verifying the correctness of parallel program behavior. There are two main types of data race detection: static and dynamic. Each analysis has its own set of limitations, and tools utilizing one type of analysis suffer from these drawbacks.

We present Dynamatic, a hybrid race analysis tool which builds off of HPCRace for static analysis and Google's Thread Sanitizer for dynamic analysis. Dynamatic performs analysis on C++ and Fortran code that is compiled down to LLVM's IR. In particular, Dynamatic analyzes programs utilizing the OpenMP API for parallelization. The tool is able to leverage the best elements of both types of analysis - the level of coverage that static tools provide, and the low false positive rates of dynamic tools. Thus, the tool is able to mitigate these drawbacks

through its hybrid approach and analysis optimizations. Dynamatic efficiently and accurately detects data races in OpenMP programs and is competitive with tools such as Archer and ROMP on benchmarks.

DEDICATION

To our friends, families, Dr. Huang, O2 Lab, and peers who supported us throughout the research process.

ACKNOWLEDGEMENTS

Contributors

We would like to thank our faculty advisor, Dr. Jeff Huang, and our fellow lab researcher, Fatmaelzahraa Alaa Eldien Anwar Ibrahim Elsheimy, for their guidance and support throughout the course of this research.

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

The source code of HPCRace used for Dynamatic was provided by Dr. Jeff Huang. The TSAN modifications used for Dynamatic were provided by Fatmaelzahraa. The analyses depicted in results for Archer and TSAN were conducted in part by the Data Race Bench team at Lawrence Livermore National Lab.

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

Funding Sources

This work was also made possible in part by the National Science Foundation under Grant Number 1552935. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the National Science Foundation.

NOMENCLATURE

HPC	High Performance Computing
TSAN	Google's Thread Sanitizer
LLVM	Low Level Virtual Machine
IR	Intermediate Representation
OpenMP	Open Source Multi-Processing API

1. INTRODUCTION

1.1 Background Information

1.1.1 Software Bugs

Computer science is a relatively new field spanning over the past century. This field has seen rapid growth and development in that time, and while software programs have changed drastically, one thing remains constant - the existence of software bugs. Software bugs are flaws or errors in software that cause unintended side effects. These side effects can result in incorrect results, failures to compile, and more. With the ever increasing importance of software, it has also become a necessity to reduce the amount of software bugs present in an application. Doing so could save companies and organizations time and money. Therefore a field within software has emerged aimed at creating software analysis and debugging tools. These types of tools are aimed at helping developers find and eliminate the bugs that may exist in their code.

1.1.2 Concurrency

When a program has code that executes in parallel, it is called a concurrent program. Concurrent programs are notoriously difficult to debug. If not designed carefully, the scheduling of the program's parallel execution can lead to non-deterministic outcomes. In some concurrent programs, there could be an extremely unlikely scenario which results in a crash or bug. It is possible that a software engineer tests it locally and the program performs as expected, but when shipped out to millions of users, a user encounters a fatal crash. This problem is a very difficult one to solve, as detecting all potential interleavings of code and the corresponding outcomes becomes exceedingly more time consuming and complicated for every new parallel section or line of code.

1.1.3 Data Races

A data race is a specific kind of concurrent program bug that can cause many unwanted side effects. To understand a data race, let us first define the terms events and unordered. "Events" indicates a memory access event, eg. a read or a write on a variable. "Unordered" means that in a concurrent program, no happens-before relation can be drawn between a set of events. Happens-before is a partial-order relation on code statements. Happens-before relations exist when a statement must happen before another statement. This can tell us the order of statements in a program. If statement A happens-before statement B, then statement B cannot happen before statement A. This seems like an obvious conclusion, but it can be much more complicated in practice with concurrent programs. A lack of happens-before relation means the statements can occur in any order. An example of this is that on two parallel execution threads, one thread could encounter the shared variable data before or after the other thread writes to it. This can lead to the program expecting a variable to have one value, but it has not been assigned that yet, causing a different outcome. Thus, a data race bug occurs when there exists a pair of two unordered events on shared data, at least one of which is a write.

Consider the following code segment, *Figure 1.1.3*, where a data race occurs on x. The check of x's value and the write to x (x=1) could happen in any order. This could cause the program to either quit or print x=1, with the user/programmer having no control.

Data Race Example		
1: x = 0		
Thread 1	Thread 2	
2: x = 1	if (x == 0)	quit()
3:	else	print("x = 1")

Figure 1.1.3: Simple data race example on the variable x

1.1.4 Program Analysis

1.1.4.1 General Information

Program analysis is the process of programmatically analyzing software programs and code to determine different aspects of the program. These aspects can vary depending on the user's desires. From detecting scalability issues, to data races, to syntactical bugs and more, program analysis is a helpful tool utilized by developers to ensure maximum efficiency of their programs. It is especially helpful on larger products with huge code bases. By programmatically analyzing the large code bases, bugs and other data can be determined a lot quicker and more efficiently compared to manual detection. Some various types of program analysis include testing, control flow analysis, data flow analysis, static analysis, dynamic analysis, and more. This paper will specifically focus on static analysis and dynamic analysis.

1.1.4.2 Static Analysis

Static analysis is the analysis of programs by reading and analyzing the code without actually running the program. Static analysis has access to all of the code, not just the segments encountered during execution, and is therefore theoretically able to detect more bugs than dynamic analysis. The drawback of static analysis is that it is often very difficult to implement precisely and without becoming too slow to be useful. One key component and bottleneck of static analysis is pointer analysis, which analyzes memory addresses and pointers to determine what variables point to where. Pointer analysis, and by extension static analysis, are a key part of detecting data races in programs. Flow-insensitive pointer analysis is where memory locations are analyzed with reference to all pointers that refer to the memory location, regardless of that pointer's life cycle in the program's execution. This tends to be faster than flow-sensitive

analysis due to the less concise analysis. With these tradeoffs of concise analysis for speed however, static analysis can result in imperfect results [1].

1.1.4.3 Dynamic Analysis

Where static analysis inspects the actual code, dynamic analysis targets analyzing the execution of the program. It gathers data through watching the memory accesses and function calls of the running program. On-The-Fly analysis is analysis done while the program is running, and post-mortem analysis is after the program has finished. On-The-Fly analysis is typically performed through having another program run alongside the program, keeping track of memory events and interpreting them as it runs. Post-mortem analysis is typically a program which analyses special data that has been collected from the program's execution, called a trace. The amount of information collected during the on-the-fly and post-mortem analyses can be configured, although without direct access to the code, the amount of understanding of the program is limited. Dynamic analysis therefore traditionally suffers from failures to detect data races.

1.1.5 OpenMP

With the rise of using computers to run models, simulations, and perform complicated calculations, sequential programs were not fast enough. A concurrency API was developed to help mathematicians and scientists easily prepare highly-parallel programs to speed up their models and systems. This API, called OpenMP, is one of the most popular ways that large-scale C++ & Fortran programs have sped up their runtime in recent years.

1.1.6 High Performance Computing

High performance computing (HPC) is the ability to perform large, complex calculations at extremely fast speeds. This is typically achieved through powerful hardware and software,

such as utilizing tons of cores and threads on a supercomputer for parallel processing. Essentially, high performance computing is utilized to run code at faster speeds for faster computations. This is different from running code on a typical desktop or laptop, as standard PCs are often limited in computational power and space. But even with HPC, complex models on HPC machines can still take weeks to months and more to finish calculations. That is why the emergence of parallel computing in HPC has become a big topic, as this can further enhance computational speed. HPC is most often used within businesses, academia, and the scientific community.

1.2 Current Research

1.2.1 Existing Tools

Many OpenMP-focused tools exist which only utilize either static or dynamic analysis. Thread Sanitizer is a popular dynamic analysis tool which inserts statements analyzing it during runtime. HPCRace is a static analysis tool which analyzes a partially compiled state of the source code to find races without running the program.

Archer is an analysis tool most similar to our Dynamatic tool [5]. They use static and dynamic analysis, incorporating a custom TSAN build very similar to our approach. This usage of dynamic analysis through TSAN acted as one of our inspirations for our research. The main difference between our tool and the Archer tool's technique is in the static analysis. Their static analysis is just to identify potential race areas and they solely detect races dynamically, whereas our tool identifies races statically, and verifies them dynamically. This enables us to leverage the greater detection coverage of static analysis, and account for false positives detected statically by testing if they have also been dynamically detected. If a race is dynamically detected and statically detected, we deem it a "dynamatic" race which implies a higher confidence in the

race's validity. If the race is detected statically but is unable to be verified dynamically, then it tells the programmer it may be a false positive race. If the race is detected dynamically but is not detected statically, then it most likely represents a flaw or shortcoming in the static analysis.

1.2.2 Existing Benchmarks

There are many types of data races to detect, and with the OpenMP concurrency library there are a lot of library functions that could cause interesting scenarios which could be undetected by analysis tools. To measure the effectiveness of Dynamatic, we use currently available benchmark tests to evaluate our tool versus other analysis tools. In particular, we utilize a collection of benchmarks called Data Race Bench, which tests various race scenarios using the OpenMP concurrency API in C/C++ & Fortran [2,3].

1.3 Dynamatic Research

1.3.1 Goals

This research aims to combine both the power of static analysis and dynamic analysis together; hence the name Dynamatic. The Dynamatic tool will have a special emphasis on Fortran code, and the tool will provide users with data race detection. The tool utilizes LLVM intermediate representation files, generated from the Fortran compiler Flang, to execute static and dynamic analysis on. The tool is built on top of HPCRACE, a static analysis tool, and TSAN, a dynamic analysis tool.

1.3.2 LLVM

LLVM, the low level virtual machine, is a compiler infrastructure that allows a program to essentially be compiled without being processor specific. Compilers are able to generate an intermediate representation file, known as IR, that is standardized to LLVM. By analyzing IR files, the analysis can be more standardized even for different languages like C++ and Fortran.

Special compilers have been created in order to generate LLVM IR files. Clang is one such compiler for C++ and Flang is one for Fortran code.

1.3.3 HPCRace

HPCRace is a static analysis tool on LLVM compiled programs. HPCRace reads and interprets the IR file directly to find the memory access events and threads. By combining pointer analysis to determine if the memory accessed is the same with thread analysis to see if it can happen in parallel, HPCRace is able to detect data races without running the program.

1.3.4 TSAN

Google's Thread Sanitizer (TSAN) is a dynamic analysis tool on LLVM compiled programs [4]. TSAN works by inserting calls inside the IR of the program's code, which when compiled and executed, enables TSAN to keep track of parallel sections & threads, memory access events, and then analyze which of these could happen in any order (unordered). This therefore can detect data races within code.

2. METHODS

To combine the analysis from HPCRace and TSAN into the Dynamatic tool, modifications were made to the two projects. These modifications allow for the static and dynamic analysis tools to interface with each other to analyze, detect, and verify data races in source code. In this section we will go into detail about the modifications made to the tools and how they come together to form Dynamatic.

2.1 The Dynamatic Tool

At a high level, we have modified HPCRace to create a special file called the instrumentation include list based off of the races detected. We have then altered TSAN to take this include list and only instrument the sections of code around the statically detected race. This allows us to do two things: one, we are able to verify the race, and two, we can optimize and speed up the dynamic instrumentation. By comparing these results we can then relate the races from static analysis to those of dynamic analysis, and can confirm races “dynamatically”. We will go more in depth into these changes later.

To create the Dynamatic executable, we modified the compilation of the HPCRace executable. Instead of the executable entry point being the main function of HPCRace, a new main function for Dynamatic was created. This allows us more flexibility over the inputs and outputs of the tool, as well as keeping a separation of interests within our files. We wanted HPCRace code modifications to only deal with optimizations and static analysis information, not dynamatic analysis. Additionally, in order to compile Dynamatic, LLVM and the modified TSAN need to be compiled first.

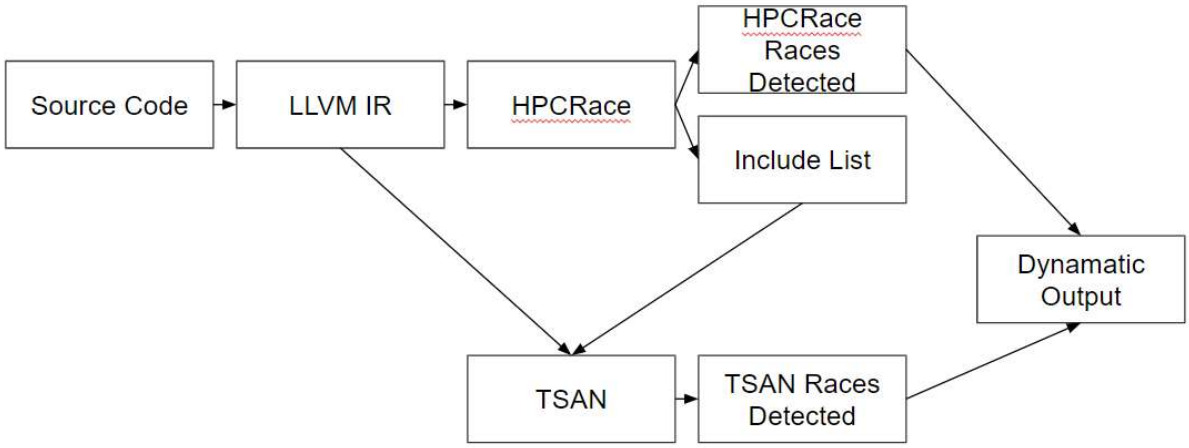

Figure 2.1: Flow diagram displaying the process of Dynamatic

Thus, the overall format of the Dynamatic executable runtime is as follows. First, the arguments are collected to modify what functionality of Dynamatic is run. Then, the source code is compiled into the low level intermediate stage called LLVM IR. The Dynamatic code passes the IR code to HPCRace's main function, and HPCRace detects races and generates the include list for TSAN. Afterwards, TSAN then runs on this list and the IR to detect races separately, then we match up the results and report this to the user. A graphical representation of Dynamatic's flow can be found in *Figure 2.1*.

We have also included a set of flags to help improve the user experience of Dynamatic. One such flag is the "savetemps" flag. Throughout the Dynamatic process, several additional files, that we call artifacts, are created. LLVM creates LL files, bitcode files, and more. HPCRace creates races.json and includelist.txt. TSAN creates tmp.log, error.log, and an instrumented executable. After Dynamatic has completed analysis, it will automatically delete these artifacts unless the "savetemps" flag is specified by the user. Furthermore, we allow the option to run only static analysis or only dynamic analysis. If only dynamic analysis is specified, then the entire program/source code provided is instrumented as opposed to just the functions

specified by an include list. And lastly, we allow the user to have the output of the dynamatic analysis be presented to them in the console by specifying the “console” flag. Otherwise, the output is written to the dynamatic_results.txt file.

2.2 Modifying HPCRace and Thread Sanitizer

HPCRace provides the detected races output. However, in order to dynamically verify these, additional information must be gathered and provided to TSAN. TSAN runs by instrumenting (inserting function calls) into the LLVM IR bitcode functions. Our Race Data from HPCRace contains the source code lines and the variable/memory location that the race occurs on. One thought is that we instrument the IR in the source line locations provided by HPCRace. With testing however, we concluded that there is a lot of inaccuracy in these provided source lines, particularly with Fortran. We attempted to fix this, but it could not be done without substantial changes to the compilers of LLVM. Another idea is to instrument everything, and match up the races based on these locations and information about the variable involved. This solution is too slow for large programs however, as Thread Sanitizer incurs a 10x slowdown on average. We approached the problem from what TSAN currently handles. TSAN is able to accept a blacklist of Source or IR function names and files which prevent it from instrumenting that code. If instead, we provide a list of the functions that should be instrumented and avoid everything not in the list, we can instrument the potentially race containing code without instrumenting everything in the code. In order to do this, we need to alter HPCRace to provide us with either the IR function or Source Code function names that encapsulate the race area. Our first implementation used the Source Code function names. This worked well, however would often cause much more of the code than is necessary to be added to the list. In a function that is

very lengthy, or has internal function calls this could be very problematic. Consider the example in *Figure 2.2*.

Source Code Function Inefficiency Example

```
1: Main {
2:         EntireProgram()
3:         –PotentialRaceArea–
4: }
```

Figure 2.2: Instrumenting with Source Code function names

The list of functions to instrument would be only Main, but all functions called within the functions instrumented are also instrumented, so this would instrument the entire program just to analyze the race area at the end. This is a problem that could result in a large and unnecessary slowdown to the analysis. We swapped over to using the IR functions instead, and found that LLVM compilers will put the OpenMP API sections in their own IR functions. This is very useful, as by using these functions, we can isolate just the parallel section that the race is inside. We made modifications to detect and send these IR functions to a file called an Include List which will be read in by TSAN.

2.3 Dynamatically Analyzing The Results

Now that results have been collected from both tools, they are combined together and the detected races are reported to the user. In doing so, a few challenges arose.

One such challenge is how can we be sure that the detected race from the static analysis through HPCRace matches up with the race detected dynamically through TSAN? If there are multiple races in a function we may run into an issue of which one was detected. The information we have available is the line number and the IR function that it occurred in. The line numbers are known to be inaccurate in the static analysis part however, so we cannot associate them directly. Instead we use an approximation that the ordering of the line numbers should be

the same. The earliest line number in HPCRace's race report is assumed to be synced up with the earliest line number in TSAN's report, and so on. This method is probably sufficient, as in the tests with the line numbers, they were all incorrectly offset by the same amount from the real line.

Another problem is that TSAN tends to report duplicate races, breaking our previous algorithm. If TSAN reports the same race twice, we end up associating the same race with multiple static races. We eliminate this problem by checking if the function and line number are the same and removing the duplicates. This system assumes that there are not multiple races per source line, which is likely.

Now that we have found which races in both reports correspond to each other, we can generate our report. We provide the user with a report with the name of the IR functions associated with the race (usually separated into the outlined and debug function names). We provide the race locations, with their file, line, and column numbers. This information is provided for every race, although if it is not detected "Dynamatically" is separated into a static-only or dynamic-only section. This organization was chosen to be very simple but user friendly. We chose this design because the race report from TSAN is relatively difficult to read and parse, and is different from HPCRace. By making a simple and standardized report format it is very clear to the programmer what the detected race is and what tool(s) detected it.

2.4 Dynamatic Analysis Example

Here is a simple example of our tool in action:

First we take the Source Code and compile into LLVM IR. This can be seen in *Figure 2.4.1* and *Figure 2.4.2*.

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, const char *argv[]) {
    int x = 0;
    #pragma omp parallel
    {
        x++;
    }
}
```

Figure 2.4.1: Example source code of a C program utilizing OpenMP and containing a data race.

```
; Function Attrs: noinline norecurse nounwind optnone uwtable
define internal void @.omp_outlined.(i32* noalias, i32* noalias, i32* dereferenceable(4)) #2 !dbg !41 {
  %4 = alloca i32*, align 8
  %5 = alloca i32*, align 8
  %6 = alloca i32*, align 8
  store i32* %0, i32** %4, align 8
  call void @llvm.dbg.declare(metadata i32** %4, metadata !42, metadata !DIExpression()), !dbg !43
  store i32* %1, i32** %5, align 8
  call void @llvm.dbg.declare(metadata i32** %5, metadata !44, metadata !DIExpression()), !dbg !43
  store i32* %2, i32** %6, align 8
  call void @llvm.dbg.declare(metadata i32** %6, metadata !45, metadata !DIExpression()), !dbg !43
  %7 = load i32*, i32** %6, align 8, !dbg !46
  %8 = load i32*, i32** %4, align 8, !dbg !46
  %9 = load i32*, i32** %5, align 8, !dbg !46
  %10 = load i32*, i32** %6, align 8, !dbg !46
  call void @.omp_outlined._debug__(i32* %8, i32* %9, i32* %10) #4, !dbg !46
  ret void, !dbg !46
}
```

Figure 2.4.2: LLVM IR for the example source code.

Once we have obtained the IR, we send it to HPCRace, which creates the Include List file and the HPCRace Report, respectively shown in *Figure 2.4.3* and *Figure 2.4.4*.

```
fun:.omp_outlined.
fun:.omp_outlined._debug__
```

Figure 2.4.3: Include list generated by HPCRace for the example source code.

```
"races":[{"access1":{"col":9,"dir":"","filename":"/root/../src/hpcrace/TestCases/PI/simplerace.c","lin
e":8,"snippet":" 6|    #pragma omp parallel\n 7|    {\n>8|        x++;\n 9|    }\n
10|}\n","sourceLine":" 8|
x++;\n","stacktrace":[]},"access2":{"col":10,"dir":"","filename":"/root/../src/hpcrace/TestCases/PI/si
mplerace.c","line":8,"snippet":" 6|    #pragma omp parallel\n 7|    {\n>8|        x++;\n 9|    }\n
10|}\n","sourceLine":" 8|
x++;\n","stacktrace":[]},"isOmpRace":true,"ompInfo":{"callingCtx":["main"],"snippet":">6|    #pragma
omp parallel\n 7|    {\n 8|        x++;\n 9|    }\n
10|}\n"},"sharedObj":{"dir":"","filename":"/root/../src/hpcrace/TestCases/PI/simplerace.c","line":5,"s
ourceLine":" 5|    int x = 0;\n"}}]
```


Figure 2.4.4: HPCRace's original static analysis output to races.json.

Next, we run the TSAN's instrumentation process using the include list, and analyze dynamically, producing the TSAN report. The TSAN report can be seen in *Figure 2.4.5*.

```
==================
WARNING: ThreadSanitizer: data race (pid=142)
  Write of size 4 at 0x7ffccb572bcc by thread T9:
    #0 .omp_outlined._debug__ /src/build/bin/simplerace.c:8:10 (tsanexecutable+0x4b24c2)
    #1 .omp_outlined. /src/build/bin/simplerace.c:7:5 (tsanexecutable+0x4b2528)
    #2 __kmp_invoke_microtask <null> (libomp.so+0xa92d2)

  Previous write of size 4 at 0x7ffccb572bcc by thread T4:
    #0 .omp_outlined._debug__ /src/build/bin/simplerace.c:8:10 (tsanexecutable+0x4b24c2)
    #1 .omp_outlined. /src/build/bin/simplerace.c:7:5 (tsanexecutable+0x4b2528)
    #2 __kmp_invoke_microtask <null> (libomp.so+0xa92d2)
SUMMARY: ThreadSanitizer: data race /src/build/bin/simplerace.c:8:10 in .omp_outlined._debug__
==================
ThreadSanitizer: reported 1 warnings
```

Figure 2.4.5: Thread Sanitizer's original dynamic analysis output to error.log.

Finally, we combine these reports and generate our Dynamatic report. An example report is shown in *Figure 2.4.6*.

```
1   Dynamatic Race Report:
2
3   --Dynamatically Found Races--
4   Race #1:
5   outlined function: .omp_outlined.
6   debug function: .omp_outlined._debug__
7       event #1 location:
8           file: simplerace.c
9           line: 8
10          col: 9
11      event #2 location:
12          file: simplerace.c
13          line: 8
14          col: 10
15
16  --Static-Only Races--
17
18  --Dynamic-Only Races--
19
```

Figure 2.4.6: Dynamatic report to dynamatic_results.txt combining information from both HPCRace and TSAN.

3. RESULTS

3.1 Data Race Benchmark Testing

We conducted tests on a publicly available benchmark set called Data Race Bench, created by a team at Lawrence Livermore National Laboratory. This benchmark set contains 177 tests, consisting of tests containing races and not containing races. Using our tool on these benchmarks, we can compare our tool to the other tools. *Table 3.1* documents our results on the benchmarks. A true positive is when the tool correctly identifies a data race, whereas a true negative is when the tool correctly identifies that there is no datarace. Totals may not be the same due to compiler differences and OpenMP pragma compatibility.

Table 3.1: Table showing the results of different data race analysis tools on Data Race Bench micro-benchmarks.

	True Positive	False Positive	True Negative	False Negative
Dynamatic	51	0	85	32
HPCRace (without modifications)	67	31	60	16
TSAN (without modifications)	63	1	84	16
Archer	63	1	80	17

Note that HPCRace has better true positive rates than any tool, but it has the drawback of having an extremely high amount of false positives. TSAN has an extremely strong False

Positive rate, but falls behind the True Positives of HPCRace. Inspecting the results of our tool, we can see it combines the strengths of TSAN and HPCRace. Dynamatic is able to never report any false positives, which when compared to HPCRace's 31 false positives is a huge reduction. Additionally, Dynamatic has the best true negative rate at 85 detected, while still maintaining fairly good true positive numbers. We were able to reduce HPCRace's false positive rate dynamically, though it came at the cost of losing some true positives. Furthermore, our tool currently suffers from inheriting the false negatives of both the static and the dynamic tools, reporting both the 16 false negatives from TSAN and the 16 from HPCRace.

4. CONCLUSION

4.1 Final Remarks

Data races are a major type of bug in parallel programs. In this paper, we have presented our hybrid analysis tool Dynamatic. With Dynamatic, we hope to help programmers automatically detect data races within their programs. The hybrid race analysis tool utilizes HPCRace for static analysis and Google's Thread Sanitizer for dynamic analysis as they are both highly performant and useful tools within their fields. We have presented background information on both HPCRace and TSAN, and how they were modified to work together. Additionally, we discuss how Dynamatic utilizes the results from each tool to detect data races. Dynamatic is able to leverage the best elements of both types of analysis - the level of coverage that static tools provide, and the low false positive rates of dynamic tools. Thus, the tool is able to mitigate static or dynamic analysis drawbacks through its hybrid approach and analysis optimizations.

Dynamatic performs well when it comes to false positive reports on data races. Within the Data Race Bench micro-benchmarks, Dynamatic is able to correctly provide zero false positive reports. Additionally, Dynamatic has performed better than other tools in the number of true negatives detected. There are current faults in the tool, however. Dynamatic inherits the false negatives from both HPCRace and TSAN, and thus has a higher false negative rate.

We believe that Dynamatic has the potential to be a leading data race analysis tool in the industry. While there is still work to be done, early versions of the tool show a lot of promise.

4.2 Future Work

Dynamatic is currently undergoing further research and improvements. There are many ways to further improve the tool and to correct some of the shortcomings noticed in benchmarks. One such improvement is further optimizing how the static and dynamic tools communicate, as our solution currently is not able to utilize the benefits of both tools fully. Also, we can use dynamic profiling to determine hot spots in the source code to help the tool focus on these parts for smarter instrumentation choices and additional analysis. Lastly, our tool is only in a very basic state, not able to handle very large and complicated compilations. We want this tool to be able to handle large-scale real-world applications, so we hope to improve the capability of Dynamatic to accept such input.

REFERENCES

[1] Bora, U., Das, S., Kukreja, P., Joshi, S., Upadrasta, R., & Rajopadhye, S. (2020). LLOV: A Fast Static Data-Race Checker for OpenMP Programs. *ACM Transactions on Architecture and Code Optimization*. doi:10.1145/3418597

[2] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, Ian Karlin. DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools (best paper finalist). Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, pp. 11:1-11:14, ISBN 978-1-4503-5114-0, Denver, CO, USA, November 12-17, 2017. pdf

[3] Chunhua Liao, Pei-Hung Lin, Markus Schordan and Ian Karlin, A Semantics-Driven Approach to Improving DataRaceBench's OpenMP Standard Coverage, IWOMP 2018: 14th International Workshop on OpenMP, Barcelona, Spain, September 26-28, 2018, pdf

[4] 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. DOI:https://doi.org/10.1145/1791194.1791203

[5] S. Atzeni *et al*., "ARCHER: Effectively Spotting Data Races in Large OpenMP Applications," *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Chicago, IL, USA, 2016, pp. 53-62, doi: 10.1109/IPDPS.2016.68.