The Report Committee for Alexander Joshua Hoganson
Certifies that this is the approved version of the following Report:

SHM Racer: Dynamic Race Condition Detection Using Shared Memory Traps

APPROVED BY
SUPERVISING COMMITTEE:

August Shi, Supervisor

Robert Leyendecker

SHM Racer: Dynamic Race Condition Detection Using Shared Memory Traps

by

Alexander Joshua Hoganson

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2021

Acknowledgements

Professor August Shi and Robert Leyendecker provided excellent feedback and direction for this report. Professor Milos Gligoric’s lectures on the application of Java Pathfinder were especially applicable to some of the challenges faced in this project. While the implementation does not specifically utilize JPF style tools, the lectures about object listeners inspired the SHM Library design.

Abstract

SHM Racer: Dynamic Race Condition Detection Using Shared Memory Traps

Alexander Joshua Hoganson, M.S.E.

The University of Texas at Austin, 2021

Supervisor: August Shi

This project investigated different types of data races that can exist within C code, and then incrementally built a race condition detector, SHM Racer, using an assortment of thread safety violation/data race violation detection methods. The objective of this report was to demonstrate the tradeoffs/benefits of certain race condition finding tools, and construct a tool that could minimize false positive race errors while maximizing the number of found data races. Race condition detection using static analysis can log false positives/false negatives due to a missed understanding of complex synchronization patterns. Dynamic analysis can often minimize false positives, but its ability to slow runtime performance can make this type of analysis impractical. The decided approach was to develop SHM Racer: a shared memory library that provides a shared memory access interface to multiple threads. SHM Racer would also use its knowledge of thread contexts and near-miss analysis data to dynamically modify its shared memory response performance with the intent of expanding critical sections. The aim of this approach was to increase the likelihood of race condition occurrence without significantly altering runtime performance. Test observations demonstrated that while expanding critical sections does increase the probability of detecting data races without producing false positives, there are significant performance/race condition detection impacts if the critical sections are expanded without accommodating for happens-before and already found race errors. Using a combination of near miss analysis and dangerous pair pruning, there was an observable performance improvement in analyzed code performance. However, while there were no false positives detected by SHM Racer, there were still missed race violations due to the tool's effects on thread synchronization patterns. Even though SHM Racer did miss finding race violations, its false positive minimization and configurable delay thresholds make it very effective at catching basic data races without having to learn synchronization patterns.

Table of Contents

List of Tables

List of Figures

1. INTRODUCTION

1.1 Background

Concurrent systems, whether they are embedded Linux or multi-stack web applications, often require resource sharing between parallel system contexts. While there may be significant benefits associated with implementing concurrent systems, there are data correctness risks within concurrent system design. A concurrent system can be negatively influenced by race conditions: flaws which occur when the real time synchronization of events within a software system impacts the system's data correctness [1]. There are many aspects of concurrent software systems' designs that can cause race conditions to occur: context switches, signals, hardware interrupts, and data races. This report will specifically analyze the data race type: "simultaneous access to the same memory location by multiple threads, where at least one of the accesses modifies the memory location" [2].

1.1.1 System Impact of Data Races

The impact of data races can either be observable, or unobservable due to the flaky nature of concurrent synchronization patterns. For example, if a test suite executes all system functionality without observing a data race, that does not necessarily mean all code flows are protected from data races. The difficulty of locating race conditions during development has occasionally led to catastrophic system failures in the field. In 2003, a race condition inside a GE energy management system's alarm subsystem caused a power line fault to go unnoticed, resulting in the North American Blackout of 2003 [3]. Between 1985 and 1987 a data race bug in a computer-controlled radiation therapy machine known as the Therac-25 led to patients receiving massive overdoses of radiation and 6 deaths [4].

Figure 1.1 describes a thread unsafe function **setValue** that should increment *value* by 30 every time it is called. If there was only a single thread X executing **setValue**, the end *value* would predictably be 32 + 30 = 62. However, if another thread, thread Y, were introduced after thread X has loaded *value* into *tmp* but before thread X has executed its final write to *value*, then the result could vary from 62 to 92. 62 would result if thread X was first to load value into *tmp*, but executes its last write to value after thread Y has executed its last write. 92 would result if thread X was first to load *value* into *tmp*, but executed its last write to value before thread Y loads *value* into *tmp*. 92 is the intended result of any two calls to this function: 32 + 2*(30) = 92.

```
volatile int value = 32;

void setValue(void *args) {
    int tmp = value;
    for(int i = 0; i < 30; i++) {
        value = (tmp++);
    }
}
```

Figure 1.1: Basic Data Race Violation Example

Even with a basic example the range of possible outcomes varies significantly when a race condition occurs. If the data race leads to *value* becoming the correct value 92, the data race still poses a threat to correctness that could be exposed during another execution.

Another example of data races impacting program behavior are “dirty writes” [5], where one thread is attempting to write a value to an address while another thread is also trying to write to the same address. If the write procedure to update volatile shared memory does not have enforced serialization constraints, then data corruption is a possible outcome. Shared memory libraries as described in section 1.1.3 often mitigate these errors using a serialized write lock mechanism. It’s worth noting that the absence of “dirty-writes” does not prevent race conditions like those present in Figure 1.1 from occurring, since even if the data accesses themselves are *serialized*, their *ordering* may not be guaranteed.

```
typedef struct{
    uint8_t a;
    uint8_t b;
}sCustomType;

volatile sCustomType var = {.a = 0, .b = 0};

void threadX_setVar(void *args) {
    var.a = 1;
    var.b = 1;
}

void threadY_setVar(void *args) {
    var.a = 2;
    var.b = 2;
}
```

Figure 1.2: “Dirty Write” Example

Figure 1.2 shows two functions, **threadX_setVar** and **threadY_setVar**, that write different information to a shared variable *a*. If these two functions are executed concurrently, the ordering of the individual writes dictates the final result. If **threadX_setVar** finishes both writes before **threadY_setVar** starts writing, then the final result would be {2,2}, but if there’s an interleaving of writes between the various threads then the result could be {1,2} or {2,1}. If the *sCustomType* is trying to represent a UINT16 type variable then the non-serialization of writes in this example could potentially leave *var* in a corrupt state.

Race conditions such as Write-after-read (WAR), Read-after-write (RAW), and Write-after-write (WAW) are the primary subject of this report. WAR occurs when a thread Y writes to a variable before it is read by thread X, such that thread X uses the newly written value instead of the older value. RAW occurs when a thread Y reads a value right before thread X overwrites it, such that thread Y uses an older value. WAW occurs when a thread Y writes to a value before thread X covers up that write with a write to the same value. All of these race conditions are explained using pipelined processor semantics in Table 1.1.

Race condition	WAR	RAW	WAW
Example	t1: R1 = R2 + **R4** t2: **R4** = R2 + R3	t1: **R1** = R2 + R4 t2: R3 = **R1** + R5	t1: **R1** = R2 + R3 t2: **R1** = R3 + R4
Explanation	t2 needs to wait for t1 to read R4 before writing to R4	t2 needs to wait for t1 to write R1 before reading R1	t2 needs to wait for t1 to write R1 before writing R1

Table 1.1: Race Conditions Explained Using Pipelined Processor Semantics

Using the information in Table 1.1, we can identify in Figure 1.1 that there are race conditions, but they are more abstract than pipeline processor semantics, and harder to intuitively spot. The biggest race condition is a RAW hazard that exists when thread Y is attempting to read *value* before X has fully finished writing *value*. There is also a potential for a WAR hazard where thread X is writing *value* before waiting for thread Y to read *value*. While a WAW race condition exists, that hazard does not impact the correctness of the function if the other race conditions are addressed.

1.1.2 Data Race Detection Techniques

Race condition detection has been researched for decades, and there is still not a standardized technique in part because finding all "feasible general or data races is an NP-hard problem" [6]. The difficulty of finding all feasible general data races stems from a dependence on "intricate sequences of low-probability events"; a slight modification to a program's execution state can either cause the fault to affect the program or pass without harm [7]. Even when data races impact a program’s state, data race errors can be still difficult to detect since an affected program often would not indicate a failure has occurred. Researchers have proposed and developed static synchronization analysis, dynamic synchronization analysis, and dynamic delay-injection tools to detect data races within concurrent programs.

Static synchronization analysis tools execute during build time and attempt to analyze a code base for obscure code paths or synchronization patterns that can lead to data races. Warlock [8] and RacerX [9] are examples of static analysis tools that build control flow graphs (CFGs) of programs and attempt to cycle through all CFG paths while checking for synchronization states. Both of these tools, while they are more likely to find more data races since they can analyze rare code flows not covered during runtime, are also more likely to report false positives if those tools encounter unknown synchronization patterns.

Dynamic synchronization analysis tools execute during runtime and attempt to analyze the dynamic state of locking mechanisms and shared memory accesses as they occur. Eraser analyzes held lock states at run time and determines if shared data is accessed during intervals without a held lock [10]. Happens-before (HB) detectors, such as RaceTrack [11] and ThreadSanitizer [12], take advantage of Lamport’s HB relation to establish a temporal order between memory access operations and determine whether the order can be guaranteed to have one operation “happen before” the other [13]. These tools do not suffer from the same false positive issues exhibited by static analysis tools because analysis is limited to executed code paths. However, this reliance on executed code paths also implies these tools cannot certify a program of being free from data race errors. Another issue with dynamic race detection tools is their impact on a program’s dynamic performance, which interestingly could lead to fewer data race detections [14].

Delay injection tools attempt to expose race conditions via inserting delays during runtime. These tools are excellent at preventing false positives because they only modify the timing of the program, and they do not make assumptions about code synchronization mechanisms. RaceFuzzer [15], CTrigger [16], DataCollider [17], and TSVD [18] are examples of delay injection tools. RaceFuzzer and CTrigger perform dynamic and static analysis of a program's synchronization/data access patterns to determine the optimal locations to insert delays. DataCollider performs little to no analysis and places delays within a program in various locations with a fixed probability. TSVD uses a lighter weight dynamic analysis than CTrigger or RaceFuzzer to dynamically insert delays in thread unsafe areas of a program based on near miss (NM) and runtime HB **inferences**. One of the drawbacks of delay injection tools are that they directly influence program timing. Dynamic tools with large delays can miss race conditions that exist in shorter time intervals [14].

1.1.3 SHARED MEMORY LIBRARIES

Shared memory (SHM) provides numerous advantages to a software system: less disk space, centralized maintenance of data definitions, and multi-process communication capabilities [19]. A drawback of using SHM is that read/write operations require a synchronization mechanism in order to prevent data corruption problems or data races. SHM libraries provide a synchronized interface to SHM with techniques that enforce serialization of read/writes.

However, even if a SHM library can enforce serialization, a SHM library cannot enforce ordering of read/write requests. Since the ordering of read/write execution determines whether a race condition exists, a serialized SHM library cannot mitigate race conditions. Figure 1.3 shows an example of two functions, **threadX_setVar** and

threadY_setVar, that access SHM via SHM library functions **SHM_READ** and **SHM_WRITE**. If these functions were executed concurrently, a race condition can still influence the value of *a* even with the SHM library providing memory access serialization.

```
volatile uint8_t a = 0;

void threadX_setVar(void *args) {
    uint8_t tmp = 0;
    SHM_READ(&tmp, &a);
    tmp |= 0x10;
    SHM_WRITE(&tmp, &a);
}

void threadY_setVar(void *args) {
    uint8_t tmp = 0;
    SHM_READ(&tmp, &a);
    tmp |= 0x20;
    SHM_WRITE(&tmp, &a);
}
```

Figure 1.3: Serialized SHM Library with Race Conditions

The threads executing **threadX_setVar** and **threadY_setVar** are both attempting to perform a version of Read Modify Write (RMW). However, these RMWs do not occur atomically and are broken up into three smaller parts: the read(R), the modify(M), and the write(W). Since neither of these operations have their RMW ordering guaranteed relative to another thread's write operations, there is a possibility that *a* can be equal to any of 0x10, 0x20, or 0x30. If the RMW is treated as a single atomic operation, then the final result will consistently be 0x30 regardless of which thread starts executing first. It can be tempting to assert that since making this particular RMW operation atomic resolves the race condition, that extending the SHM library with a **SHM_RMW** command will fix most race conditions. However, in large-scale enterprise codebases, usually the M part of RMW is not as straightforward as applying an OR mask and varies significantly between processes.

1.2 Problem Definition

Now that shared data accesses are the norm in concurrent software systems, race conditions/data races are consistent problems challenging software determinism and correctness. Many tools, including those referenced in section 1.1, have detected race conditions with varying degrees of success. At an enterprise scale, the best tools will attempt to limit the number of false positives while also attempting to find as many true positives as possible. Minimizing false positives prevents engineering organizations from spending time on non-issues, whereas maximizing reported true positives helps an engineering organization improve concurrent software quality. TSVD had very promising results, although that tool was specifically designed to analyze C# programs [18]. Since C/C++ are primarily the languages used for embedded system development [20], there is a motivation to implement a TSVD themed tool in C or C++.

Therefore, we propose SHM Racer, a solution modeled after TSVD that uses a C-based SHM library to set data race traps according to near miss data. This will minimize the quantity of false positives while providing a C compatible technique to perform near miss (NM) data collection and maximize reported true positives.

2. SOLUTION: SHM RACER

SHM Racer was built iteratively in 2 phases, starting with a basic solution that inserts fixed weight delays within each SHM access instruction (Phase 1), and then incrementally adding features from TSVD to optimize SHM Racer's detection and runtime performance (Phase 2). This solution is specifically written to perform race condition analysis on sets of concurrent pthread programs that use a common SHM library. The constraint that all programs must use a common SHM library still accommodates enterprise-scale embedded software development environments. SHM access interfaces at the enterprise level are moving towards standardization [21] and thread/process meta-data can easily be made into an input requirement using compiler macros. SHM Racer code is currently located within the SHM Racer GitHub v1.0 release [22].

2.1 Solution Architecture

SHM Racer is primarily a SHM library facilitating SHM read or write transactions. This SHM library receives concurrent requests, serializes its SHM activities, and performs the requested read/write actions. In addition to its routine operations, this SHM library leverages pthread-specific context information to set data race *traps* and delay responses to requests. A *trap* (section 2.2.2) is a technique to expand a critical section such that another thread can expose a WAR, RAW, or WAW race condition. The phase 1 design will implement a fixed delay trap mechanism. The phase 2 SHM library will cache NM race conditions, or race conditions that could have occurred if the gap between two SHM requests were reduced below a certain threshold. NM data will be combined with TSVD's dangerous pair pruning techniques to dynamically construct a performance-optimal trap set.

The high level SHM Racer design is illustrated by Figure 2.1. Pthread Clients issue **SHM_OP** requests to an Interface Module, which forwards these requests to a Trap Module. The Trap Module determines trap delays, possibly holds current requests until traps clear, and then returns to the Interface Module. The Interface Module then forwards **SHM_OPs** to the SHM Access Module, which unwraps the **SHM_OPs** and performs a serialized read/write on the actual SHM. Finally, the Interface Module will send a return status to the Pthread Client in addition to SHM data if the request was a read.

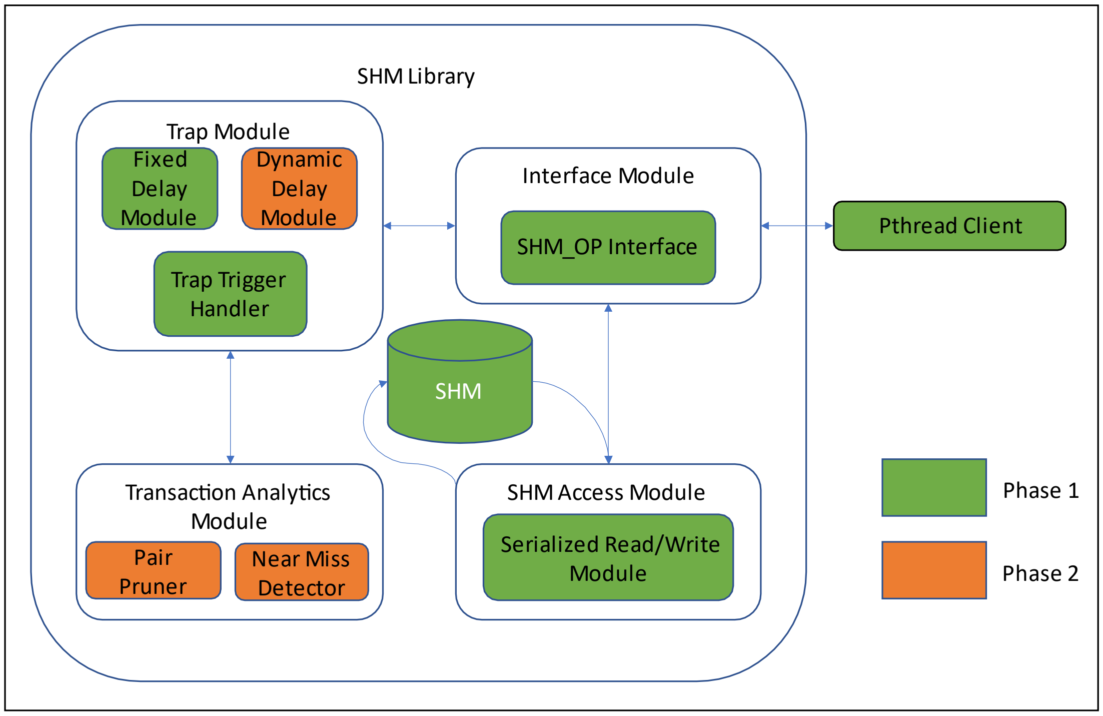

Figure 2.1: Solution High Level Design Diagram

2.2 Phase 1 Solution

2.2.1 Interface Module

The interface module allows pthread clients to access the SHM Library over a common **SHM_OP** interface. The **SHM_OP** interface will automatically extract calling context meta-data (thread id, file, function, line, etc.) from the pthread client, and pass this data to the Trap and SHM Access Modules within the SHM library. Interface Module interactions with the Trap Module may lead to delayed responses to **SHM_OP** requests, since the Trap Module may attempt to catch race conditions by intentionally holding a request.

2.2.2 Trap Module

The Trap Module is responsible for setting traps and detecting **SHM_OP** requests that trigger traps. A trap essentially delays one thread's **SHM_OP** request while waiting for other threads to execute **SHM_OPs** during that delay interval. Figure 2.2 illustrates this concept using two threads X and Y. When a thread X executes a read **SHM_OP**, the Trap Module will delay thread X. Meanwhile, if thread Y executes a write **SHM_OP** on the same memory location during the time thread X has a trap delay, then the Trap Module will record a trap violation that indicates a WAR race condition. When this module receives a **SHM_OP** request, this module will first check if there are any active traps that are being triggered by the current request. This involves the Trap Module searching through a table and determining whether a trap entry has memory overlapping with the current request and at least one of those **SHM_OP** access types is a write. If a trap violation is detected, and it was not previously detected, then an error message will be logged.

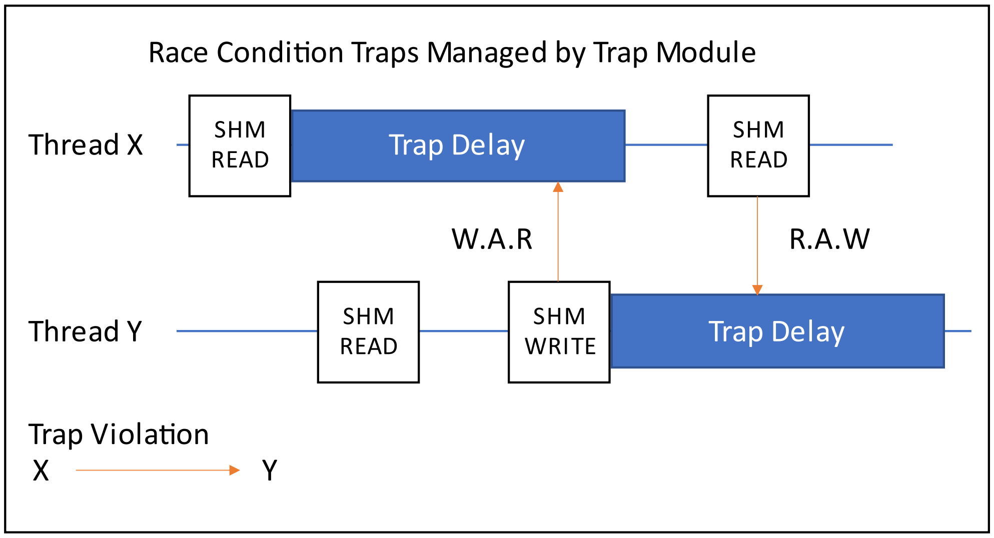

Figure 2.2: Race Condition Trap Example

Once trap violations caused by the current request have been recorded, then the Trap Module will determine the trap delay for the current **SHM_OP** request. The phase 1 Trap Module will use a fixed trap delay. The Trap Module will insert an entry into the trap table, sleep by this delay, and remove the trap from the trap table before allowing the **SHM_OP** read/write operation to complete.

Table 2.1 represents a concurrent execution of the **setValue** function from Figure 1.1. For this example, the trap delays vary based on what thread executes a **SHM_OP**. Thread X will use a 1.0 second fixed trap delay and thread Y will use a 0.5 second fixed trap delay. Having different fixed trap delays for threads X and Y allows the detection of a WAR that otherwise would not have been detected with uniform fixed delays.

Time(s)	Thread ID	SHM_OP Type	Trap Violation	Trap Delay
0.0s	X	Read	N/A	1.0s
0.1s	Y	Read	N/A	0.5s
0.6s	Y	Write	WAR	0.5s
1.0s	X	Write	WAW	1.0s
1.1s	Y	Write	WAW	0.5s

Table 2.1: Execution Trap Sequence Table for Figure 2.1

Thread X in this particular scenario is first to issue a **SHM_OP** command to read *value* into *tmp*. SHM Library checks the list of known traps, sees there are none currently set, and then sets a trap for 1.0 second(s). While the SHM Library holds thread X for 1.0 second(s), thread Y issues a **SHM_OP** command to read *value* into *tmp*. Read after read is not a data race, so the SHM Library does not log a trap violation. The SHM Library does however decide to set a trap for thread Y for 0.5s. Since there are no concurrent accesses to SHM from thread X during this time since thread X is held in a trap, thread Y's trap expires without catching any violations. Then thread Y executes its first write to value at 0.6 second(s), which violates the trap set by thread X at 0.0 second(s). Thread Y sets a trap for 0.5 second(s) after reporting its own trap violation. When thread X's original trap releases, then thread X will execute its first write to value at 1.0 second(s), violating the trap set by thread Y at 0.6 second(s). Finally, when thread Y executes its second write to value it will violate the trap set by thread X at 1.0 second(s). This particular trap setting sequence actually catches all but two potential race conditions within this function, and contains no false positives. If thread Y has its read operation delayed to occur after thread X has a trap set during a write, then the trap module would have detected a RAW. Also, if thread Y held its trap for the read longer, then thread X could have triggered a WAR violation instead of thread Y. The WAR trap violation is mutually exclusive to whichever thread violates it first in this particular example.

This example provides concrete evidence that delay injection tools can find race condition issues, but with the caveat that they cannot necessarily find all race conditions due to delay injection effects on ordering. Since a delay can expand a critical section using a trap, this expansion theoretically should make it easier for other threads to trigger trap violations. In reality, the expansions of a critical section can prevent the trapped threads from triggering other traps as they wait in standby states for extended periods of time. Not only can delays cause race conditions to be missed, but they can also negatively impact program performance. For the Figure 1.1 example, if all thread X **SHM_OP**s are delayed by 1.0 seconds(s) and all thread Y **SHM_OP**s are delayed by 0.5 second(s), then the expectation for concurrent execution time is 31*(1.0s) = 31.0 second(s) rather than less than 0.1 seconds. This same observation occurred when TSVD was executed on programs where the delays were fixed and did not decay, at one time recording maximum overhead of 6600% for a single module [18].

2.2.3 SHM Access Module

The SHM Access Module reads from or writes to volatile shared memory based on the request from the Interface Module. This module has serialized data access, and simply moves data between volatile shared memory and **SHM_OP** request data structures.

2.3 Phase 2 Solution

The phase 2 solution inherits the Interface Module, Trap Module, and SHM Access Module from the phase 1 design. The phase 2 design will attempt to trim down the trap delays so that they are small enough to not severely impact program performance while also exposing as many race conditions as possible. Near miss (NM) data can be used to track dangerous pairs of memory accesses, eliminate pairs that would cause excess program delays, and provide future program executions with prior program execution data. This solution persists NM and trap data across execution cycles, allowing each additional execution to use information gathered in previous runs to influence trap settings in future runs. This persistent concept is represented by a variable known as a cycle count, or the number of program executions since the NM and trap data were clear. We added the Transaction Analytics Module to the Phase 2 Solution to help determine dynamic trap delays based on NM data. Modifications were made to the Trap Module to consume data provided by the Transaction Analytics Module and calculate optimal trap delays. HB inferences were originally considered as a candidate component for the phase 2 solution, but that component was excluded due to its ability to cause true race conditions to be ignored (section 4.1). The time required to develop HB inference detection also contributed towards this decision.

2.3.1 Transaction Analytics Module

The Transaction Analytics Module records NM race condition violations and uses data collected by the Trap Module to construct a dangerous pair table indicating pairs of NM **SHM_OP**s that need to influence trap delays. As **SHM_OP**s are serviced, their execution times are compared to previously serviced **SHM_OP**s. If at least one of two **SHM_OP**s $\{a,b\}$ are a write, the two operations occur within a certain threshold D_{nm} (Table 2.2), and a trap does not currently exist for that pair, then those two **SHM_OPs** are a NM pair, or NM(a,b). If a trap violation does not exist for a **SHM_OP** pair then that pair gets added to the dangerous pair table DP_List_a, a table that represents dangerous pairs that could lead to triggered race conditions against SHM_OP a's traps. If a **SHM_OP** b

triggers a trap that was set by a then the Pair Pruner removes any dangerous pairs from DP_List_a that link to b.

The DP_List_a stores a single preferred trap delay for each dangerous pair NM(a,b) using the average of NM intervals for that pair. This module is also responsible for detecting when a trap gets set for a **SHM_OP** a, and incrementing $\mathrm{T_{nm}}(a)$: a quantity signifying the number of traps set since a NM occurred. Whenever a new NM involving a **SHM_OP** a occurs, then this module will clear $\mathrm{T_{nm}}(a)$. $\mathrm{T_{nm}}(a)$ represents the staleness of NM data for a given **SHM_OP** a since it counts the number of traps set since the most recent NM.

2.3.2 Trap Module

The phase 2 Trap Module has the sole purpose of creating delays that are performance optimal and catching as many race conditions as possible. The internals of the phase 2 Trap Module are mostly the same as the phase 1 Trap Module with the exception of how delays are determined. Those delays, represented by D_i, are a either a default delay D_o or the averaged NM data for a **SHM_OP** a, $\delta_\mathrm{i}(a)$, multiplied by a coefficient C_i that corresponds to the program's real time assessment of race condition risk with a particular **SHM_OP**. The formulas for calculating D_i and other relevant variables are provided in Table 2.2. C_i is a decay function that scales $\delta_\mathrm{i}(\mathrm{a})$ lower as new traps get set due to $\mathrm{T_{nm}}(a)$. A newly detected NM will increase C_i back to 1, whereas each new trap will scan the relevant near miss entries and exponentially decay C_i further. A single NM detected during program initialization for example will not be allowed to penalize future runtime performance if that NM only occurs one time. If NMs are transformed into caught race conditions, then they will be removed from the dangerous pair table to prevent race condition redetection and unnecessary trap delays.

Variable	Formula
A/B	Set of all SHM_OPs from the perspective of the primary thread(A) or secondary thread(B) {Time, Thread ID, Read/Write, File/Function/Line, Offset}
a	$= \in \mathrm{A}$
b	$= \in B$
$\mathrm{D_{nm}}$	$= 5 \text{ seconds}$
$\mathrm{D_o}$	$= 0 \text{ seconds}$
$\mathrm{D_i(a)}$	$= \begin{cases} \mathrm{\delta_i(a) * C_i(a)}, & \mathrm{\delta_i} \text{ Defined} \\ \mathrm{D_o}, & DP_{List_a} = \emptyset \end{cases}$
$\mathrm{C_i(a)}$	$= \dfrac{1}{(3^{\mathrm{T_{nm}(a)}})}$
$\mathrm{\delta_i(a)}$	$= \dfrac{1}{\mathrm{n}}\left(\sum_{\mathrm{\delta_{nm} \in DP\_List_a}} \mathrm{\delta_{nm}}\right)$
$\mathrm{DP\_List_a}$	$= NM(a,b)\ \forall\, b\ \in B$
$\mathrm{NM(a,b)}$	$= \begin{cases} \mathrm{b.time - a.time}, & \begin{array}{c} \exists\ \mathrm{x} \in \{\mathrm{a,b}\}\ such\ that\ x.rw = w \\ \mathrm{a.offset \cap b.offset} \\ b.time - a.time \leq \mathrm{D_{nm}} \\ \nexists\ \{a,b\}\ in\ Caught\ Traps \end{array} \\ \mathrm{N/A}, & \text{Otherwise} \end{cases}$
$\mathrm{T_{nm}(a)}$	$= \begin{cases} \mathrm{T_{nm}(a) + 1}, & \text{Trap set for nm pair} \\ 0, & \text{Near miss occurs for nm pair} \end{cases}$

Table 2.2: Phase 2 Delay Formulas

3. TESTING/RESULTS

3.1 Testing Configuration

SHM Racer phase 1 and 2 solutions, as well as ThreadSanitizer [12] for comparison, were tested on 12 “thread set” C programs; these programs were written specifically for testing, and exercise a diverse combination of synchronization patterns and data races. These test programs are located within the SHM Racer GitHub v1.0 release [22]. We developed these programs based on previous experiences with race conditions in school and work settings, with an emphasis on creating thread safe and thread unsafe programs. A “thread set” data structure was used to define sets of concurrent threads and their corresponding functions to execute as pthreads, in addition to global timeout settings. When the test program executes, it iterates through and runs each thread set, clearing existing lock/synchronization states in between each set. Threads can either exit normally or get cancelled by the main thread if a timeout occurs. At the end of each thread set a log is printed indicating the amount of time the thread set took to complete in addition to detected race conditions.

Since this tool analyzes race conditions through a shared memory library, the thread sets under test had to access volatile shared memory using **SHM_OP** commands. Other synchronization mechanisms like semaphores or mutexes did not require additional modification. Trap delay functionality was disabled during ThreadSanitizer evaluation, but programs still accessed SHM through the **SHM_OP** command interface.

The thread set programs were closely analyzed before testing to determine potential race condition violations. Overall, the thread sets generated for testing were 50% thread safe and 50% data race prone. There is an equal emphasis on testing thread safe programs since the presence of thread safe techniques can potentially slow the performance of SHM

Racer for both phase 1 and phase 2 designs. A sample thread set with 5 threads and 60 data races, THREAD_SET_RMW_RACE_MAX, is provided in Figure 3.1. There are a few macro functions like **THREAD_SET_BUILDER**, **THREAD_FUNC_BUILDER**, and **PTHREAD_HELPER_BUILDER** that help the pre-compiler build thread sets. Basically, this code will create a thread set with 5 pthreads where each pthread will execute **readModifyWrite** once started and timeout after 100 seconds.

The phase 1 SHM Racer had its fixed delay interval altered from 0 to 5 seconds between testing runs, whereas the phase 2 SHM Racer had its cycle count adjusted from 0 to 6 cycles to allow NM data on previous runs to be used in later runs. This was done specifically to observe how these parameters can influence race condition detection capabilities and overall runtime.

```
void readModifyWrite(void *args) {
    uint8_t tmp = 0;
    SHM_OP(SHM_READ, a, tmp);
    tmp |= 0x20;
    SHM_OP(SHM_WRITE, a, tmp);
}

PTHREAD_HELPER_BUILDER(RUN_FUNC_RMW_RACE, readModifyWrite)

THREAD_SET_BUILDER(THREAD_SET_RMW_RACE_MAX, 100)
    THREAD_FUNC_BUILDER(0, RUN_FUNC_RMW_RACE),
    THREAD_FUNC_BUILDER(1, RUN_FUNC_RMW_RACE),
    THREAD_FUNC_BUILDER(2, RUN_FUNC_RMW_RACE),
    THREAD_FUNC_BUILDER(3, RUN_FUNC_RMW_RACE),
    THREAD_FUNC_BUILDER(4, RUN_FUNC_RMW_RACE)
}
```

Figure 3.1: THREAD_SET_RMW_RACE_MAX Example

3.2 Results

3.2.1 ThreadSanitizer Results

ThreadSanitizer did not detect any race conditions when programs accessed shared memory through the serialized SHM Library. We made additional changes to the SHM

Library to specifically deserialize the SHM Access Module for ThreadSanitizer to see if those changes led to race condition detections. This modification resulted in a single RAW true positive detection within THREAD_SET_COUNTER_RACE_DOUBLE. Running this test multiple times yielded varying detection results, but none of those runs produced more than 2 race condition detections. One of the potential reasons for ThreadSanitizer's poor performance is its reliance on shared data access occurring within a relatively short window before reporting a data race violation. The thread sets under test execute over a relatively short run time and do not appear to have two concurrent accesses occurring close enough together during normal execution to trigger a ThreadSanitizer warning.

3.2.2 Phase 1 Results

Overall, the results were very promising in terms of finding race conditions. However, due to the frequency of delays the runtime of thread sets were often greatly stretched, with some thread sets exceeding timeouts. Condensed phase 1 testing results with data race thread sets are in Table 3.1. Complete results are located in Appendix A with the label "Phase 1 Test Results". There were no false positives reported in any of the examples, but there were 57 missed race conditions with the best-case fixed delay. The missed race conditions occurred specifically in thread sets like THREAD_SET_RMW_RACE_MAX where ordering made it impossible to actually catch race conditions in traps on the first run.

Adding fixed length trap delays enforced a global penalty to shared memory accesses that ultimately led to unnecessary decreases to program performance without a meaningful increase in race condition findings. In some cases, the increase in the trap delay actually led to the phase 1 design finding more race conditions. For example, a 5 second trap delay caught 33 race conditions versus 32 race conditions caught with a 1 second trap delay. However, this added detection came with a 735 second delay to thread set performance whereas a smaller delay would have sufficed. As the fixed delay increased to 10 seconds all thread sets with multiple **SHM_OPs** delayed significantly without catching more race conditions than the 5 second delay.

Test Case	Race Conditions Found Given Fixed Delay		Actual Race Conditions
	1s	5s	
THREAD_SET_COUNTER_RACE_DOUBLE	3	3	6
THREAD_SET_COUNTER_MIXED	3	3	6
THREAD_SET_RMW_RACE_DOUBLE	2	2	6
THREAD_SET_RMW_HB_FAKE	1	2	6
THREAD_SET_MAILMAN	3	3	6
THREAD_SET_RMW_RACE_MAX	20	20	60
TOTAL	32	33	90

Table 3.1: Phase 1 Condensed Race Conditions Found Results

3.2.3 Phase 2 Results

Similar to the phase 1 testing, the phase 2 testing covered the same test cases and had no false positives. Condensed phase 2 testing results with data race thread sets are in Table 3.2. The complete phase 2 test results are located within Appendix A with the label “Phase 2 Test Results”. Instead of executing the test cases using a configurable fixed delay these test cases were executed using a varying cycle number to run through thread sets multiple times. As the cycle numbers were increased, the number of race conditions detected increased until eventually plateauing for most test cases. The phase 2 SHM Racer detected more race conditions than the phase 1 SHM Racer for all thread sets at 72 with 6 cycles, but higher cycle counts contributed to longer runtimes up to 183 seconds for 6 cycles.

Test Case	Race Conditions Found Given Cycle Count		Actual Race Conditions
	2 Cycles	6 Cycles	
THREAD_SET_COUNTER_RACE_DOUBLE	2	5	6
THREAD_SET_COUNTER_MIXED	2	5	6
THREAD_SET_RMW_RACE_DOUBLE	2	4	6
THREAD_SET_RMW_HB_FAKE	2	3	6
THREAD_SET_MAILMAN	3	6	6
THREAD_SET_RMW_RACE_MAX	20	49	60
TOTAL	31	72	90

Table 3.2: Phase 2 Condensed Race Conditions Found Results

Even though this tool scales a trap delay according to the average near miss interval, there is still the issue of traps not catching race conditions. This is caused specifically by the Pair Pruner module not allowing traps to get set for SHM_OPs that have caught all potential near misses. If the first thread to execute THREAD_SET_COUNTER_RACE_DOUBLE for instance has successfully caught all of the second thread's race violations impacting the first thread's SHM_OPs, then on the next execution cycle, the first thread will not set any traps at all and therefore it is possible that it could not even have the opportunity to perform slow enough to trigger the second thread's traps set in response to near misses that occurred during previous execution cycles. However, this Pair Pruner behavior also leads to catching race conditions that would have otherwise not been caught by the fixed delay SHM Racer. By eliminating traps for resolved near miss pairs, the phase 2 design is able to speed up threads enough for those threads to trigger other race condition traps or set other traps during earlier execution windows for other threads to trigger. This behavior was evident in THREAD_SET_RMW_RACE_MAX in addition to THREAD_SET_COUNTER_RACE_DOUBLE.

Using the average of near misses for a given SHM_OP definitely allowed the tool to find a delay that tried to cover specific SHM_OP race conditions without penalizing other SHM_OPs. One of the issues with using the average near miss time is that the presence of traps themselves can cause these times to be longer than needed. When there is a near miss attributed to the trap delays' impact on dynamic performance, that near miss might not be achievable on the next cycle. This was certainly the case with THREAD_SET_COUNTER_RACE_DOUBLE, where a RAW near miss was detected. However, since all near miss pairs concerning a read in the first thread are resolved, that first thread will not trap delay that read on the following execution cycle. This type of behavior ensures the RAW can never be triggered. However, from the perspective of the second thread, the near miss for that write is still cached and therefore the second thread will continue to set traps for each of its 30 write operations, assuming that each trap has an opportunity to catch the first thread's read operation. This is a big motivation to use a delay decay function that takes into consideration the age of a near miss, such that the second thread will not continue to suffer a performance penalty for an older near miss.

3.2.4 Comparing Phase 1&2 Results

Both phase 1 and phase 2 SHM Racers performed better than the other at some points during testing. Table 3.1 demonstrates the phase 2 SHM Racer does well as the cycle count increases, whereas the phase 1 has an exponential increase in runtime without seeing a bigger improvement in race condition detection. However, as the cycle count decreases for the phase 2 SHM Racer, it has potentially a lot more delays to insert because it has just recently learned the average near miss times, and has not run long enough to start pruning dangerous pairs. If there are many near misses close to the near miss threshold, then the average near miss delay will skew high. In addition to the trap delays being much larger, the phase 2 SHM Racer also needs to run through the code multiple times. If the race conditions are all relatively close to each other, the Trap Module could possibly select delays that are much too large, and skewed by the larger near miss intervals created by the trap delays.

Design Phase	Race Conditions Found	Runtime (seconds)
PHASE 1(5 sec)	33	737
PHASE 2(6 cycle)	72	183
PHASE 1(0.00001 sec)	31	2
PHASE 2(2 cycle)	31	28

Table 3.3: Phase 1 vs. Phase 2 Comparison

The phase 1 SHM Racer struggled much more than the phase 2 SHM Racer when handling thread safe programs with fixed delays over 1 second. The phase 2 SHM Racer took 34 seconds to execute 6 cycles of THREAD_SET_COUNTER_SAFE_DOUBLE, but the phase 1 SHM Racer took 63 seconds using a 1 second delay to execute just 1 cycle. This is an interesting observation, because it demonstrates how the NM data combined with the exponential decay function helps the phase 2 SHM Racer naturally lower its trap delays as dangerous pairs become less dangerous. The phase 1 SHM Racer's fixed delays cause 1 second traps to be set nearly 60 seconds after what the phase 2 design would have considered to be a NM.

With race condition detectors, the primary incentive is to find as many data races as possible in a reasonable amount of time. This is why we believe the phase 2 SHM Racer is a better option than the phase 1 SHM Racer. While phase 1 can do exceptionally well runtime-wise with smaller interval data races, its inability to adjust its delay during runtime makes it very difficult to accommodate wider data race intervals, and find extra race conditions that are hidden by fixed trap delays. Phase 2 achieved a higher race condition detection ability without compromising as much on runtime performance. The phase 2 design detected more than double the race conditions detected by the phase 1 design, and had its highest detection result occur with a runtime that was roughly 1/4th that of the phase 1 design.

4. FUTURE WORK

Phase 1 and 2 designs do not currently have a technique to prevent inserting delays in locations where there are HB relationships. This could lead to excess delays and unnecessary increases to the SHM Racer's runtime performance. While originally considered as a candidate for the Phase 2 design, HB inference detection was not included because it could make HB inferences that obscured true race conditions. HB inference detection could be designed to more accurately identify true HB pairs and mitigate the risks of making invalid HB inferences if we had more time.

4.1 HB Inference Detection

```c
volatile uint8_t a = 0;
sem_t x_to_y_mutex;

void *threadX_setVar(void *args) {
    uint8_t tmp = 0;
    SHM_READ(&tmp, &a);
    tmp |= 0x10;
    SHM_WRITE(&tmp, &a);
    sem_post(&x_to_y_mutex);
}

void *threadY_setVar(void *args) {
    sem_wait(&x_to_y_mutex);
    uint8_t tmp = 0;
    SHM_READ(&tmp, &a);
    tmp |= 0x20;
    SHM_WRITE(&tmp, &a);
}

int main() {
    sem_init(&x_to_y_mutex, 0, 0);
    pthread_t X, Y;
    pthread_create(&X, NULL, threadX_setVar, NULL);
    pthread_create(&Y, NULL, threadY_setVar, NULL);
    pthread_join(X, NULL);
    pthread_join(Y, NULL);
    sem_destroy(&x_to_y_mutex);
    return 0;
}
```

Figure 4.1: HB Relationship Example

To help explain HB inference, Figure 4.1 shows a simple thread safe program based on Figure 1.3 that exhibits a HB relationship between thread X and thread Y. In this example the semaphore design leads to **threadX_setVar** executing before **threadY_setVar**, since the mutex is initialized to 0 and **threadY_setVar**’s **sem_wait** depends on **threadX_setVar** incrementing the mutex with **sem_post**. If this program were tested using either the phase 1 or 2 design, unnecessary traps would get set at each **SHM_OP**, and the dynamic performance of the program would be slowed exactly in proportion to the inserted delays. HB relationships would ideally be inferred by execution, and used to avoid delaying code paths that have them.

If **threadX_setVar** sets two traps that both lead to **threadY_setVar** executing its **SHM_OP**s after thread X’s traps’ have cleared, the traps’ impact on program performance can be used to infer a HB relationship. The sequence graph in Figure 4.2 outlines the HB inferences that a future design could make. If the introduction of trap delays by thread X causes thread Y’s **SHM_OP**s to be delayed at least by that amount then a HB inference will be made.

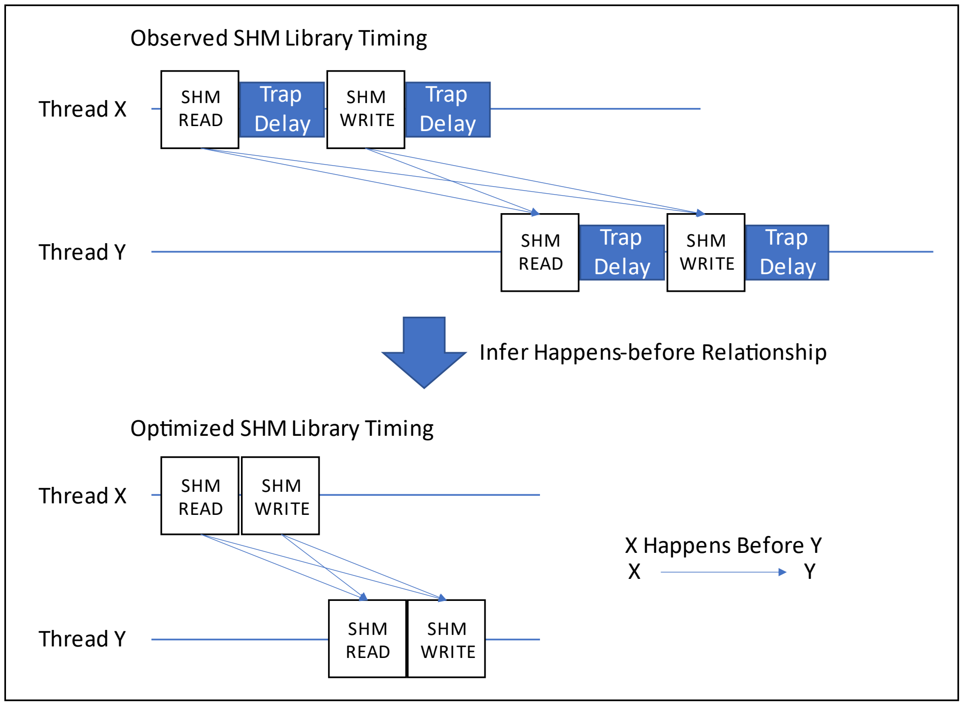

Figure 4.2: Inferring HB Relationships

This HB inference approach has a few flaws, specifically if the reason for thread Y's delayed **SHM_OP**s was due to non-synchronization-based behavior. If **threadY_setVar** replaced the **sem_wait** with a **sleep**(*t*), then a HB inference would be made that did not actually exist, and a trap delay in thread X roughly greater than *t* could have exposed race conditions. If HB was used to prevent traps from being set by thread X, then it is possible the HB inference would be responsible for preventing race condition detection in specific executions. This is the major reason why HB inference detection was excluded from the final phase 2 design. However, with more time to solve these issues, HB inference could be a valuable runtime optimizing feature for SHM Racer.

5. CONCLUSION

This project analyzed various types of race condition detectors and developed a delay injection tool, SHM Racer, based on the TSVD near-miss approach [18]. SHM Racer used a shared memory library design to listen for shared memory access requests and set traps for race condition violations. In the first phase of the design, the trap module used fixed length delays. Testing data showed that while the tool could catch race conditions, fixed delays did not provide enough race condition coverage or scalable runtime performance. The second phase of SHM Racer used dynamic delay calculations based on near miss race condition detections and was able to detect more race conditions after cycling through program sets multiple times. While the second phase design improved on the first phase design, there were still missed race violations due to delay injections causing near misses to occur that would not hold true on future execution cycles. Altogether there were no false positives, and the tool caught 80% of race condition violations during testing. An adjustment to this tool that would increase the number of race violations found is setting shared memory traps from both perspectives of a violation instead of just from the perspective of the older shared memory access. If the tool had this improvement, it could hold an operation it thinks can trigger a trap belonging to another thread that has not been set yet. Also, even though HB inferences can lead to race violations not being detected, this tool could certainly benefit from HB inferences eliminating unnecessary delays. The second phase SHM Racer's configurability, solution architecture, and race condition detection performance make for an effective delay injection race detection tool, but certainly with room for improvement.

Appendix A

Phase 1 Test Results (Race Conditions Found)

Test Case	Race Conditions Found Given Fixed Delay				Actual Race Conditions
	0s	0.00001s	1s	5s	
THREAD_SET_EMPTY_SINGLE	0	0	0	0	0
THREAD_SET_EMPTY_DOUBLE	0	0	0	0	0
THREAD_SET_COUNTER_RACE_DOUBLE	0	3	3	3	6
THREAD_SET_COUNTER_SAFE_DOUBLE	0	0	0	0	0
THREAD_SET_COUNTER_MIXED	0	3	3	3	6
THREAD_SET_READ_RACE_DOUBLE	0	0	0	0	0
THREAD_SET_RMW_RACE_DOUBLE	0	2	2	2	6
THREAD_SET_RMW_SAFE_DOUBLE	0	0	0	0	0
THREAD_SET_RMW_HB_DOUBLE	0	0	0	0	0
THREAD_SET_RMW_HB_FAKE	0	0	1	2	6
THREAD_SET_MAILMAN	0	3	3	3	6
THREAD_SET_RMW_RACE_MAX	0	20	20	20	60
TOTAL	0	31	32	33	90

Phase 1 Test Results (Runtime Performance)

Test Case	Execution Time Given Fixed Delay (seconds)			
	0s	0.00001s	1s	5s
THREAD_SET_EMPTY_SINGLE	<1	<1	<1	<1
THREAD_SET_EMPTY_DOUBLE	<1	<1	<1	<1
THREAD_SET_COUNTER_RACE_DOUBLE	<1	<1	32	155
THREAD_SET_COUNTER_SAFE_DOUBLE	<1	<1	63	310
THREAD_SET_COUNTER_MIXED	<1	<1	32	155
THREAD_SET_READ_RACE_DOUBLE	<1	<1	2	5
THREAD_SET_RMW_RACE_DOUBLE	<1	<1	3	10
THREAD_SET_RMW_SAFE_DOUBLE	<1	<1	5	20
THREAD_SET_RMW_HB_DOUBLE	<1	<1	5	20
THREAD_SET_RMW_HB_FAKE	2	2	5	12
THREAD_SET_MAILMAN	<1	<1	9	40
THREAD_SET_RMW_RACE_MAX	<1	<1	3	10
TOTAL	2	2	159	737

Phase 2 Test Results (Race Conditions Found)

Test Case	Race Conditions Found Given Cycle Number				Actual Race Conditions
	1	2	3	6	
THREAD_SET_EMPTY_SINGLE	0	0	0	0	0
THREAD_SET_EMPTY_DOUBLE	0	0	0	0	0
THREAD_SET_COUNTER_RACE_DOUBLE	0	2	5	5	6
THREAD_SET_COUNTER_SAFE_DOUBLE	0	0	0	0	0
THREAD_SET_COUNTER_MIXED	0	2	5	5	6
THREAD_SET_READ_RACE_DOUBLE	0	0	0	0	0
THREAD_SET_RMW_RACE_DOUBLE	0	2	3	4	6
THREAD_SET_RMW_SAFE_DOUBLE	0	0	0	0	0
THREAD_SET_RMW_HB_DOUBLE	0	0	0	0	0
THREAD_SET_RMW_HB_FAKE	0	2	3	3	6
THREAD_SET_MAILMAN	0	3	6	6	6
THREAD_SET_RMW_RACE_MAX	0	20	34	49	60
TOTAL	0	31	56	72	90

Phase 2 Test Results (Runtime Performance)

Test Case	Execution Time Given Cycle Number (Seconds)			
	1	2	3	6
THREAD_SET_EMPTY_SINGLE	<1	<1	<1	<1
THREAD_SET_EMPTY_DOUBLE	<1	<1	<1	<1
THREAD_SET_COUNTER_RACE_DOUBLE	<1	2	6	15
THREAD_SET_COUNTER_SAFE_DOUBLE	<1	3	10	34
THREAD_SET_COUNTER_MIXED	<1	2	6	15
THREAD_SET_READ_RACE_DOUBLE	<1	<1	<1	<1
THREAD_SET_RMW_RACE_DOUBLE	<1	2	6	22
THREAD_SET_RMW_SAFE_DOUBLE	<1	3	8	38
THREAD_SET_RMW_HB_DOUBLE	<1	3	8	38
THREAD_SET_RMW_HB_FAKE	2	8	12	36
THREAD_SET_MAILMAN	<1	3	5	5
THREAD_SET_RMW_RACE_MAX	<1	2	5	14
TOTAL	2	28	56	183

References

1. Huffman, D.. *The Synthesis of Sequential Switching Circuits.* Journal of the Franklin Institute, vol. 257, no. 3, Elsevier Ltd, 1954, pp. 161–190, doi:10.1016/0016-0032(54)90574-8.

2. Boehm, H.. *How to Miscompile Programs with "Benign" Data Races.* HotPar. 2011. pp. 1-4.

3. Andersson, G., et al. *Causes of the 2003 major grid blackouts in North America and Europe, and recommended means to improve system dynamic performance,* in IEEE Transactions on Power Systems, vol. 20, no. 4, IEEE, 2005, pp. 1922-1928, doi: 10.1109/TPWRS.2005.857942.

4. Leveson, N., and C. Turner. *An Investigation of the Therac-25 Accidents.* Computer (Long Beach, Calif.), vol. 26, no. 7, IEEE, 1993, pp. 18–41, doi:10.1109/MC.1993.274940.

5. Celko, J.. *Joe Celko's Sql for Smarties: Advanced Sql Programming / Joe Celko*. Fifth edition., Morgan Kaufmann, 2015, pp. 21-27.

6. Netzer, R., and B. Miller. *What Are Race Conditions? Some Issues and Formalizations.* ACM Letters on Programming Languages and Systems, vol. 1, no. 1, ACM, 1992, pp. 74–88, doi:10.1145/130616.130623.

7. Klein, P., et al. *Detecting Race Conditions in Parallel Programs That Use Semaphores.* Algorithmica, vol. 35, no. 4, Springer-Verlag, 2003, pp. 321–45, doi:10.1007/s00453-002-1004-3.

8. Sterling, N. *Warlock-A Static Data Race Analysis Tool.* USENIX Winter 1993 Conference, USENIX, 1993.

9. Engler, D., and K. Ashcraft. *RacerX: Effective, Static Detection of Race Conditions and Deadlocks.* Operating Systems Review, vol. 37, no. 5, ACM, 2003, pp. 237–52, doi:10.1145/1165389.945468.

10. Savage, S., et al. *Eraser: a Dynamic Data Race Detector for Multithreaded Programs.* ACM Transactions on Computer Systems, vol. 15, no. 4, ACM, 1997, pp. 391–411, doi:10.1145/265924.265927.

11. Yu, Y., et al. *RaceTrack : Efficient Detection of Data Race Conditions via Adaptive Tracking.* Operating Systems Review, vol. 39, no. 5, ACM, 2005, pp. 221–34, doi:10.1145/1095809.1095832.

12. Serebryany, K., and T. Iskhodzhanov. *ThreadSanitizer: Data Race Detection in Practice.* Proceedings of the Workshop on Binary Instrumentation and Applications, ACM, 2009, pp. 62–71, doi:10.1145/1791194.1791203.

13. Lamport, L.. *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, vol. 21, no. 7, ACM, 1978, pp. 558–65, doi:10.1145/359545.359563.

14. Yu, M., et al. *Efficient Noise Injection for Exposing Hidden Data Races.* The Journal of Supercomputing, vol. 76, no. 1, Springer Nature B.V, 2020, pp. 292–323, doi:10.1007/s11227-019-03031-0.

15. Sen, K.. *Race Directed Random Testing of Concurrent Programs.* Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2008, pp. 11–21, doi:10.1145/1375581.1375584.

16. Park, S., et al. *CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places.* Computer Architecture News, vol. 37, no. 1, 2009, pp. 25–36, doi:10.1145/2528521.1508249.

17. Erickson, J., et al. *Effective Data-Race Detection for the Kernel.* OSDI. Vol. 10. No. 10. 2010, pp. 1-16.

18. Li, G., et al. *Efficient Scalable Thread-Safety Violation Detection: Finding thousands of concurrency bugs during testing.* ACM SIGOPS SOSP '19, ACM, 2019, pp. 162-180, doi:10.1145/3341301.3359638.

19. Adve, S., and K. Gharachorloo. *Shared Memory Consistency Models: a Tutorial.* Computer (Long Beach, Calif.), vol. 29, no. 12, IEEE, 1996, pp. 66–76, doi:10.1109/2.546611.

20. Edwards, S.. *Languages for Digital Embedded Systems / Stephen A. Edwards.* Kluwer Academic Publishers, 2000.

21. Dagum, L., and R. Menon. *OpenMP: An Industry Standard API for Shared-Memory Programming.* IEEE Computational Science & Engineering, vol. 5, no. 1, IEEE, 1998, pp. 46–55, doi:10.1109/99.660313.

22. Hoganson, A.. *Initial SHM Racer Release*. GitHub Release, v1.0, doi: 10.5281/zenodo.4738421.