DYNAMIC DATA RACE DETECTION IN CONCURRENT PROGRAMS

by

Önder Kalacı

B.S., Computer Engineering, Middle East Technical University, 2011

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering
Boğaziçi University

2014

DYNAMIC DATA RACE DETECTION IN CONCURRENT PROGRAMS

APPROVED BY:

Assoc. Prof. Alper Şen (Thesis Supervisor)	.....
Prof. Can Özturan	.....
Assist. Prof. Barış Aktemur	.....

DATE OF APPROVAL: 21.05.2014

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Assoc. Prof. Alper Şen for his guidance, support and patience throughout the whole process of this work. His knowledge, vision and guidance helped me in all the time of research and writing of this thesis.

I thank my fellow friend Gökçehan Kara for the stimulating discussions and sharing his prior knowledge during my research. I also thank my friend Taylan Işıkdemir for his support and valuable friendship during the two years in Istanbul. I am grateful to my friend Osman Kaya for his encouragement. And many thanks to Aybike Avşaroğlu for her patience and support during the last three years.

Last but not the least, I would like to thank my parents Münevver and Nesip, my sister Gaye and twin brother Ender for their valuable support. I would like to dedicate my thesis to all of them.

ABSTRACT

DYNAMIC DATA RACE DETECTION IN CONCURRENT PROGRAMS

Recent advances in hardware drive software to be more concurrent than ever. Concurrency in software is achieved by multithreading, which creates verification challenges. These challenges include problems such as deadlocks, race conditions and atomicity violations, all of which are notoriously difficult to detect due to the non-deterministic nature of concurrent software. Data races result from the concurrent access of shared data by multiple threads and can result in unexpected program behaviors. In this thesis, we describe techniques to detect data races in multithreaded applications. We developed a hybrid algorithm that is a combination of the state-of-the-art happens-before and lockset data race detection algorithms. We take advantage of lockset algorithm and happens-before algorithm for discarding false negatives and false positives, respectively. Our algorithm works on the binary of the program (without the need for the source code), hence makes it applicable to industrially-deployed software. Since it is a dynamic technique, it has execution time and memory overhead. We utilized the concept of segments to decrease these overheads, where a segment is formed by consecutive memory accesses of a single thread. We performed experiments to validate the effectiveness of our hybrid race detector by comparing it with a happens-before and lockset-based race detector. Our experiments on several benchmarks showed that our hybrid detector is 20% faster than happens-before detector and produces 50% less potential data races than the lockset detector. We proposed four different optimizations to further decrease the execution time and enhance the usability.

ÖZET

KOŞUTZAMANLI PROGRAMLARDA DİNAMİK VERİ YARIŞI SAPTAMA

Yakın zamanda donanım alanında yaşanan gelişmeler, yazılımı her zamankinden daha koşutzamanlı olmaya yöneltti. Yazılımda koşutzamanlılığa çoklu iş parçacığı ile erişilir, bu doğrulamada zorluklar yaratır. Bu zorluklar koşutzamanlılığın gerekirci olmayan doğası nedeniyle saptanması zor problemler olan kaynak bekleme, veri yarışları ve bölünmezlik ihlallerini içerir. Veri yarışları paylaşılmış verilerin çoklu iş parçacıkları tarafından koşutzamanlı erişilmesi sonucuda ortaya çıkar ve beklenmedik program davranışlarına neden olabilir. Bu tezde çoklu iş parçacıklı uygulamalarda veri yarışlarını saptama tekniklerini tanımlayacağız. Önce-gerçekleşen ve kilit kümesi algoritmalarının kombinasyonu olan melez veri yarışı algoritmasını geliştirdik. Kilit kümesi algoritmasından yaralanarak yanlış eksileri ve önce-gerçekleşen algoritmasindan yararlanarak ise yanlış artıları attık. Algoritmamız uygulamanın ikili kodu üzerinde çalışır (kaynak koda ihtiyaç duymadan), bundan dolayı endüstride konuşlanmış yazılımlara uygulanabir. Dinamik veri yarışı saptama teknikleri uzun yürütüm süresi ve yüksek hafıza gereksiniminden muzdariptir. Bahsi geçen ek yükleri azaltmak için kesit konseptinden yararlandık, bir kesit bir iş parçacığı tarafından arka arkaya yapılan hafıza erişimlerinden oluşur. Melez veri yarışı detektörümüzün etkinliğinin geçerliliğini denetlemek için bir önce-gerçekleşen ve bir kilit kümesi detektörü ile karşılaştırarak deneyler yaptık. Çeşitli sabit noktalarla gerçeklestirdiğimiz deneyler gösterdi ki melez detektörümüz önce-gerçekleşen detektöre kıyasla %20 daha hızlıdır ve kilit kümesi detektöre kıyasla %50 daha az veri yarışı üretir. Kullanılırlığı ve performansı artıracak 4 farklı optimizasyon uyguladık.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$A \rightarrow B$	A Happens Before B
br	Identifier of a barrier
$C(a)$	Logical vector clock of event a
cv	Identifier of a condition variable
l	Identifier of a lock
t_i	ID of ith thread
x, y, z	Shared memory address identifiers

LIST OF ACRONYMS/ABBREVIATIONS

API	Application programming interface
BBL	Basic Block
DBI	Dynamic binary instrumentation
LS	Lockset
HB	Happens-Before
JIT	Just-In-Time
VC	Vector Clock

1. INTRODUCTION

1.1. Motivation

Until mid 2000's, hardware manufacturers met faster computation requirements by increasing the clock frequency of CPU's. However, when the manufacturers hit the physical limits, the solution they provided was to increase the number of computation units, namely cores [3]. The increased number of cores provides more computation power, however, in order to utilize the increased power, concurrent software must be designed and written.

Concurrent application development, more specifically its correctness and verification, is difficult due to different thread interleavings. These interleavings depend on the schedulers, thus each execution of an application may exhibit different behavior. Therefore, rerunning the application may not help find concurrency bugs. Some of the most important bugs that arise due to concurrency are deadlocks, data races and atomicity violations [4]. In this work, we focus on data races since they are observed quite often in large scale real life applications and fixing them requires substantial effort.

```
1: int global_counter = 0;
2: int increment()
3: {
4:   int tmp = global_counter;
5:   tmp++;
6:   global_counter = tmp;
7: }
```

Figure 1.1. Increment a counter once for each thread.

Thread 1	Thread 2
int tmp = global_counter;	
tmp++;	
global_counter = tmp;	
	int tmp = global_counter;
	tmp++;
	global_counter = tmp;

(a)

Thread 1	Thread 2
int tmp = global_counter;	
tmp++;	
	int tmp = global_counter;
	tmp++;
global_counter = tmp;	global_counter = tmp;

(b)

Figure 1.2. Two different thread interleavings of Figure 1.1, (a) first interleaving, (b) second interleaving.

Data race is defined as two memory accesses on a single memory address where at least one access is write and there is no appropriate synchronization among the accesses [5]. Consider a simple example of data race in Figure 1.1, where an integer counter is expected to be incremented once by each thread. However, as shown in Figure 1.2, for two different thread interleavings the value of the counter may be different at the end of the execution. In Figure 1.2a, the global counter is two at the end of the execution, which is expected since two threads increments the counter once. Now consider Figure 1.2b, though the expected value of the counter is two, its value is one at the end of the execution. As this simple example illustrates, data races may effect the result of the execution and may lead to inconsistencies.

1.2. Data Race Detection Techniques

Data races are one of the most common bugs experienced while multithreaded software is being developed. Detecting data races and fixing them increases quality of the software. There are many proposed solutions regarding to data race detection, which are categorized in two fundamental approaches: static data race detection and dynamic data race detection. In this section, we discuss both approaches.

1.2.1. Static Data Race Detection

Static data race detection tools get source code as an input and decide whether the source code includes possible data races or not.

There are some important points that need to be explored related to static data race detection. Firstly, static data race detection tools have problems with dynamically allocated data, since memory locations allocated dynamically are determined on the fly and static tools have no information about the run time. Secondly, detecting all possible data races by static analysis is known to be an NP-hard problem [5]. Therefore, many of the static data race detection tools focus on finding a subset of possible data races. Another issue with the static detection is that all possible thread execution paths and thread interleavings must be explored which may require too much memory [6]. Lastly, static analysis examines interleavings that may not be possible during execution of the program as they are not aware of the dynamic context of the program.

There are many proposed static data race detection algorithms such as RELAY [7], LOCKSMITH [8]. These algorithms statically implement Eraser [1], a lockset data race detection technique which is explained in Chapter 3. Basically, these tools check which code is executed in parallel and which lock is intended for which shared variable and try to find potential data races.

Compared to dynamic data race detection, static detection techniques produce more false positives. The main reason is that all possible thread interleavings are generated and potential data race is searched among all these interleavings. However, some of the interleavings are not possible to happen with any input data. Although some of the generated interleavings may happen with some specific input data, they are not likely to happen due to actual operating system thread schedulers. In contrast to false positives, generating all possible interleavings reduces the total number of false negatives, which means that static detectors miss less number of potential data races.

1.2.2. Dynamic Data Race Detection

Dynamic analysis tools, contrary to static analysis tools, analyse source code while it is running. Dynamic analysis is done via instrumentation. The term *Instrumentation* is widely used to indicate adding some extra code to programs. Basically, Dynamic Binary Instrumentation (DBI) tools are able to instrument binaries on run time which enables to perform various analyses of program execution. They provide ability to instrument and monitor every memory access. Dynamic data race detection tools track these memory access patterns in order to find potential data races. There are several DBI tools such as PIN [2], DynamoRIO [9] and Valgrind [10]. In our implementation, we utilized PIN. It is developed by Intel and allows to instrument executables which are written in C/C++. The deficiency of dynamic tools is that they consider an execution with a single input data, which limits the coverage of the analysis. In order to overcome this problem, dynamic analysis must be repeatedly executed with different input data. Another problem with the dynamic analysis is that, it leads to considerable run time overhead. On the other hand, an advantage of dynamic analysis is that the application that is analyzed needs no pre-processing such as compiling or linking. Thus, dynamic analysis does not require source, recompilation or relinking.

Post-mortem analysis, a variation of dynamic analysis, records some important events during the execution, then analyses the recorded information after execution finishes. Main application area of post-mortem analysis is record/replay type analysis such as [11, 12]. The non-deterministic nature of multithreaded applications makes post-mortem analysis a suitable analysis tool for them. The bugs that are encountered in a single execution can be tracked and solved. Post-mortem analysis also suffers from the coverage of the analysis problem. Another problem with post-mortem analysis is that it is not suitable for long running applications since the size of recorded data may become huge and unprocessable.

Static and dynamic analysis tools are considered to be complementary. While static analysis tools are considered to be sound, since they consider all execution paths of a source code, dynamic tools are considered to be unsound due to their limited

coverage. However, when precision is considered, dynamic tools are more precise compared to static tools, since dynamic tools analyse an application with realistic input data [13].

1.3. Contribution

The contribution of this work consists of two main parts. First, we developed a dynamic data race detection framework and implemented three different data race detection algorithms on PIN [2]. The algorithms are lockset algorithm [1], happens-before algorithm [14] and segment-based hybrid algorithm [15]. We performed experiments using benchmark applications, analyzed their results and compared the algorithms in detail. Secondly, we thoroughly examined segment-based hybrid algorithm. We discuss the reasoning behind the algorithm and proposed four different optimizations. Two of these optimizations increases the performance of data race detection without sacrificing the precision. The remaining two optimizations provide a trade-off between the number of potential data races detected and the performance of data race detection.

2. BACKGROUND

In this chapter, we introduce an execution model of multithreaded programs. Then, in the next chapter we utilize the program execution model and define dynamic data race detection algorithms.

2.1. Program Execution Model

A multithreaded program consists of threads, memory addresses, locks, barriers, condition variables and semaphores. During the execution of a program, a sequence of operations is generated by instrumenting the program. These operations are described in the next section using the following notations that form a multithreaded program:

- t_i**:** represents ith thread in the execution.
- x**:** represents a shared memory address.
- l**:** represents a lock in the program execution. Each lock is identified with a unique address in the execution.
- cv**:** represents a condition variable in the program execution. Each condition variable is identified with a unique address in the execution.
- br**:** represents a barrier in the program execution. Each barrier is identified with a unique address in the execution.
- s**:** represents a semaphore in the program execution. Each semaphore is identified with a unique address in the execution.
- $thread(e)$**:** is thread id of the thread that executed event e. Events are described below.

2.2. Operations Generated During the Execution of a Multithreaded Program

In this work, we consider multithreaded C/C++ applications that use the pthread library [16]. All the operations described below are atomic. Note that *operation* and

event are used interchangeably in the rest of the document.

- $READ(x, t_i)$**:** Memory address x is read by thread t_i. An example of this event is:
  (i) $y = x + 1$
- $WRITE(x, t_i)$**:** Memory address x is written by thread t_i. An example of this event is:
  (i) $x = y + 1$
- $ACCESS(x, t_i) = READ(x, t_i) \vee WRITE(x, t_i)$: Memory address x is read or written by thread t_i
- $WR\_LOCK(l, t_i)$**:** Lock l is acquired write-held by thread t_i. Examples of this event are:
  (i) *pthread_wrlock_wrlock*, *pthread_mutex_lock*
- $RD\_LOCK(l, t_i)$**:** Lock l is acquired read-held by thread t_i. Examples of this event are:
  (i) *pthread_wrlock_rdlock*
- $LOCK(l, t_i)$**:** Lock l is acquired write-held or read-held by thread t_i. Examples of this event are:
  (i) *pthread_wrlock_rdlock*, *pthread_wrlock_wrlock*, *pthread_mutex_lock*
- $UNLOCK(l, t_i)$**:** Lock l is released by thread t_i. Examples of this event are:
  (i) *pthread_wrlock_unlock*, *pthread_mutex_unlock*
- $SIGNAL(cv, t_i)$**:** Condition variable cv is notified by thread t_i. An example of this event is:
  (i) *pthread_cond_signal*
- $SIGNAL\_ALL(cv, t_i)$**:** Condition variable cv is notified all by thread t_i. An example of this event is:
  (i) *pthread_cond_broadcast*
- $WAIT(cv, t_i)$**:** Thread t_i waits on condition variable cv. An example of this event is:
  (i) *pthread_wait_signal*
- $SEM\_POST(s, t_i)$**:** Semaphore s is posted by t_i. An example of this event is:
  (i) *sem_post*

- $SEM\_WAIT(s, t_i)$**:** Thread t_i waits on semaphore variable s. An example of this event is:
  (i) *sem_wait*
- $BARRIER\_INIT(br, n)$**:** Barrier br is initialized to n. An example of this event is:
  (i) *pthread_barrier_init*
- $BARRIER\_WAIT(br, t_i)$**:** Thread t_i waits on the barrier br. An example of this event is:
  (i) *pthread_barrier_wait*
- $THREAD\_CREATE(t_i, t_j)$**:** Thread t_i creates thread t_j. An example of this event is:
  (i) *pthread_create*
- $THREAD\_JOIN(t_i, t_j)$**:** Thread t_i joins to thread t_j. An example of this event is:
  (i) *pthread_join*
- There are some event pairs that guarantee happens-before relation in the execution. For ease of discussion, we represent this kind of event pairs with $SYNC\_POINT\_START$ and $SYNC\_POINT\_END$ events. The events before the $SYNC\_POINT\_START$ are executed before the events executed after $SYNC\_POINT\_END$. Examples of these event pairs shown below. For each pair $SYNC\_POINT\_START$ and $SYNC\_POINT\_END$ events are listed respectively. Note that, for different algorithms other pairs of events may form happens-before relation.
  (i) $SIGNAL(cv, t_i), WAIT(cv, t_j)$
  (ii) $SIGNAL\_ALL(cv, t_i), WAIT(cv, t_y)$
  (iii) $THREAD\_CREATE(t_i, t_j), THREAD\_JOIN(t_j, t_i)$

Some pthread function calls can be represented with two or more operations. For instance, consider *pthread_mutex_lock*, this pthread call could be represented either $LOCK$ or WR_LOCK operations. The reason is that some algorithms differentiate between writer locks and reader locks whereas others do not.

2.3. Background on Data Race Detectors

All of the operations defined in Section 2.2 are direct representation of pthread and standard library functions. However, in order to define data race detection algorithms with multithreaded execution trace, some more operations should be defined. In this section we define these operations and related subjects.

2.3.1. Happens Before Relation

Happens-before relation is defined as a partial order on operations happening in a distributed system [17]. In this work, this relation is extended to threads running on a single machine. Happens-before relation ($\rightarrow$) among two events e_x and e_y must satisfy the following properties (where $x \neq y$ and $i \neq j$):

(i) Single Thread Execution: Two events e_x and e_y are executed by the same thread and e_x is executed by e_y in program order.
- $(e_x \in t_i \wedge e_y \in t_i) \wedge (e_x \ is \ executed \ before \ e_y) \Rightarrow (e_x \rightarrow e_y)$

(ii) Synchronization Points:
- $e_x = SYNC\_POINT\_START(t_i, t_j) \wedge e_y = SYNC\_POINT\_END(t_i, t_j) \Rightarrow e_x \rightarrow e_y$

(iii) Transitive Closure:
- $(e_x \rightarrow e_y) \wedge (e_y \rightarrow e_z) \Rightarrow (e_x \rightarrow e_z)$

Definition of *concurrent events* are strictly related to happens-before relation. Two events, e_x and e_y are concurrent if neither of them happens-before each other.

- $e_x || e_y \Leftrightarrow (\neg(e_x \rightarrow e_y) \wedge \neg(e_y \rightarrow e_x))$

2.3.2. Vector Clock

Dynamic data race detectors utilize vector clocks to generate happens-before relation among events during program execution. In our definition, happens-before

relation ($\rightarrow$) defines a relation among two events indicating which event is executed before the other. A *vector clock* VC consists of vector of N integers where N is the total number of threads in the execution. VC_i identifies the vector clock of thread t_i and $VC_i[j]$ holds the logical time that thread t_i has learned about thread t_j during the execution. Vector clock operations corresponding to some events may differ with respect to data race detection algorithm. The vector clock operations that are specific to our algorithms are described in detail in Chapter 3.

(i) Initialize a Vector Clock:

$$INIT(VC_i) = \begin{cases} VC_i[n] = 1 & i = k \\ VC_i[n] = 0 & i \neq k \end{cases}$$

(ii) Increment a Vector Clock:

$$INCREMENT(VC_i) : VC_i[i] = VC_i[i] + 1;$$

(iii) Receive Vector Clock:

$$RECV(t_j, t_i) : VC_j[k] = max(VC_j[k], VC_i[k]) \quad for \quad k = 1, \ldots, N$$

(iv) Compare Two Vector Clocks:

For defining happens-before relation among events, it is necessary to define the comparison operation among vector clocks.

$$VC_i < VC_j = \begin{cases} 1 & VC_i[k] \leq VC_j[k] \ \ for \text{ k=1,...,N } \ and \ \exists k \ st \ VC_i[k] < VC_j[k] \\ 0 & else \end{cases}$$

2.3.3. LockSet

Locksets are utilized by lockset or hybrid dynamic data race detection algorithms for detecting data races. *Locksets* keep track of the locks that threads acquire/release. In other words, locksets contain the locks that are currently held by the thread. Some algorithms or implementations differentiate between writer and reader locks which is described in Chapter 3. We denote a lockset by LS.

- Initially, for any thread t_i, its lockset is empty $LS(t_i) := \{\}$.
- When t_i acquires a lock l, using the operation $LOCK(l, t_i)$ then: $LS(t_i) := LS(t_i) \cup \{l\}$.
- When t_i releases a lock l, using the operation $UNLOCK(l, t_i)$ then: $LS(t_i) := LS(t_i) - \{l\}$.

2.3.4. Trace

Traces are sequences of events executed by a single thread t_i. A trace consists of "Single Enter-Multiple Exit" block of code. Consider Figure 2.1, at some point in $dummy\_function$, there exists an $if \ldots else$ code block. The code block is entered from a single point(e_start) and can be exited from three different points ($e\_end\_0, e\_end\_1, e\_end\_2$), which is an example of a trace.

Similar to [18], we utilize vector clocks for representing the partial ordering and determining parallelism among traces. Moreover, we use locksets for keeping track of the locks held. Now, we formally define a trace.

- $prev(e_i) = e_j$. Returns the event that is previously executed by the same thread as e_i. In other words, e_j and e_i executed by the same thread consecutively.
- e_start = An event that is single entrance point.
- e_end = An event that is the last of the possible exit points executed where e_start is the entrance point.

```
string dummy_function(int x)
{
 ...
 //single enterence point (e_start)
 if (x < 5)
  return "x is smaller than 5"; //First possible exist point (e_end_0)
 else if (x > 5)
  return "x is greater than 5"; //Second possible exist point (e_end_1)
 else
  return "x equals to 5";       //Third possible exist point (e_end_2)
 ...
}
```

Figure 2.1. Trace example.

$$
\begin{aligned}
trace = \{e_1, \ldots, e_n | \\
& \forall i \in 1 \ldots n \\
& \wedge (e_i = e_1 \vee prev(e_i) = e_{i-1}) \\
& \wedge (e_i = ACCESS(x, thread(e_i)) \\
& \wedge thread(e_i) = thread(e_1) \\
& \wedge e_i.VC = e_1.VC \\
& \wedge e_i.LS = e_1.LS \\
& \wedge e_1 = e\_start \\
& \wedge e_n = e\_end\}
\end{aligned}
$$

Some properties of traces are defined as follows:

(i) $thread(trace) = thread(e_i)$, where $e_i \in trace$. In other words, $thread(trace)$ returns the thread id that the trace is generated by.

(ii) $VC(trace) = VC_i$, where $i = thread(trace)$. Note that, vector clock of a trace is stable.

(iii) $LS(trace) = LS(t_i)$, where $i = thread(trace)$. Note that, lockset of a trace is stable.

(iv) PREV($trace_i$) = $trace_j$, returns the trace that is previously generated by the same thread. In other words, $trace_i$ is the next trace that is generated by the same thread that generated $trace_j$.

2.3.5. Segment

Segment is defined as an extended version of trace. The distinction between segment and trace is that a segment may be constructed by one or more consecutive traces provided that no synchronization events happen among the traces. Definition of segment is as follows:

$$
\begin{aligned}
segment = \{&trace_1, \ldots, trace_n | \\
&((trace_i = trace_1) \vee (PREV(trace_i) = trace_{i-1})) \\
&\wedge (THREAD(trace_i) = THREAD(trace_1)) \\
&\wedge (VC(trace_i) = VC(trace_1)) \\
&\wedge (LS(trace_i) = LS(trace_1))\}
\end{aligned}
$$

3. DATA RACE DETECTION ALGORITHMS

In this chapter, we firstly describe state-of-the-art algorithms used in dynamic data race detection, happens-before and lockset data race detection algorithms. Then, we describe our segment-based hybrid algorithm.

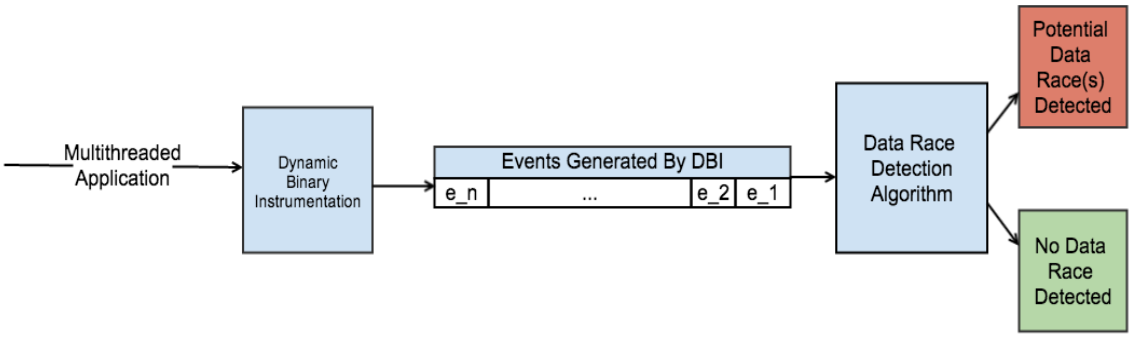

Figure 3.1. Overview of dynamic data race detection.

In Figure 3.1, an overview of dynamic data race detection procedure is displayed. A dynamic binary instrumentation tool instruments a multithreaded application and generates events, which are input to the data race detection algorithm. Lastly, the algorithm decides whether the application has potential data race(s) or not.

3.1. Lockset Data Race Detection Algorithm

3.1.1. Introduction

Lockset algorithm checks whether two threads access a shared variable while holding a common lock or not. If there is a common lock, the algorithm concludes that the lock is intended for protecting the variable.

3.1.2. Algorithm

Lockset algorithm is proposed and implemented in Eraser [1] for the first time. Tools that implement the algorithm are called pure lockset data race detectors. For each shared memory address, the algorithm keeps a candidate lockset. The name *candidate* is given since the algorithm cannot determine which lock is intended for which memory address(es). Thus, via candidate locksets, the algorithm attempts to infer whether a shared memory is protected by a unique lock throughout the execution. When a memory address is accessed for the first time during the execution, its candidate lockset is assigned to include all the locks that has appeared during the execution history up to that time. Then, on each access its candidate lockset is updated to its intersection with the thread's lockset that is executing the access. This lock refinement step aims to find the unique locks which protect the variable during the execution. If the intersection ends up with an empty set, the algorithm concludes that there exist no locks for protecting the variable, thus, there is a potential race. More formally:

- $CLS(x)$ = Candidate lockset for memory address x.
- $all\_locks = \{l_x | \forall x \ (\forall i \ (LOCK(l_x, t_i))\}$.
- On first access to memory address x, $CLS(x) = all\_locks$.

Input: Thread t_i Generates A Memory Access Operation On Memory Address x

Output: Potential Data Race Detected Or Not

1: $CLS(x) := CLS(x) \cap LS(t_i)$
2: **if** $CLS(x) == \{\}$ **then**
3: return true; ▷ Race Found
4: **end if**
5: return false; ▷ No Race Found

Figure 3.2. Lockset-based data race detection algorithm.

In Figure 3.3, an execution of lockset algorithm is displayed. Assume that time increases downwards in the figures that show execution of an algorithm, including this figure.

Thread 1	Thread 2	CLS(x)	LS(t_1)	LS(t_2)
LOCK(l_1, t_1)		{}	{l_1}	{}
WRITE(x, t_1)		{l_1}	{l_1}	{}
UNLOCK(l_1, t_1)		{l_1}	{}	{}
	LOCK(l_2, t_2)	{l_1}	{}	{l_2}
	WRITE(x, t_2)	{} **Race Found!!**	{}	{l_2}
	UNLOCK(l_2, t_2)	{}	{}	{}

Figure 3.3. Execution of lockset algorithm.

On each memory access, the algorithm in Figure 3.2 is executed. Firstly, Thread 1 acquires lock l_1 and executes a write access to variable x. After the write is executed, the candidate lockset of x, $CLS(x)$, includes only l_1. Then, Thread 1 releases the lock and $LS(t_1)$ becomes empty. Thread 2 acquires lock l_2 and executes a write access to variable x. After the access is executed, $CLS(x)$ becomes empty, which indicates a data race. The potential data race condition is detected since the variable is accessed by two threads without a common lock.

3.1.3. Improvements on the Algorithm

3.1.3.1. Reader Locks - Writer Locks. Most of the thread libraries, including the Pthread library [16], allows both reader and writer locks. Basically,

- *pthread_rwlock_rdlock*: The calling thread acquires the lock if a writer does not hold the lock and no writers are blocked on the lock.
- *pthread_rwlock_wrlock*: The calling thread acquires the write lock if no other reader thread or writer thread holds the lock.

When, WR_LOCK and RD_LOCK are considered together, the unmodified algorithm may miss some real races. An improvement on the algorithm is displayed in Figure 3.4. Consider Figure 3.5, where two threads acquire the read lock, and write to a shared variable. The caller thread of RD_LOCK gets the lock and continues its execution if there are no writers waiting on the same lock. Thus, for the correctness of any execution, when a thread acquires a read lock, it is expected to execute read accesses. However, though the execution in Figure 3.5 could be considered as a bad style of programming, it is expected that the algorithm detects the potential race on variable x.

Input: Thread t_i Generates A Memory Access Operation On Memory Address x

Output: Potential Data Race Detected Or Not

1: **if** $ACCESS(x, t_i) == READ(x, t_i)$ **then**
2: $\quad CLS(x) := CLS(x) \cap (WRLS(t_i) \cup RDLS(t_i))$
3: **else if** $ACCESS(x, t_i) == WRITE(x, t_i)$ **then**
4: $\quad CLS(x) := CLS(x) \cap WRLS(t_i)$
5: **end if**
6: **if** $CLS(x) == \{\}$ **then**
7: $\quad$ return true; ▷ Race Found
8: **end if**
9: return false; ▷ No Race Found

Figure 3.4. Lockset-based data race detection algorithm - reader/writer locks.

This problem is solved by making a small modification in the algorithm given in Figure 3.2. For each thread, two locksets are maintained, one for read locks and one for write locks. Then, if the access is a read access, the candidate lockset is intersected with all locks acquired by the thread, as the algorithm behaves in its basic form. However, when the access is a write access, the candidate lockset is intersected with only write locks. Thus, the algorithm detects races when a thread makes a write access by holding only read locks. Applying the improved algorithm, the false negative result on the execution in Figure 3.5 is turned into a true positive result.

Thread 1	Thread 2	CLS(x)	LS(t_1)	LS(t_2)
RD_LOCK(l_1, t_1)		{}	{l_1}	{}
WRITE(x, t_1)		{l_1}	{l_1}	{}
RD_LOCK(l_1, t_1)		{l_1}	{}	{}
	RD_LOCK(l_1, t_2)	{l_1}	{}	{l_1}
	WRITE(x, t_2)	{l_1} **Missed!!**	{}	{l_1}
	RD_LOCK(l_1, t_2)	{l_1}	{}	{}

Figure 3.5. Execution of lockset algorithm - read/write locks.

3.1.3.2. Read Shared Variables. Some variables are written only when they are created, then throughout the execution only read by other threads. These variables are called *Read-Shared Variables* and these variables are not in danger of data race because they are merely concurrently read. The algorithm in Figure 3.2 cannot handle read-shared variables. In Figure 3.6, the read accesses of Thread 2 and Thread 3 will be reported as data race using the algorithm in Figure 3.2. The reason is that on both read accesses, Thread 1 and Thread 2 do not hold any locks. Thus, intersecting candidate lockset of x with thread's lockset always yields empty set, which indicates a data race.

Thread 1	Thread 2	Thread 3
WRITE(x, t_1)	-	-
THREAD_CREATE(t_1, t_2)	-	-
THREAD_CREATE(t_1, t_3)	READ(x, t_2)	-
		READ(x, t_3)

Figure 3.6. Execution of lockset algorithm - read shared.

To handle this problem, the improved algorithm assigns a state information for each variable. There are 4 possible states, and below these variable states are explained. Note that the algorithm in Figure 3.2 cannot decide whether the variable will be shared

during the execution or not. Thus, it starts considering a variable only when it is accessed by a second thread and this observation is reflected on the four states.

- Virgin: When a variable is first allocated, its state is assigned to be virgin.
- Exclusive: A variable is in exclusive state if it is accessed by only a single thread. In this state, the candidate lockset of the variable is not updated and race detection is not executed for the variable.
- Shared: If only read-sharing happens for a variable, it is considered to be in shared state. In this state, algorithm keeps track of the candidate lockset, however, does not inform about the potential data races found.
- Shared-Modified: The state becomes shared-modified when a variable is written by a second thread. In this state, the algorithm informs about data races.

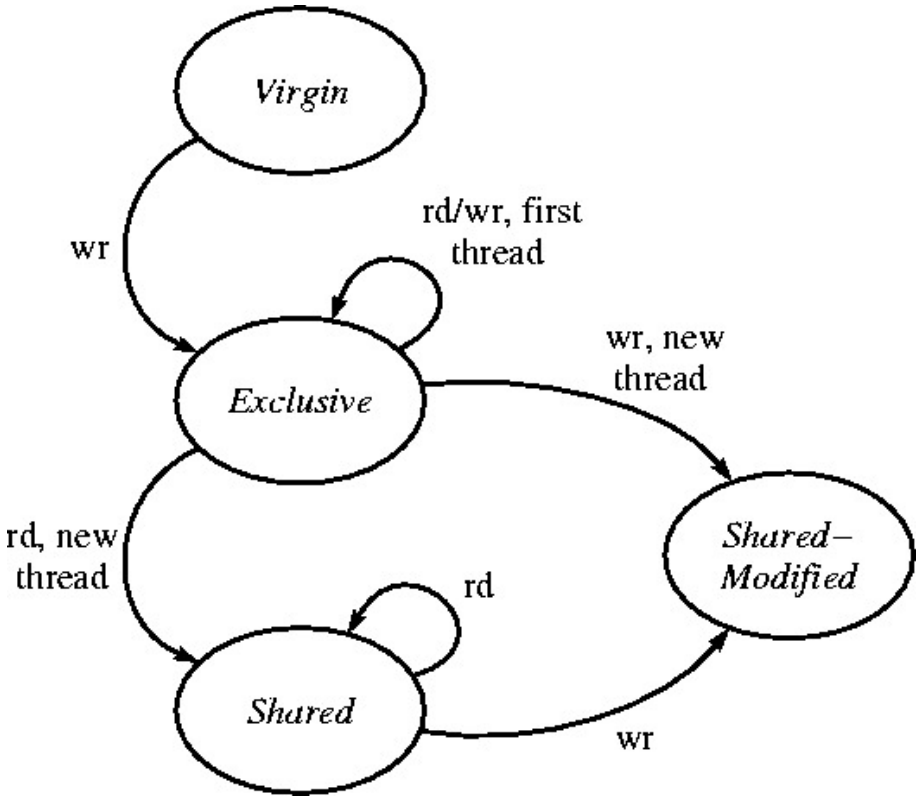

Figure 3.7. Eraser state transition for variables [1].

State transitions for variables are shown in Figure 3.7. Via the state transitions, the false positive result on the execution in Figure 3.6 is turned into a true negative

result. Note that, the state transitions may lead the algorithm to miss some potential data races. Assume that a variable is being initialized with more than one write accesses of a thread. During the initialization, the state of the variable is *Exclusive*, since only a single thread has accessed it. If the variable is written by another thread when the initialization has not been finished yet, the algorithm misses the potential race.

3.1.4. Discussion on Lockset Data Race Detection

Lockset data race detectors are vulnerable to false positives. The source of false positives is explained as follows. Programmers code in such a way that there is no common lock held while accessing the variable, however, there is an implicit or explicit "happens-before" relation among the accesses. That is, the coder is absolutely sure that one access is going to happen-before the other. However, lockset detector ignores the happens-before relation due to operations such as $SIGNAL$ and $WAIT$.

Thread 1	Thread 2
WRITE(x, t_1)	
SIGNAL(cv, t_1)	
	WAIT(cv, t_2)
	WRITE(x, t_2)

Figure 3.8. Lockset-based detectors produce warning.

Consider Figure 3.8, lockset detectors produce warning for the execution. However, there is no potential data race since write accesses are ordered by happens-before relation. In general, the detectors produce false alarms for each of the shared memory address in concurrent applications that the synchronization among threads are generated by only condition variables, where condition variables are used for generating happens-before relation.

3.2. Happens-Before Data Race Detection Algorithm

3.2.1. Introduction

Happens-before data race detection algorithms are based on Lamport's happens-before relation described in Chapter 3. Tools that implement these algorithms are called pure happens-before data race detectors. Happens-before relation defines a partial order among the events generated during the execution of a distributed system. This relation could be extended for applications using shared memory as well. Happens-before data race detection algorithms, such as $DJIT^+$ [14] and *LiteRace* [19] utilize vector clocks for maintaining the happens-before relation. The advantage of using happens-before relation for data race detection is that it does not produce false positives. Whenever a happens-before race detector produces a warning, there is an alternative thread schedule that two accesses may happen simultaneously [20]. The disadvantage of happens-before detectors is that they may miss real races. In other words, they may generate false negatives.

3.2.2. Algorithm

In this section $DJIT^+$ algorithm is described as the happens-before algorithm. There are also other implementations of the happens-before race detection in the literature such as TRaDe [21]. The algorithm utilizes vector clock operations, which are are defined in Chapter 3.

- For each thread t_i:
  - (i) Maintain one vector clock, VC_i.
  - (ii) On startup, initialize $VC_i : INIT(VC_i)$.
- For each synchronization object s:
  - (i) Maintain one vector clock $s.VC$.
  - (ii) Initialize $s.VC$:

    $s.VC := [0 \ldots 0]$
  - (iii) On each $UNLOCK(s, t_i)$ or $SIGNAL(s, t_i)$ or $SEM\_POST(s, t_i)$:

Update $s.VC$: $RECV(s.VC, VC_i)$

Increment VC_i : $INCREMENT(VC_i)$

(iv) On each $LOCK(s, t_i)$ or $WAIT(s, t_i)$ or $SEM\_WAIT(s, t_i)$:

Update VC_i :$RECV(VC_i, s.VC)$

- For each memory address x:
  - (i) Keep two vector clocks, $x_r.VC$ and $x_w.VC$.
  - (ii) $x_r.VC(t_i)$ records the clock of the last read x by thread t_i.
  - (iii) $x_w.VC(t_i)$ records the clock of the last write x by thread t_i.
  - (iv) On each read on variable x by thread t_i check for data race. Check if the read happens after the last writes of all threads.
  - (v) On each write on variable x by thread t_i check for data race. Check if the write happens after the last writes of all threads and the last reads of all threads.

Input: Thread t_i Generates A Memory Access Operation On Memory Address x

Output: Potential Data Race Detected Or Not

```
1: if ACCESS(x, t_i) == READ(x, t_i) then
2:     if x_w.VC < VC_i then
3:         return false;                    ▷ No Race, Happens Before Ordered
4:     end if
5: else if ACCESS(x, t_i) == WRITE(x, t_i) then
6:     if (x_w.VC < VC_i) ∧ (x_r.VC < VC_i) then
7:         return false;                    ▷ No Race, Happens Before Ordered
8:     end if
9: end if
10: return true;                            ▷ Race Found
```

Figure 3.9. Happens-Before based data race detection algorithm.

In Figure 3.10, an execution of happens-before algorithm is displayed. On each memory access the algorithm in Figure 3.9 is executed. Below, all operations are examined in detail.

(i) Line 0:

Initialize all the necessary vector clocks.

- $INIT(VC_i)$
- $l.VC := [0 \dots 0]$

(ii) Line 1:

VC_1 is updated via lock l's vector clock since t_1 acquires l.

- $RECV(VC_1, l.VC)$

(iii) Line 2:

Thread 1 writes to variable x, thus execute the algorithm in Figure 3.9. Firstly, check for the race, then since there is no race, update $x_w.VC$'s $0th$ element. ***No Race, True Negative.***

- $(x_w.VC < VC_1) => True$
- $(x_r.VC < VC_1) => True$
- $x_w.VC(0) := VC_1(0)$

(iv) Line 3:

Thread 1 releases lock l. Thus, firstly update l's vector clock via VC_1, then increment VC_1 since it is an internal event.

- $RECV(l.VC, VC_1)$
- $INCREMENT(VC_1)$

(v) Line 4:

VC_2 is updated via lock l's vector clock since t_2 acquires l.

- $RECV(VC_2, l.VC)$

(vi) Line 5:

Thread 2 writes to variable x, thus execute the algorithm in Figure 3.9. Firstly, check for the race, then since there is no race, update $x_w.VC$'s $1th$ element. ***No Race, True Negative.***

- $(x_w.VC < VC_2) => True$
- $(x_r.VC < VC_2) => True$
- $x_w.VC(1) := VC_1(1)$

(vii) Line 6:

Thread 2 releases lock l. Thus, firstly update l's vector clock via VC_2, then

increment VC_2 since it is an internal event.

- $RECV(l.VC, VC_2)$
- $INCREMENT(VC_2)$

Line	Thread 1	Thread 2	VC_1	VC_2	$l.VC$	$x_w.VC$	$x_r.VC$
0			$<1,0>$	$<0,1>$	$<0,0>$	$<0,0>$	$<0,0>$
1	LOCK(l,t_1)		$<1,0>$	$<0,1>$	$<0,0>$	$<0,0>$	$<0,0>$
2	WRITE(x,t_1)		$<1,0>$	$<0,1>$	$<0,0>$	$<1,0>$	$<0,0>$
3	UNLOCK(l,t_1)		$<2,0>$	$<0,1>$	$<1,0>$	$<1,0>$	$<0,0>$
4		LOCK(l,t_2)	$<2,0>$	$<1,1>$	$<1,0>$	$<1,0>$	$<0,0>$
5		WRITE(x,t_2)	$<2,0>$	$<1,1>$	$<1,0>$	$<1,1>$	$<0,0>$
6		UNLOCK(l,t_2)	$<2,0>$	$<1,2>$	$<1,1>$	$<1,1>$	$<0,0>$

Figure 3.10. Execution of happens-before algorithm - vector clocks.

3.2.3. Discussion on Happens Before Data Race Detection

Happens-before detectors inspect data races by verifying the happens-before relation among memory accesses. The happens-before relation is represented by vector clocks in many happens-before based detectors such as $DJIT+$ [14] and *LiteRace* [19]. Contrary to lockset detectors, happens-before detectors do not produce any false positives.

Traditional happens-before detectors suffer from high execution time and memory overhead. Size of vector clocks are proportional to the thread count in the system. Therefore, memory requirements and comparison time of vector clocks are proportional to the thread count. In order to overcome these performance issues some algorithms are proposed. *FastTrack* [22] implements a scalar data structure *epoch*, consisting of two integers, and replaces vector clocks with epochs whenever possible. This replacement does not affect the precision of happens-before algorithm. *PACER* [23] proposes a sampling method on FastTrack algorithm. PACER makes a proportionality guarantee such that it detects potential data races at a rate equal to the sampling rate. *LiteRace* [19] is another happens-before data race detection algorithm. By applying another sampling method, the algorithm detects 70% of data races only sampling 2% of memory

accesses. In *Sound Predictive Race Detection* [24], they propose a new relation called "Casually Precedes" such that the relation is a more generalized relation than happens-before relation. This new relation does not scarify from the precision of happens-before relation, instead, it enables the detector to produce less false negatives.

The disadvantage of happens-before detectors is that they may miss real races. Consider two threads in Figure 3.11. The two different thread interleaving in Figure 3.12a and Figure 3.12b lead the algorithm to yield different results. In the first interleaving, initially Thread 1 starts execution, when it finishes Thread 2 starts its execution. In Figure 3.12b, the execution order of threads is reversed. Though there is a potential data race on variable x, it is missed by the algorithm in Figure 3.12a since the algorithm results in a happens-before relation due to the unlock/lock operations, which are meant to protect variable y.

```
void thread_1()
{
        x = 1;
        pthread_mutex_lock(&lock);
        y += 1;
        pthread_mutex_unlock(&lock);
}
```

(a)

```
void thread_2()
{
        pthread_mutex_lock(&lock);
        y += 1;
        pthread_mutex_unlock(&lock);
        x = 2;
}
```

(b)

Figure 3.11. Happens-Before algorithm misses potential data races. (a) First thread, (b) second thread.

In other words, happens-before based detectors may miss real races mostly due to the transitivity of happens-before relation. For instance, in the specific example displayed in Figure 3.12a, lock l is intended for protecting the shared variable y. However, UNLOCK(l, t_1) and UNLOCK(l, t_2) operations pair result in generating happens-before relation among the two threads. Thus, the accesses on the shared memory address x becomes happens-before ordered and the potential data race is missed by the detector due to an unrelated happens-before relation.

Line	Thread 1	Thread 2	VC_1	VC_2	$l.VC$	$x_w.VC$	$y_w.VC$
0			$< 1, 0 >$	$< 0, 1 >$	$< 0, 0 >$	$< 0, 0 >$	$< 0, 0 >$
1	WRITE(x, t_1)		$< 1, 0 >$	$< 0, 1 >$	$< 0, 0 >$	$< 1, 0 >$	$< 0, 0 >$
2	LOCK(l, t_1)		$< 1, 0 >$	$< 0, 1 >$	$< 0, 0 >$	$< 1, 0 >$	$< 0, 0 >$
3	WRITE(y, t_1)		$< 1, 0 >$	$< 0, 1 >$	$< 0, 0 >$	$< 1, 0 >$	$< 1, 0 >$
4	UNLOCK(l, t_1)		$< 2, 0 >$	$< 0, 1 >$	$< 1, 0 >$	$< 1, 0 >$	$< 1, 0 >$
5		LOCK(l, t_2)	$< 2, 0 >$	$< 1, 1 >$	$< 1, 0 >$	$< 1, 0 >$	$< 1, 0 >$
6		WRITE(y, t_2)	$< 2, 0 >$	$< 1, 1 >$	$< 1, 0 >$	$< 1, 0 >$	$< 1, 1 >$
7		UNLOCK(l, t_2)	$< 2, 0 >$	$< 1, 2 >$	$< 1, 1 >$	$< 1, 0 >$	$< 1, 1 >$
8		WRITE(x, t_2)	$< 2, 0 >$	$< 1, 2 >$	$< 1, 1 >$	$< 1, 2 >$	$< 1, 1 >$

(a)

Line	Thread 1	Thread 2	VC_1	VC_2	$l.VC$	$x_w.VC$	$y_w.VC$
0			$< 1, 0 >$	$< 0, 1 >$	$< 0, 0 >$	$< 0, 0 >$	$< 0, 0 >$
1		LOCK(l, t_2)	$< 1, 0 >$	$< 0, 1 >$	$< 0, 0 >$	$< 0, 0 >$	$< 0, 0 >$
2		WRITE(y, t_2)	$< 1, 0 >$	$< 0, 1 >$	$< 0, 0 >$	$< 0, 0 >$	$< 0, 1 >$
3		UNLOCK(l, t_2)	$< 1, 0 >$	$< 0, 2 >$	$< 0, 1 >$	$< 0, 0 >$	$< 0, 1 >$
4		WRITE(x, t_2)	$< 1, 0 >$	$< 0, 2 >$	$< 0, 1 >$	$< 0, 2 >$	$< 0, 1 >$
5	WRITE(x, t_1)		$< 1, 0 >$	$< 0, 2 >$	$< 0, 1 >$	**Race**	$< 0, 1 >$
6	LOCK(l, t_1)		$< 1, 1 >$	$< 0, 2 >$	$< 0, 1 >$	—	$< 0, 1 >$
7	WRITE(y, t_1)		$< 1, 1 >$	$< 0, 2 >$	$< 0, 1 >$	—	$< 1, 1 >$
8	UNLOCK(l, t_1)		$< 2, 1 >$	$< 0, 2 >$	$< 1, 1 >$	—	$< 1, 1 >$

(b)

Figure 3.12. Different interleavings of the threads in Figure 3.11. (a) First interleaving, (b) second interleaving.

Now consider the interleaving in Figure 3.12b. With such an interleaving, happens-before based detector is able to detect the potential race since the happens-before relation does not affect the accesses to variable x. Therefore, it can be concluded that happens-before data race detectors are dependent on the scheduler and may miss some potential data races.

3.3. Segment-Based Hybrid Data Race Detection Algorithm

3.3.1. Introduction

We have so far described two fundamental approaches, happens-before and lockset approaches. Happens-before approach misses some potential data races, in other words, produces false negatives. However, whenever it detects a potential data race, there is an alternative schedule such that the potential race literally happens in that schedule. Thus, happens-before detectors are considered to be *precise* detectors. On the other hand, lockset detectors output potential races even when there is no potential data race, in other words, they produce false positives. When these two approaches are examined in terms of performance, lockset detectors are scalable whereas happens-before based detectors are not. Happens-before detectors are implemented with a high memory and processing overhead. Our goal is to combine the benefits of both approaches described earlier and obtain a hybrid data race detector that have a good performance and few false positive results.

3.3.2. Algorithm

Any hybrid approach for data race detection is expected to provide abilities of lockset approaches and happens-before approach together. Our segment-based hybrid approach is based on Threadsanitizer [15]. We modify this algorithm in many different ways to improve performance and reduce false positives.

Two locksets are maintained in lockset detectors for each memory address. Similarly, two vector clocks are maintained for each memory address in happens-before detectors. For a hybrid approach, both of the locksets and the vector clocks must be maintained for each memory address. These requirements lead to huge memory overhead, which ends up with slow down in the execution. Our hybrid approach utilizes the concept of segment in order to overcome the overhead. A segment is formed by consecutive memory accesses of a single thread. No function calls or synchronization operations are allowed inside a segment. The outcome of this is that, all the instruc-

Table 3.1. Segment-based hybrid algorithm implementation data structures.

Thread t_i	
Vector Clock	VC_i
Writer Lockset	$WRLS(t_i)$
Reader Lockset	$RDLS(t_i)$
Segment s_i	
Vector Clock	$s_i.VC$
Writer Lockset	$WRLS(s_i)$
Reader Lockset	$RDLS(s_i)$
Condition Variable cv	
Vector Clock	$cv.VC$
Barrier br	
Vector Clock	$br.VC$
Memory Address x	
Writer Segment Set	$WRST_x.$
Reader Segment Set	$RDST_x.$

tions inside a segment are executed while the thread holds the same vector clock and locks. Thus, the vector clocks and the locksets could be kept on the granularity of segments, where the total number segments are much less than the total number of memory addresses in an execution for most of the applications. Therefore, this approach reduces the memory requirements and increases the performance considerably. Some other definitions and implementations of segment are discussed in [15, 18, 25]. Our formal definition of segment is discussed in Chapter 2. Shadow memory is used to track and store information about the execution of the instrumented application. The data structures that our algorithm utilizes on the shadow memory are shown in Table 3.1. Note that, for each memory address x, the writer segment set is, $WRST_x$, utilized to keep track of the segments that x is written in. Similarly, the reader segment set, $RDST_x$, is utilized to keep track of the segments that x is read in.

The algorithm in Figure 3.13 displays how our segment-based hybrid algorithm reacts each operation during execution.

Input: Thread t_i Generates An Operation *op*
Output: Potential Data Race Detected Or Not

1: **if** *op* is thread initialization **then**
2: $INIT(VC_i)$ ▷ Initialize Vector Clock
3: $WRLS(t_i) := \{\}$ ▷ Initialize Writer Lockset to empty set
4: $RDLS(t_i) := \{\}$ ▷ Initialize Reader Lockset to empty set
5: **else if** *op* is segment initialization **then**
6: $s_i.VC := t_i.VC$ ▷ Init clock, equal to t_i's VC
7: $WRLS(s_i) := WRLS(t_i)$ ▷ Init WRLS, equal to t_i's WRLS
8: $RDLS(s_i) := RDLS(t_i)$ ▷ Init RDLS, equal to t_i's RDLS
9: **else if** *op* is condition variable initialization **then**
10: $cv.CV := [0 \dots 0]$ ▷ Initialize $cv.VC$ such that all elements are zero.
11: **else if** *op* is condition variable wait **then**
12: $RECV(VC_i, cv.VC)$ ▷ Update thread's clock VC_i
13: **else if** *op* is condition variable signal **then**
14: $RECV(cv.VC, VC_i)$ ▷ Update CV's clock $cv.VC$
15: $INCREMENT(VC_i)$ ▷ Increment t_i's Vector Clock
16: **else if** *op* is write lock acquire **then**
17: $WRLS(t_i) := WRLS(t_i) \cup \{l\}$ ▷ Update WRLS of current thread
18: **else if** *op* is write lock release **then**
19: $WRLS(t_i) := WRLS(t_i) - \{l\}$ ▷ Update WRLS of current thread
20: $INCREMENT(VC_i)$ ▷ Increment t_i's Vector Clock
21: **else if** *op* is read lock acquire **then**
22: $RDLS(t_i) := RDLS(t_i) \cup \{l\}$ ▷ Update RDLS of current thread
23: **else if** *op* is read lock release **then**
24: $RDLS(t_i) := RDLS(t_i) - \{l\}$ ▷ Update RDLS of current thread
25: $INCREMENT(VC_i)$ ▷ Increment t_i's Vector Clock
26: **else if** *op* is write memory access **then**
27: $WRITE\_ACCESS(x, t_i)$
28: **else if** *op* is read memory access **then**
29: $READ\_ACCESS(x, t_i)$
30: **end if**

Figure 3.13. Segment-based hybrid data race detection algorithm - all operations.

Memory accesses are handled by the algorithms 3.14 and 3.15. The algorithm in Figure 3.14 is executed on each write access. Similarly, the algorithm in Figure 3.15 is executed on each read access.

Both read access and write access algorithms start similarly. Firstly, thread's current segment that is executing the access is fetched. Then, for the write access, writer segment set of the memory address is updated so that it only includes concurrent segments to the thread's current segment. Moreover, the current segment is added to writer segment set of the memory address. Similarly, reader segment set of the memory address is updated so that it only includes concurrent segments to the thread's current segment. For the read access, reader segment set of the memory address is updated so that it only includes concurrent segments to the thread's current segment. Then, the current segment is added to reader segment set of the memory address. Note that both $WRST$ and $RDST$ consist of concurrent segments. The reason is that non-concurrent segments, which are happens-before ordered, cannot cause a potential data race on a variable. In fact, by keeping concurrent segments we try to preserve the segments that may cause a potential data race on a variable if other conditions for a data race are met.

Input: Thread t_i Generates A Memory Access Operation On Memory Address x

Output: Potential Data Race Detected Or Not

1: **procedure** $WRITE\_ACCESS(x, t_i)$

2: $s_i := CurrentSegment(t_i)$ ▷ Current segment of thread t_i is segment s_i

▷ Update $WRST_x$ so it only includes concurrent writer segments with s_i

3: $WRST_x := \{s_x | s_x \in WRST_x \wedge \neg(s_x \rightarrow s_i)\} \cup s_i$

▷ Update $RDST_x$ so it only includes concurrent reader segments with s_i

4: $RDST_x := \{s_x | s_x \in RDST_x \wedge \neg(s_x \rightarrow s_i)\}$

5: $checkRace(WRST_x, RDST_x)$ ▷ Check race among $WRST$ and $RDST$

6: **end procedure**

Figure 3.14. Segment-based hybrid data race detection algorithm - write access.

As described above, read and write accesses update segment sets differently. On write accesses, both writer and reader segment sets are updated, however, on read accesses, only reader segment set is updated. The reason is as follows, on write accesses, it is safe to remove any of the read accesses from $RDST$. Remember that, $RDST_x$ consists of concurrent segments that x is read in. Since there is no read-read type of race, removing any segment from $RDST$ does not lead to missing any races. On the contrary, on read accesses it is not safe to remove any segment from $WRST$. The reason is that, it may lead to missing a write-write race because the removed segment could have a potential race with one of the segment in the same set. The outcome of this is that all segments within any segment set is concurrent with each other. However, not all segments in $RDST$ are concurrent with all segments in $WRST$, which is handled while checking race among $WRST$ and $RDST$.

Input: Thread t_i Generates A Memory Access Operation On Memory Address x
Output: Potential Data Race Detected Or Not

1: **procedure** $READ\_ACCESS(x, t_i)$
2: $s_i := CurrentSegment(t_i)$ ▷ Current segment of thread t_i is segment s_i
▷ Update $RDST_x$ so it only includes concurrent reader segments with s_i
3: $RDST_x := \{s_x | s_x \in RDST_x \wedge \neg(s_x \rightarrow s_i)\} \cup s_i$
4: $checkRace(WRST_x, RDST_x)$ ▷ Check race among $WRST$ and $RDST$
5: **end procedure**

Figure 3.15. Segment based hybrid data race detection algorithm - read access.

Checking race among $WRST$ and $RDST$ is described in the algorithm in Figure 3.16, which requires two parameters as input. First parameter is a segment set such that all elements in the set are concurrent and write accessed to a variable x. Similarly, second parameter is a segment set such that all elements in the set are concurrent and read accessed to a variable x. Note that, not all segments in first parameter are concurrent with all elements in the second parameter.

Input: Writer Segment Set $WRST$, Reader Segment Set $RDST$

Output: Potential Data Race Detected Or Not

```
1:  procedure checkRace(WRST, RDST)
2:      for wr_1 ∈ WRST do
3:          for wr_2 ∈ WRST do
4:              if wr_1.WRLS ∩ wr_2.WRLS = {} then
5:                  return true;                          ▷ Write-Write Race Found
6:              end if
7:          end for
8:          for  rd ∈ RDST do
9:              if ¬(wr_1 → rd) ∧ (wr_1.WRLS ∩ rd.RDLS = {}) then
10:                 return true;                          ▷ Read-Write Race Found
11:             end if
12:         end for
13:     end for
14:     return false;                                     ▷ No Race Found
15: end procedure
```

Figure 3.16. Segment-based hybrid algorithm check potential data race.

The algorithm checks for common lock among concurrent segments such that one segment is a writer segment and other is either a reader or a writer segment. If the algorithm could not find a common lock among any concurrent segments, it returns true for indicating a data race. At this point of the algorithm, it is more obvious that our algorithm is a hybrid of lockset and happens-before algorithms. If any two segments are ordered by happens-before relation, they are not even checked for race, which is more closer to happens-before detection. Moreover, if two segments are concurrent they are checked for holding a common lock, which is more closer to lockset detection.

3.3.3. Sample Executions of Segment-Based Hybrid Approach

In this section three different sample executions of segment-based hybrid approach are illustrated.

3.3.3.1. Writer Lock: No Potential Race. In this example, an execution of segment-based hybrid approach with write-held locks is illustrated. The code that is executed is shown in Figure 3.17. The shadow memory state on each line is described in Figure 3.18. Also, more detailed explanation of each step is shown below. For this specific example, two threads write to a shared variable while holding a common write-held lock. The execution ends up with no race, which is **True Negative**.

Line	Thread 1	Thread 2
1	WR_LOCK(l_1, t_1)	
2	WRITE(x, t_1) $\in s_1$	
3	WR_UNLOCK(l_1, t_1)	
4		WR_LOCK(l_1, t_2)
5		WRITE(x, t_2) $\in s_2$
6		WR_UNLOCK(l_1, t_2)

Figure 3.17. A program execution with segment-based hybrid algorithm - 1.

Line	VC_1	WRLS(t_1)	RDLS(t_1)	VC_2	WRLS(t_2)	RDLS(t_2)	$WRST_x$	$RDST_x$
0	$<1,0>$	{}	{}	$<0,1>$	{}	{}	{}	{}
1	$<1,0>$	$\{l_1\}$	{}	$<0,1>$	{}	{}	{}	{}
2	$<1,0>$	$\{l_1\}$	{}	$<0,1>$	{}	{}	$\{s_1\}$	{}
3	$<2,0>$	{}	{}	$<0,1>$	{}	{}	$\{s_1\}$	{}
4	$<2,0>$	{}	{}	$<0,1>$	$\{l_1\}$	{}	$\{s_1\}$	{}
5	$<2,0>$	{}	{}	$<0,1>$	$\{l_1\}$	{}	$\{s_1, s_2\}$	{}
6	$<2,0>$	{}	{}	$<0,2>$	{}	{}	$\{s_1, s_2\}$	{}

Figure 3.18. Update of data structures during the execution of the program in Figure 3.17.

(i) Line 0:

On Initialization, the algorithm in Figure 3.13 is executed for both first and second threads with input operation as *thread initialization*.

- $VC_1 :=< 1, 0 >$
- $WRLS(t_1) := \{\}$
- $RDLS(t_1) := \{\}$
- $VC_2 :=< 0, 1 >$
- $WRLS(t_2) := \{\}$
- $RDLS(t_2) := \{\}$
- $WRST_x := \{\}$
- $RDST_x := \{\}$

(ii) Line 1:

Thread 1 acquires the write lock l_1, thus the algorithm in Figure 3.13 is executed with input operation as *write lock acquire*.

- $WRLS(t_1) := WRLS(t_1) \cup \{l_1\}$

(iii) Line 2:

Firstly, the algorithm in Figure 3.13 is executed with input operation as *segment initialization*. Then, thread 1 writes to variable x, thus the algorithm in Figure 3.14 is executed.

- $s_1.VC := t_1.VC$
- $WRLS(s_1) := WRLS(t_1)$
- $RDLS(s_1) := RDLS(t_1)$
- $WRST_x := \{s_x | s_x \in WRST_x \wedge \neg(s_x \rightarrow s_1)\} \cup s_1$
- $RDST_x := \{s_x | s_x \in RDST_x \wedge \neg(s_x \rightarrow s_1)\}$
- $checkRace(WRST_x, RDST_x)$ returns 0, ***No Race, True Negative.***

(iv) Line 3:

Thread 1 releases the write lock l_1, thus the algorithm in Figure 3.13 is executed with input operation as *write lock release*.

- $WRLS(t_1) := WRLS(t_1) - \{l_1\}$
- $INCREMENT(VC_1)$

(v) Line 4:

Thread 2 acquires the write lock l_1, thus the algorithm in Figure 3.13 is executed with input operation as *write lock acquire*.

- $WRLS(t_2) := WRLS(t_2) \cup \{l_1\}$

(vi) Line 5:

Firstly, the algorithm in Figure 3.13 is executed with input operation as *segment initialization*. Then, thread 2 writes to variable x, thus the algorithm in Figure 3.14 is executed.

- $s_2.VC := t_2.VC$
- $WRLS(s_2) := WRLS(t_2)$
- $RDLS(s_2) := RDLS(t_2)$
- $WRST_x := \{s_x | s_x \in WRST_x \wedge \neg(s_x \rightarrow s_2)\} \cup s_2$
- $RDST_x := \{s_x | s_x \in RDST_x \wedge \neg(s_x \rightarrow s_2)\}$
- $checkRace(WRST_x, RDST_x)$ returns 0, ***No Race, True Negative.***

(vii) Line 6:

Thread 2 releases the write lock l_1, thus the algorithm in Figure 3.13 is executed with input operation as *write lock release*.

- $WRLS(t_2) := WRLS(t_2) - \{l_1\}$
- $INCREMENT(VC_2)$

3.3.3.2. Reader-Writer Lock: Potential Race. In this example, an execution of segment-based hybrid approach with write-held and read-held locks is illustrated. The code that is executed is shown in Figure 3.19. The shadow memory state on each line is described in Figure 3.20. Also, more detailed explanation of each step is shown below. For this specific example, two threads write to a variable while holding a common lock. However, one thread holds the lock as a writer lock and the other thread holds the lock as a reader lock. The execution ends up with race, which is **True Positive**.

(i) Line 0:

On Initialization, the algorithm in Figure 3.13 is executed for both first and second threads with input operation as *thread initialization*.

Line	Thread 1	Thread 2
1	WR_LOCK(l_1, t_1)	
2	WRITE(x, t_1) $\in s_1$	
3	WR_LOCK(l_1, t_1)	
4		RD_LOCK(l_1, t_2)
5		WRITE(x, t_2) $\in s_2$
6		RD_LOCK(l_1, t_2)

Figure 3.19. A program execution with segment-based hybrid algorithm - 2.

Line	VC_1	WRLS(t_1)	RDLS(t_1)	VC_2	WRLS(t_2)	RDLS(t_2)	$WRST_x$	$RDST_x$
0	$<1,0>$	{}	{}	$<0,1>$	{}	{}	{}	{}
1	$<1,0>$	$\{l_1\}$	{}	$<0,1>$	{}	{}	{}	{}
2	$<1,0>$	$\{l_1\}$	{}	$<0,1>$	{}	{}	$\{s_1\}$	{}
3	$<2,0>$	{}	{}	$<0,1>$	{}	{}	$\{s_1\}$	{}
4	$<2,0>$	{}	{}	$<0,1>$	{}	$\{l_1\}$	$\{s_1\}$	{}
5	$<2,0>$	{}	{}	$<0,1>$	{}	$\{l_1\}$	$\{s_1, s_2\}$	{}
6	$<2,0>$	{}	{}	$<0,2>$	{}	{}	$\{s_1, s_2\}$	{}

Figure 3.20. Update of data structures during the execution of the program in Figure 3.19.

- $VC_1 :=< 1,0 >$
- $WRLS(t_1) := \{\}$
- $RDLS(t_1) := \{\}$
- $VC_2 :=< 0,1 >$
- $WRLS(t_2) := \{\}$
- $RDLS(t_2) := \{\}$
- $WRST_x := \{\}$
- $RDST_x := \{\}$

(ii) Line 1:

Thread 1 acquires the write lock l_1, thus the algorithm in Figure 3.13 is executed with input operation as *write lock acquire*.

- $WRLS(t_1) := WRLS(t_1) \cup \{l_1\}$

(iii) Line 2:

Firstly, the algorithm in Figure 3.13 is executed with input operation as *segment initialization*. Then, thread 1 writes to variable x, thus the algorithm in Figure 3.14 is executed.

- $s_1.VC := t_1.VC$
- $WRLS(s_1) := WRLS(t_1)$
- $RDLS(s_1) := RDLS(t_1)$
- $WRST_x := \{s_x | s_x \in WRST_x \wedge \neg(s_x \rightarrow s_1)\} \cup s_1$
- $RDST_x := \{s_x | s_x \in RDST_x \wedge \neg(s_x \rightarrow s_1)\}$
- $checkRace(WRST_x, RDST_x)$ returns 0, ***No Race, True Negative.***

(iv) Line 3:

Thread 1 releases the write lock l_1, the algorithm in Figure 3.13 is executed with input operation as *write lock release*.

- $WRLS(t_1) := WRLS(t_1) - \{l_1\}$
- $INCREMENT(VC_1)$

(v) Line 4:

Thread 2 acquires the read lock l_1, thus the algorithm in Figure 3.13 is executed with input operation as *read lock acquire*.

- $RDLS(t_2) := RDLS(t_2) \cup \{l_1\}$

(vi) Line 5:

Firstly, the algorithm in Figure 3.13 is executed with input operation as *segment initialization*. Then, thread 2 writes to variable x, thus the algorithm in Figure 3.14 is executed.

- $s_2.VC := t_2.VC$
- $WRLS(s_2) := WRLS(t_2)$
- $RDLS(s_2) := RDLS(t_2)$
- $WRST_x := \{s_x | s_x \in WRST_x \wedge \neg(s_x \rightarrow s_2)\} \cup s_2$
- $RDST_x := \{s_x | s_x \in RDST_x \wedge \neg(s_x \rightarrow s_2)\}$
- $checkRace(WRST_x, RDST_x)$ returns 1, ***Race FOUND, True Positive.***

(vii) Line 6:

Thread 2 releases the read lock l_1, thus the algorithm in Figure 3.13 is executed

with input operation as *read lock release*.

- $RDLS(t_2) := RDLS(t_2) - \{l_1\}$
- $INCREMENT(VC_2)$

3.3.3.3. Signal/Wait: No Potential Race. In this example, an execution of segment-based hybrid approach with condition variables is illustrated. The code that is executed is shown in Figure 3.21. The shadow memory state on each line is described in Table 3.22. Also, more detailed explanation of each step is shown below. For this specific example, a condition variable is notified by a thread and the notified thread writes to a variable that first thread already written. The execution ends up with no race, which is **True Negative**. This execution shows that the hybrid algorithm could detect happens-before relations that lockset algorithm is not able to detect.

Line	Thread 1	Thread 2
1	WRITE$(x, t_1) \in s_1$	
2	SIGNAL(cv, t_1)	
3		WAIT(cv, t_2)
4		WRITE$(x, t_2) \in s_2$

Figure 3.21. A program execution with segment-based hybrid algorithm - 3.

Line	$cv.VC$	VC_1	WRLS(t_1)	RDLS(t_1)	VC_2	WRLS(t_2)	RDLS(t_2)	$WRST_x$	$RDST_x$
0	$< 0,0 >$	$< 1,0 >$	{}	{}	$< 0,1 >$	{}	{}	{}	{}
1	$< 0,0 >$	$< 1,0 >$	{}	{}	$< 0,1 >$	{}	{}	$\{s_1\}$	{}
2	$< 1,0 >$	$< 1,0 >$	{}	{}	$< 0,1 >$	{}	{}	$\{s_1\}$	{}
3	$< 1,0 >$	$< 2,0 >$	{}	{}	$< 1,1 >$	{}	{}	$\{s_1\}$	{}
4	$< 1,0 >$	$< 2,0 >$	{}	{}	$< 1,1 >$	{}	{}	$\{s_2\}$	{}

Figure 3.22. Update of data structures during the execution of the program in Figure 3.21.

(i) Line 0:

On Initialization, the algorithm in Figure 3.13 is executed for both first and second threads with input operation as *thread initialization*. Moreover, the algorithm in Figure 3.13 is executed with input operation as *condition variable initialization*.

- $VC_1 :=< 1,0 >$

- $WRLS(t_1) := \{\}$
- $RDLS(t_1) := \{\}$
- $VC_2 :=< 0, 1 >$
- $WRLS(t_2) := \{\}$
- $RDLS(t_2) := \{\}$
- $WRST_x := \{\}$
- $RDST_x := \{\}$
- $cv.CV :=< 0, 0 >$

(ii) Line 1:

Firstly, the algorithm in Figure 3.13 is executed with input operation as *segment initialization*. Then, thread 1 writes to variable x, thus the algorithm in Figure 3.14 is executed.

- $s_1.VC := t_1.VC$
- $WRLS(s_1) := WRLS(t_1)$
- $RDLS(s_1) := RDLS(t_1)$
- $WRST_x := \{s_x | s_x \in WRST_x \wedge \neg(s_x \rightarrow s_1)\} \cup s_1$
- $RDST_x := \{s_x | s_x \in RDST_x \wedge \neg(s_x \rightarrow s_i)\}$
- $checkRace(WRST_x, RDST_x)$ returns 0, ***No Race, True Negative.***

(iii) Line 2:

Thread 1 signals condition variable cv, thus the algorithm in Figure 3.13 is executed with input operation as *condition variable signal*.

- $RECV(cv.VC, VC_1)$
- $INCREMENT(VC_1)$

(iv) Line 3:

Thread 2 acquires condition variable cv, thus the algorithm in Figure 3.13 is executed with input operation as *condition variable wait*.

- $RECV(VC_2, cv.VC)$

(v) Line 4:

Firstly, the algorithm in Figure 3.13 is executed with input operation as *segment initialization*. Then, thread 2 writes to variable x, thus the algorithm in Figure 3.14 is executed. Note that s_1 is removed from $WRST_x$ since $s_1 \rightarrow s_2$.

- $s_2.VC := t_2.VC$
- $WRLS(s_2) := WRLS(t_2)$
- $RDLS(s_2) := RDLS(t_2)$
- $WRST_x := \{s_x | s_x \in WRST_x \wedge \neg(s_x \rightarrow s_2)\} \cap s_2$
- $RDST_x := \{s_x | s_x \in RDST_x \wedge \neg(s_x \rightarrow s_2)\}$
- $checkRace(WRST_x, RDST_x)$ returns 0, ***No Race, True Negative.***

4. INTERNALS OF THE DATA RACE DETECTORS

In this chapter, we describe some of the implementation details related to segments, semaphores, barriers, timed synchronization operations, locksets, segment sets, dynamic memory allocation and arrays. These details are required for better understanding of the behaviors of our detectors.

4.1. Segments

In this section, we investigate the reasoning behind segments and explore it in detail. First of all, one of the most compelling problems of dynamic data race detection is its run time and memory overhead. These overheads prevent utilization of dynamic data race detection tools on real development environments. Thus, any algorithm proposed for dynamic data race detection should deal with the overheads.

During the execution, all information associated with *memory accesses* are maintained on the shadow memory. In lockset and happens-before implementations, the shadow memory is built on the granularity of individual *memory addresses*. This means that the memory required to detect potential races is proportional to the number of memory addresses accessed during the execution.

Since no synchronization operations are allowed inside a segment, all the memory accesses use the same vector clocks and the same locks. This observation leads to the shadow memory to be built on the granularity of *segments*. Though the potential data races happen on the granularity of memory addresses, they are observed on the granularity of the segments. As we later show in Table 6.2, on the average the total number of segments is much less than the total number memory addresses accessed during the execution. Therefore, the size of the shadow memory maintained is much less in the segment-based hybrid implementation compared to the happens-before implementations. Detailed memory requirement comparison of happens-before algorithm and segment-based hybrid approach is explored in Chapter 6.

4.1.1. Initialization and Termination of Segments

In this section initialization and termination of segments is discussed from the perspective of segment-based hybrid algorithm. Note that, initializing a new segment terminates the previous one.

4.1.1.1. Synchronization Operations. As discussed in Section 3.3, synchronization operations change either the vector clock or the lockset information of any thread. From the definition of segments, execution of these operations terminate the current segment. Similarly, a new segment is expected to be initialized after the execution of synchronization operations. Figure 4.1 shows how vector clock and lockset values are changed for some of the synchronization operations that results in initialization/termination of segments.

4.1.1.2. Traces. As discussed above, executing synchronization operations terminates a segment and initializes a new one. However, assume that there are no synchronization operations executed, then large segments consisting of too many lines of source code would appear. The length of segments is important in determining the precision of the line number of the detected races in the source files.

First of all, it is not feasible to keep track of all $READ/WRITE$ operations' line information inside a segment, instead line information of the first operation of the segment is maintained in the shadow memory. There are two reasons that the line information are not maintained for each operation. Firstly, it has computation complexity embedded in the DBI to fetch the line number of an operation in the source. Secondly, it leads to an extra memory overhead. Even most of the operations are not related to any of the data races, their line information should have been maintained. Thus, when the data race detection algorithm detects a potential race, the output would include the first line numbers of the two segments which are involved in the race.

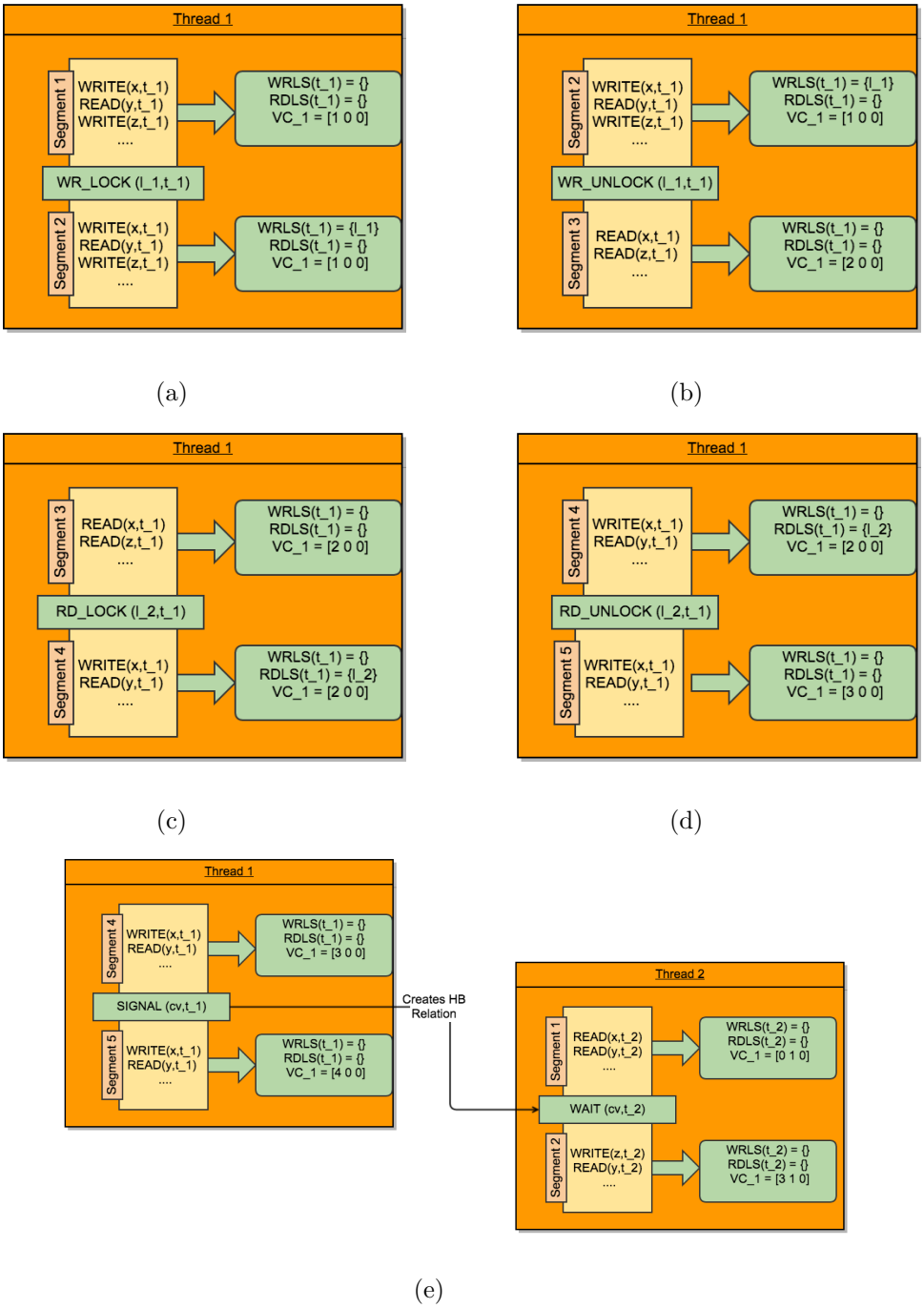

Figure 4.1. Synchronization operations resulting in initialization/termination of a segment. (a) Acquire writer lock, (b) release writer lock, (c) acquire reader lock, (d) release reader lock, (e) condition variable.

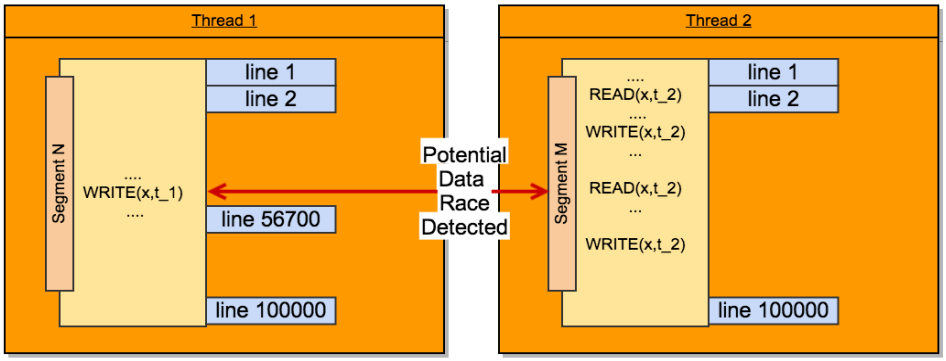

Figure 4.2. Difficult to locate the exact place of race with large segments.

To clarify how segment length is important for the outputted line information of the potential data races, we give an extreme example. Assume that two threads execute two different functions concurrently, and there are no synchronization operations in the functions. Moreover, each function consists of roughly one million lines of code. When a potential race is detected, there exists three different data identifying the race. They are the first lines of the two segments that are involved in the race and the line information of the operation where the potential data race detected. For instance, consider Figure 4.2. There are two threads, and there is a conflicting access on memory address x. When the potential race detected and output, the following information is obtained:

- Potential data race happens on line 57,600 of thread 1 on the variable x.
- The first segment that is associated with the potential race starts at line 1 of thread 1-ends at line 1,000,000.
- The second segment that is associated with the potential race starts at line 1 of thread 2-ends at line 1,000,000.

At first glance it may seem that there is enough information for understanding the cause of the race. However, the line information that the conflicting access executed on the second thread is not identified. Therefore, the programmer must search all the accesses to memory address x and should be able to identify which access is the conflicting

access. As the segment size grows, the programmer's search space for determining the exact line in the source code increases. For the above extreme example, the programmer has to search 1,000,000 lines in order to understand the actual cause of the race on thread 2. Such an output would be useless and infeasible to search from the perspective of a programmer.

In order to obtain more illuminative outputs, the sizes of segments should be limited. In our implementation, we utilize traces for this purpose. Traces usually begin at the target of a taken branch and end with an unconditional branch, including calls and returns [11]. In our implementation each trace is mapped to a segment. For the example above, if each trace were mapped to a segment, the cause of the race on thread 2 would be confined to a few lines of code (a trace). Thus, it would be easy to search and find the memory access(es) which lead to the race compared to searching 1,000,000 lines for the access(es). Although increasing the total number of segments results in increased memory requirements and computation overhead, we obtain more meaningful outputs for potential data races.

In our implementation, function calls are also treated as traces. Even though a function call may not contain a branch and/or function/return calls, it is considered as a segment with the same motivation of confining the size of segments. A function call always starts a new segment and a return statement always terminates the current segment. Limiting the segment sizes with functions simplifies output information of the detector.

In Figure 4.3, some examples of traces are shown. For the first three examples in the figure, the execution has a single entry point, and there are more than one possible exit points. The last example shows a function with no branches or synchronization operations. As mentioned above, it is sufficient to start a new segment by a function call.

Trace 1

```
/* code block entrance point (e_start)*/
if (str == "firststr")
  return 0; //first possible exit point (e_end_0)
else if (str == "secondstr")
  return 1; //second possible exit point (e_end_1)
else
  return 2;   //third possible exit point (e_end_2)
```

(a)

Trace 2

```
/* code block entrance point (e_start)*/
if (str == "firststr")
  f1(); //first possible exit point (e_end_0)
else if (str == "secondstr")
  f2(); //second possible exit point (e_end_1)
```

(b)

Trace 3

```
/* code block entrance point (e_start)*/
  switch(grade)
  {
    case 'A':
      grade_A();//first possible exit point (e_end_0)
    case 'F':
      grade_F();//second possible exit point (e_end_1)
    default:
      grade_default();//third possible exit point (e_end_2)
  }
```

(c)

Trace 4

```
int dummy_function()
{
  /* code block entrance point (e_start)*/
  int x=0;
  for (int i=0; i<100; ++i)
    x = i + x * i;

  /* even tough there is single exit in the function,
  each function is considered as a trace*/
  return x;
}
```

(d)

Figure 4.3. Trace examples: (a) exits are return statements, (b) exits are call statements, (c) exits are call statements, (d) any function is considered a trace.

4.2. Discussion on Semaphores

A semaphore can be considered as an integer with the following properties [26] :

(i) A semaphore is initialized to an integer value. After a semaphore is initialized, its value cannot be assigned to another integer, the allowed operations are increment (*sem_post*) and decrement (*sem_wait*) operations.

(ii) After a semaphore is decremented, if the integer value of the semaphore is negative, the thread executing the decrement operation is blocked until the semaphore is incremented by another thread.

(iii) After a semaphore is incremented, if there exists any threads blocked on the semaphore, one of them becomes unblocked and continues its execution.

In general, a semaphore is used to solve producer-consumer type of synchronization problems where the producers increment the semaphore, and the consumers decrement it. In Figure 4.4 an example of such situation is illustrated. The producer thread puts a produced element to a shared data structure then posts the semaphore in order to notify the consumer thread. When the consumer thread is notified via the semaphore, it continues its execution. In this particular example, the happens-before relation among the post/wait calls are clearly observed.

Thread 1	Thread 2
$products.push(product)$	
$sem\_post(s, t_1)$	
	$sem\_wait(s, t_2)$
	$product = products.get()$

Figure 4.4. Semaphore post/wait example - 1.

Now, consider Figure 4.5, there are two producers and a single consumer. Though there is a happens-before relation among one of the *sem_post* operations with the

sem_wait operation, it is not possible to determine which of the *sem_post* matched the *sem_wait*. The reason as is that a semaphore is nothing but an integer with some properties defined above and *sem_post*/*sem_wait* calls are reflection of increment/decrement of integers. Assuming the initial value of an integer is zero and two threads increment it once and one thread decrements it once, the expected result is one. However, it is not possible to determine whether the decremented value of the integer is the value that the first thread incremented or the second thread incremented.

Thread 1	Thread 2	Thread 3
$products.push(product)$	$products.push(product)$	
$sem\_post(s, t_1)$	$sem\_post(s, t_2)$	
		$sem\_wait(s, t_3)$
		$product = products.get()$

Figure 4.5. Semaphore post/wait example - 2.

In our implementation, in order to overcome this uncertainty in creating the happens-before relation, we randomly choose one of the *sem_post* calls to be matched with one of the *sem_wait* calls. The happens-before relation is created among the randomly matched *sem_post*/*sem_wait* calls.

4.3. Discussion on Barriers

In some multithreaded applications, multiple threads execute concurrently up to a specified point. Until all the required threads come to the specified point, other threads must wait. This kind of synchronization problem can be solved by *barriers*.

Consider Figure 4.6, three threads execute *do_some_other_job*() and each one must wait the others in order to continue its execution. This synchronization primitive creates a happens-before relation among the threads. Therefore, in our implementation, for all threads, we assume that events preceding the $BARRIER\_WAIT$ call are

executed prior to events following $BARRIER\_WAIT$.

Thread 1	Thread 2	Thread 3
$do\_some\_job()$		
$barrier\_wait(br, t_1)$		
	$do\_some\_job()$	
	$barrier\_wait(br, t_2)$	
		$do\_some\_job()$
		$barrier\_wait(br, t_3)$
$do\_some\_other\_job()$	$do\_some\_other\_job()$	$do\_some\_other\_job()$

Figure 4.6. Barrier example.

4.4. Discussion on Timed Synchronization Operations

In Pthread API, there are some *timed* synchronization operations such as *pthread_mutex_timedlock* and *pthread_cond_timedwait*. These *timed* operations are intended for allowing threads to stop waiting after a given amount of time. For instance, *pthread_cond_timedwait*, similar to *pthread_cond_wait*, waits for a condition to hold and to be notified by other threads. However, if the condition is not met in a specified time interval, the waiting thread is allowed to continue its execution. Return values of the function calls are utilized in order to distinguish whether the *timed* synchronization operation continues its execution as if it were its *non-timed* counterpart or its time is expired. This kind of timed synchronization primitives prevent threads to become blocked and handle problematic situations.

The example in Figure 4.7 [1] illustrates a typical usage of *timed* synchronization operations. While thread 1 is expected to get input from user, thread 2 calls *pthread_mutex_timedlock*. If thread 2 waits more than n seconds, it automatically stops waiting on the lock. Thereafter, since thread 2 knows that the required condition

[1] Example is adopted from http://tuxthink.blogspot.com.tr/2013/01/using-pthreadmutextimedlock-in-linux.html

```
void thread_1()
{
 pthread_mutex_lock(&lock);
 input = readInput();
 pthread_mutex_unlock(&lock);
}
```

(a)

```
void thread_2()
{
  ret = pthread_mutex_timedlock(&lock, maxtime);
  if (ret != SUCCESS)
  {
    print "Mutex wait timed out";
    return
  }
  useInput(input);
  pthread_mutex_unlock(&lock);
}
```

(b)

Figure 4.7. Timed mutex example, (a) first thread, (b) second thread.

is not met, it does not try to use the user input. The operations behave the same as their *non-timed* counterparts until the time limit is exceeded. In our implementations, by utilizing the return values, if the time limit is not exceeded, we treated the *timed* operations as if they were *non-timed*. However, if the time limit is exceeded, we discard the synchronization operations.

4.5. Handling of Locksets and Segment Sets

For lockset algorithm, the total number of locksets are proportional to shared memory address count. Similarly, for the hybrid approach the total number of locksets are proportional to the total segment count. However, generally many of the locksets are formed by the same locks. Consider Figure 4.8, two memory addresses are accessed while t_1 is holding l_1 and l_2. Locksets of both memory addresses consist of l_1 and l_2. Therefore, there is no need to keep two separate locksets for both memory addresses. Instead, a single lockset can be referenced via its ID. Thus, as proposed in [1], in our implementation of locksets, we keep a lockset table and each lockset is accessed by its ID in the table. Whenever a new combination of lockset is formed, it is firstly searched in the table. If the new combination does not exist, it is added to the lockset table by a new ID. Figure 4.9 illustrates the same execution with lockset tables.

Thread 1	CLS(x)	CLS(y)	LS(t_1)
LOCK(l_1, t_1)	{}	{}	$\{l_1\}$
LOCK(l_2, t_1)	{}	{}	$\{l_1, l_2\}$
WRITE(x, t_1)	$\{l_1, l_2\}$	{}	$\{l_1, l_2\}$
WRITE(y, t_1)	$\{l_1, l_2\}$	$\{l_1, l_2\}$	$\{l_1, l_2\}$

Figure 4.8. Execution of lockset algorithm with different locksets.

Thread 1	CLS(x)	CLS(y)	LS(t_1)
LOCK(l_1, t_1)	LS_0	LS_0	LS_1
LOCK(l_2, t_1)	LS_0	LS_0	LS_2
WRITE(x, t_1)	LS_2	LS_0	LS_2
WRITE(y, t_1)	LS_2	LS_2	LS_2

Lockset ID	Locks
LS_0	{}
LS_1	$\{l_1\}$
LS_2	$\{l_1, l_2\}$

Figure 4.9. Execution of lockset algorithm with different locksets and a lockset table.

For applications that consist of heavy array operations, this implementation decreases the memory requirements considerably. The reason is that many memory addresses are generally accessed consecutively while a thread is holding the same locks. Consider the example in Figure 4.10. All memory addresses of the array *shared_array* is accessed consecutively while holding locks l_1 and l_2. The lockset of each memory address is expected to include the same locks which is optimized by utilizing a lockset table.

In our implementation of hybrid algorithm, we utilized the above solution based on tables for both locksets and segment sets. Note that, the total number of segment

```
int shared_array[1000];
void calculate()
{
        pthread_mutex_lock(&l1);
        pthread_mutex_lock(&l2);

        for (int i=0; i<1000;++i)
          shared_array[i] = shared_array[i]  * 2 + i;

        pthread_mutex_unlock(&l2);
        pthread_mutex_unlock(&l1);
}
```

Figure 4.10. Locksets with arrays.

sets is proportional to the shared memory address count. For the segment sets that are formed by the same segments are not created again and again. Instead, the unique segment sets are maintained in the segment sets table and accessed by an ID.

4.6. Handling of Dynamic Memory Allocation

C/C++ applications can dynamically allocate memory via *malloc*, *calloc*, *realloc* and *new* function calls. The allocated memory is deallocated via $free$ and *delete* function calls. While potential data races are being searched, it is crucial to track the dynamic allocations and deallocations to prevent false alarms. Consider the example in Figure 4.11. Thread 1 allocates memory to $first\_array$ and writes all elements of it. Finally, Thread 1 deallocates the memory. After a while, Thread 2 allocates memory to *second_array*. Since the $first\_array$ is already deallocated, it is very likely that the operating system provides the same memory addresses of $first\_array$ to *second_array*. For this example, assume that operating system provided the same addresses to both of the arrays. The memory addresses on the *second_array* is written by the second thread without any synchronization being executed. Since the write accesses to the same memory addresses are concurrent, all data race detection algorithms conclude that there is a potential data race. Although the accesses are concurrent, the result is certainly *false positive* since each thread is meant to access non-shared memory. In our implementations, we overcome this problem by tracking all memory allocations and

deallocations. Whenever a deallocation is executed, all the shadow memory state that is kept for the corresponding allocation call is cleaned up. Therefore, for the example in Figure 4.11, when first thread calls free, all the shadow memory maintained for $first\_array$ is cleaned up, which prevents the false positive result.

```
void thread_1()
{
        int* first_array=malloc(N);
        write_all(first_array);
        free(first_array);
}

void thread_2()
{
        int* second_array=malloc(N);
        write_all(second_array);
        free(second_array);
}
```

Figure 4.11. Dynamic memory allocation example.

4.7. Potential Data Races in Arrays

An array is formed by consecutive members of the same type of data. Generally, each element of an array is accessed consecutively by a thread, no synchronization operations are called among the accesses.

```
void thread_1()
{
        for(i=0;i<n:++i)
           a[i]=a[i]+1;
}
void thread_2()
{
        for(i=0;i<n:++i)
           a[i]=a[i]*a[i];
}
```

Figure 4.12. Potential data race example for arrays.

Consider the example in Figure 4.12. Assuming that the threads are concurrent, any data race algorithm is expected to find n potential data races where n is the length of the array a. However, it is not meaningful to handle n potential data races where the sources of them are the same. Instead, a single warning in arrays is more meaningful and easy to handle for programmers. Therefore, in our implementations we generate a single warning for arrays even though there may exist more than one potential data races.

5. OPTIMIZATIONS ON SEGMENT-BASED HYBRID ALGORITHM

We proposed and implemented four optimizations on segment-based hybrid algorithm. In this chapter, we are going to explore those optimizations and their results are described in Chapter 6.

5.1. Optimization 1: Storing Vector Clock Comparison History Cache

In segment-based hybrid approach, the same vector clock comparisons are executed repeatedly. Maintaining a limited vector clock history cache for the previously calculated vector clock comparisons increase performance of data race detection. We implemented two different solutions for the caching problem. Firstly, we proposed a global cache, which is utilized for keeping vector clock comparison results for all the comparisons executed. Whenever two vector clocks are compared, the result is added to the global cache. Then, if these two vector clocks are compared again, the result is fetched from the cache. Secondly, we proposed local caches for each vector clock. In this method, the comparison result of vector clocks are added and searched in local caches implemented in each vector clock. These two methods are discussed below. Note that, caching is applied to only vector clocks of segments since after initialization their values do not change. However, vector clocks' of threads and synchronization objects change during execution, thus caching them requires extra computation. Therefore, we do not apply caching to them.

Consider the example in Figure 5.1. There are three variables x, y and z. These variables are accessed by two different threads and on each access their segment sets must be updated. As shown in the Figure, for three different variables, the same vector clock comparisons are executed, vector clocks of segment 1 and segment 2. If a cache exists, a single comparison is executed and the consecutive two comparison results is fetched from the cache.

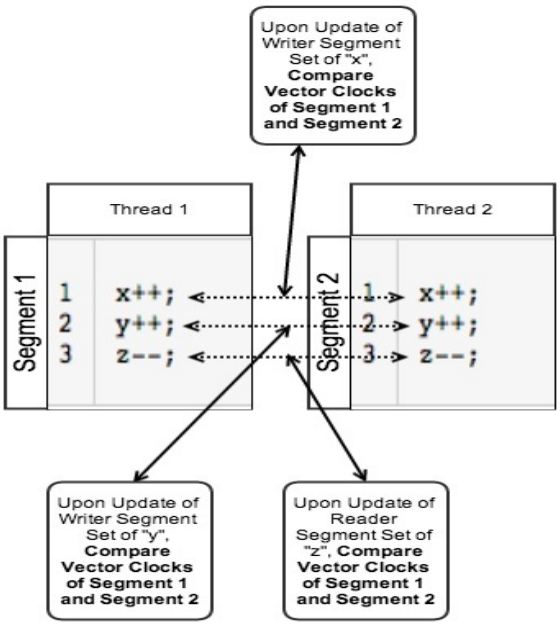

Figure 5.1. Vector clock history improvement on segment-based hybrid algorithm.

Implementing a shared global data structure and caching the results of the vector clock comparisons on that data structure is a straightforward solution. Consider the algorithm in Figure 5.2. Firstly, we define a global shared *All_Comparisons* data structure for caching purpose. On each vector clock comparison, the global shared *All_Comparisons* cache is accessed. Since it is a shared resource, it must be accessed by a common lock, thus the lock *global_lock* is acquired on each access to it. However, acquiring and releasing a lock on each vector clock comparison by any thread considerably decreases the whole performance of the data race detection procedure. All threads that execute a vector clock comparison must wait for acquiring the lock such that it becomes suitable. This creates a bottleneck in the execution. Though altering the maximum number of history maintained changes the performance, on the average slow down is 10% in this approach as shown by our experiments. We omitted these results since an optimization which leads to performance decrease is not valuable. Instead, we solved the performance problems of the proposed solution with another approach as discussed below.

Input: Vector Clock vc_1, Vector Clock vc_2, Global Shared Cache $All\_Comparisons$
Output: Comparison Result of vc_1 and vc_2

1: $LOCK(global\_lock)$;
2: $comparison\_value := All\_Comparisons(vc_1, vc_2)$;
3: **if** $comparison\_value == None$ **then** ▷ vc_1 and vc_2 has not been compared
4: $comparison\_value := compare(vc_1, vc_2)$
5: $All\_Comparisons(vc_1, vc_2) := comparison\_value$
6: **end if**
7: $UNLOCK(global\_lock)$;
8: return $comparison\_value$; ▷ Return the Comparison Result

Figure 5.2. Segment-based hybrid data race detection algorithm - store vector clock comparison history on global shared data.

Since the global shared vector clock history is not feasible, in our implementation the vector clock history is maintained on the basis of vector clocks. In this approach, instead of accessing a global shared cache, each vector clock accesses its own vector clock history cache, which prevents all threads to wait for the global vector clock history cache to become suitable to be accessed.

For each vector clock, a list to hold the previous comparisons is required. Moreover, since the same vector clock could be accessed concurrently, a lock is required for accessing the list. These two requirements increase the total memory requirement of data race detection procedure. However, compared to the previous approach, in this approach different vector clock comparisons do not interfere. In other words, each vector clock's comparison with other vector clocks could be done concurrently. The concurrency achieved provides the performance increase compared to the previous solution. The algorithmic complexity of this optimization is $O(n)$, where n is the vector clock history size.

Input: Vector Clock vc_1, Vector Clock vc_2

Output: Comparison Result of vc_1 and vc_2

1: $LOCK(vc_1\_lock)$;
2: $comparison\_value = vc_1\_prev\_comparisons(vc_2)$;
3: **if** $comparison\_value == None$ **then** ▷ vc_1 and vc_2 has not been compared
4: $comparison\_value = compare(vc_1, vc_2)$
5: $vc_1\_prev\_comparisons(vc_2) = comparison\_value$
6: **end if**
7: $UNLOCK(vc_1\_lock)$;
8: return $comparison\_value$; ▷ Return the Comparison Result

Figure 5.3. Segment-based hybrid data race detection algorithm - store vector clock comparison history per each vector clock.

Our optimization is formalized in the algorithm in Figure 5.3. For each vector clock VC_i, a list $vc_i\_prev\_comparisons$ and a lock vc_i_lock is required. On each comparison, it is firstly searched in the local cache $vc_i\_prev\_comparisons$. If the result is already added there, there is no need to make the comparison. Otherwise, the comparison is done and the result is added to $vc_i\_prev\_comparisons$.

5.2. Optimization 2: Multiple Accesses of a Single Variable in a Segment

In segment-based hybrid algorithm, whenever the same variable is accessed more than once in a segment, the accesses after the first access have no effect and can be omitted. Note that the access type must be the same. For instance, if the first access is a write access, consecutive write accesses can be omitted. In this section, we investigate how this optimization is applied to the algorithm. Consider the example in Figure 5.5. On each access of variable x, writer segment set of x must be updated. However, all the updates yield the same writer segment set, which is composed of segment 1 only. Therefore, it is safe not to instrument the second and its subsequent accesses.

Input: Thread t_i Generates A Memory Access Operation On Memory Address x
Output: Potential Data Race Detected Or Not

1: $s_i = CurrentSegment(t_i)$ ▷ Current segment of thread t_i is segment s_i
2: **if** $ACCESS(x, t_i) == WRITE(x, t_i)$ **then**
3: **if** $s_i \in WRST_x$ **then**
4: $return$; ▷ x Is Already Written By s_i
5: **else**
6: $WRITE(x, t_i)$ ▷ Call Actual Write Operation
7: **end if**
8: **else if** $ACCESS(x, t_i) == READ(x, t_i)$ **then**
9: **if** $s_i \in RDST_x$ **then**
10: $return$; ▷ x Is Already Read By s_i
11: **else**
12: $READ(x, t_i)$ ▷ Call Actual Read Operation
13: **end if**
14: **end if**

Figure 5.4. Segment-based hybrid data race detection algorithm - prevent multiple accesses of a single variable in a segment.

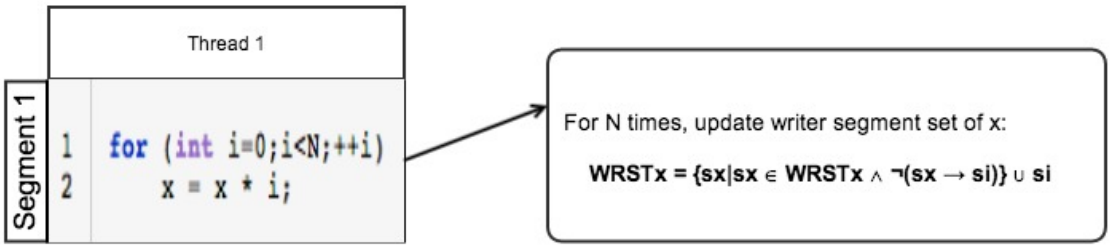

Figure 5.5. Disabling multiple access of a variable.

The implementation of the optimization is shown in the algorithm in Figure 5.4, which is slightly different than the algorithm in Figure 3.14 or the algorithm in

Figure 3.15. Whenever a memory address is read/written in a segment, it is checked whether the segment is already in reader/writer segment set of the variable. If that is the case, the algorithm concludes that the memory address is being accessed by the same segment one more time and discards the access. Otherwise, the algorithm in Figure 5.4 is executed as the algorithm in Figure 3.14 or the algorithm in Figure 3.15. Therefore, there is no extra memory requirement for implementing the optimization. The only requirement is the increased CPU utilization. The algorithmic complexity of this optimization is O(n), where n is the average number of segments in the segment sets.

5.3. Optimization 3: Limiting Total Number of Segments

In segment-based hybrid approach, many of the segments are unrelated to any of the potential data races. For instance, most of the segments do not access any of the shared variables. Or even if some shared variables are accessed, many of the segments are happens-before ordered or protected by a common lock. Therefore, discarding some of the segments may increase the performance while preserving the number of potential data races found.

In our implementation, we define a limit that identifies the maximum number of segments that can be utilized by our segment-based hybrid approach. Whenever the total number of active segments exceeds the maximum number, we discard a previously created segment and remove that segment from any of the segment sets that it is present. We utilize a FIFO queue for choosing which segment to be discarded. Consider the algorithm in Figure 5.6, when a segment is created, it is pushed to a FIFO queue irrespective of which thread executes it. If the maximum number is exceeded, the first segment in the queue is popped and discarded. It is important that the maximum number should be determined exclusively for each application. In our experiments, we choose the maximum number relative to the total segment count in original executions.

Input: Thread t_i Starts A New Segment s_i

Output: A New Segment s_i

```
1: newSegment = newSegment();                          ▷ Create Segment
2: allSegments.push(newSegment);                       ▷ Push To The Queue
3: if allSegments.size() > limit then                  ▷ Check Size Exceeds Limit
4:     tmpSegment = queue.popFirstElement();
5:     tmpSegment.destroy();                           ▷ Pop The First Segment And Destroy
6: end if
7: return newSegment;                                  ▷ Return The Created Segment
```

Figure 5.6. Segment-based hybrid data race detection algorithm - limiting total number of segments.

Although discarding some of the segments increases the performance of data race detection, it may lead to missing some of the potential data races since the discarded segment can be a part of a potential data race in the execution. We utilize a FIFO queue for limiting the number of segments. The segments that are at the back of the queue are executed before compared to segments that are at the front of the queue. Thus, when a segment is popped from the FIFO queue, it is going to be an old segment with respect to the execution time. The older segments in the execution are generally less likely to be a part of a potential data race among the other segments. The reason is that generally these segments have already been removed from most of the segment sets by the algorithm due to happens-before relations observed. Thus, they are generally less likely to be a part of a potential data race compared to the newer segments in the execution. In the worst case, the algorithmic complexity of this optimization is O(n), where n is the total number of segment sets in the execution.

5.4. Optimization 4: Proportional Detection of Data Races

It is widely accepted that dynamic data race detection requires excessive processing power and memory. This prevents dynamic data race detection tools to be

utilized in real deployed environments. However, the software that is deployed to real environments are suitable for data race detection since they are executed with plenty of different user inputs and quite different thread interleaving for a very long time. These factors increase the probability of observing potential data races compared to isolated software test environments. Therefore, taking advantage of data race detection tools on real deployed software is often desirable.

Input: Thread t_i Generates A Memory Access Operation On Memory Address x With Sample Rate $sample\_rate$

```
1: r = rand(0, 100);                          ▷ Generate Random Integer
2: if r <= sample_rate then
3:     ACCESS(x, t_i)                          ▷ Instrument Access
4: else
5:     return;                                 ▷ Do Not Instrument Access
6: end if
```

Figure 5.7. Segment-based hybrid data race detection algorithm - proportional detection of data races.

PACER [23] proposes proportional detection of data races. It makes a proportionality guarantee by detecting data races at a rate equal to the sampling rate. In our implementation we do not make such a guarantee. Our sampling approach is simpler. On each memory access, according to the sampling rate we decide whether to instrument the access or not. The algorithm is described in Figure 5.7. A random integer is generated between 0 and 100. If the generated integer is smaller than the sampling rate, we execute the instrumentation for the memory access. In the end, the percentage of instrumented memory accesses converge to the sampling rate.

There are some important requirements of implementing such an optimization. Firstly, it is expected that the random function outputs randomly even if it generates random integers as much as the total number of memory accesses of a single application, which might be a big number. Secondly, it is expected that random number

generation never slows down. These two requirements are not satisfied by the *random* functions provided by the standard math libraries. Therefore, we utilized one of the implementations of [27], which is a pseudo-random number generator algorithm.

6. EXPERIMENTS

In this section, we describe our implementation and experiments on dynamic data race detection. We implemented three different dynamic data race detection algorithms, lockset algorithm, happens-before algorithm and segment-based hybrid algorithm which are described in Chapter 3. We also show the results of our optimizations described in Chapter 5.

6.1. Implementation

In our implementation of three different dynamic data race detection algorithms, we utilized the same implementation for the common parts as much as possible. For instance, we use the same implementation of vector clocks for both the hybrid algorithm and happens-before algorithm. Similarly, we used the same implementation of locksets for both the hybrid algorithm and lockset algorithm. Moreover, we use the same dynamic binary instrumentation API function calls for all of the three implementation. These are required for a fair comparison of the three implementations.

All the experiments were performed on a PC running Linux with a 4 cores CPU of 2.27GHz and 32GB of memory. For each experiment, we ran it 10 times and used average of the results.

6.1.1. Benchmarks

We tested our dynamic data race detection implementations with 8 different applications from PARSEC benchmark [28], Firefox [29], pbzip2 [30] and httpd [31].

PARSEC is a benchmark suite composed of multithreaded applications. These applications can be executed with different input sizes such as simsmall, simmedium, simlarge and native. We choose to use simsmall input set. For each application, the execution time without any instrumentation and the number of concurrent threads

Table 6.1. Benchmark applications.

Applications		
Benchmark	Application	Description
PARSEC	bodytrack	Body tracking of a person
	canneal	Simulated annealing for optimization
	fluidanimate	Fluid dynamics for animation with (SPH) method
	freqmine	Frequent itemset mining
	streamcluster	Online clustering of an input stream
	swaptions	Pricing of a portfolio of swaptions
	vips	Image processing
	x264	H.264 video encoding
	Firefox	A popular web browser
	pbzip	Parallel implementation of bzip2 compression
	httpd	Apache HTTP server project

is described in Table 6.2. For Firefox, each test is done by starting a new instance of Firefox and opening "www.google.com" in a single tab. For pbzip2, we provide 250 MB text files as input to be compressed. The text files are randomly generated. Moreover, we choose to run pbzip2 with 16 threads concurrently. For apache httpd server, experiments are executed with 50 concurrent clients, with each client making 500 requests and each request is responded with a 8 KB static web page.

For the Firefox experiments, execution times of the experiments cannot be measured. The reason is that Firefox does not have any interface to query whether it has finished loading a page or not. Thus, for each experiment of Firefox, we provide enough time to finish the data race detection procedure and only consider the potential data races that are found. Note that, it is possible to run Firefox tests manually, by running Firefox, waiting for the page to be loaded and measuring its execution time via a chronometer. However, this would not be reliable. Therefore, we choose not to measure its execution time and exclude it from the experiments based on execution time.

Table 6.2. Benchmark information.

Application	Exe. Time(sec)	Thread Count	Memory Address	Segment Count
x264	0.18	15	4,579,993	6,225
freqmine	0.27	15	18,525,725	1,350
vips	0.22	18	22,883,398	32,044
swaptions	0.14	4	47,837	320
bodytrack	0.18	5	5,803,300	8,702
fluidanimate	0.25	4	1,303,047	2,163,828
streamcluster	0.25	8	166,555	163,641
canneal	1.90	4	3,012,749	1,380
pbzip2	1.40	16	1,050,268	157,60
httpd	26.11	22	251,943	229,900

6.1.2. Instrumentation

Dynamic analysis and the term instrumentation are mentioned in Chapter 1. In this section we present details of our dynamic binary instrumentation in PIN [2]. It is a framework for dynamic binary instrumentation developed by Intel for instrumenting C/C++ applications on the fly.

Figure 6.1 shows the architecture of the PIN framework. It consists of three main components. Firstly, PIN application consists of a virtual machine, a code cache, and an instrumentation API. The virtual machine includes a just-in-time compiler, an emulator and a dispatcher. However, our main consideration is on the second component of the PIN framework, which is Pintool. Pintools can be thought of as plugins that can modify the code generation process inside PIN. Third component is the application that is instrumented. All of the three components of PIN framework live in the same address space which enables the Pintool to access the application's data, including file descriptors and other process specific information.

There are two instrumentation modes that PIN supports: JIT mode and probe mode. Probe mode is intended for rewriting functions whereas JIT mode is intended

for instrumenting instructions or routines. In JIT mode the just-in-time compiler recompiles all code and inserts instrumentation, while in probe mode only some jump instructions added before and/or after a function for executing the instrumentation code. Since probe mode executes the code and the instrumentation natively, it supports a limited feature set of PIN, however, its performance is much better than JIT mode.

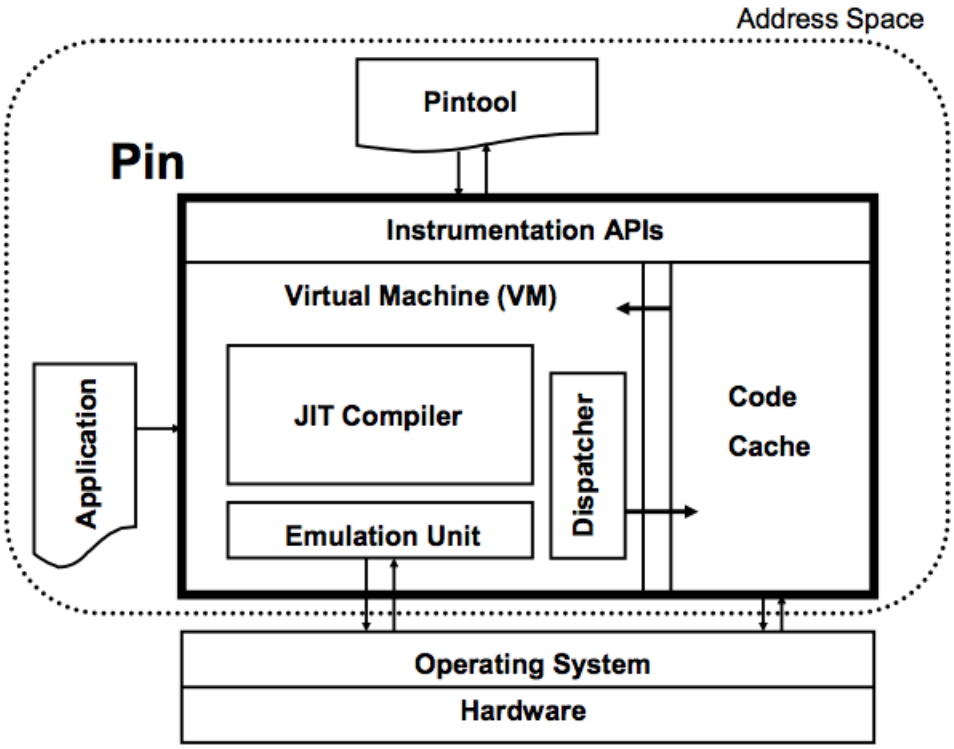

Figure 6.1. PIN software architecture [2].

In our implementation, we utilized JIT mode. It allows instrumentation to be done at different granularities such as instruction, trace, image or routines. We used routine instrumentation for the synchronization function calls. For instance, a callback is inserted after *pthread_mutex_lock* function call so that our hybrid algorithm updates the caller thread's writer lockset. For memory accesses we used trace instrumentation. Firstly, instruction instrumentation enables to insert one analysis call for every instruction executed. In fact, instruction instrumentation could be useful for inserting an analysis call for every read and write instruction so that data race detection algo-

rithms' memory access algorithms could be executed. However, reducing the number of analysis calls results in efficient instrumentation. Therefore, instead of instruction instrumentation we prefer trace instrumentation.

A trace usually begins at the target of a taken branch and ends with an unconditional branch, including calls and returns. It is guaranteed that a trace has a unique entrance and one or more exits. A trace is composed of a sequence of Basic Block (BBL). A BBL is a single entrance and single exit sequence of instructions. Thus, by trace instrumentation we firstly iterate over BBLs that forms the trace that is instrumented. Then, for each BBL we iterate over instructions and call a function for each read or write instruction that is executed. The number of analysis calls is reduced from the executed instruction count to the number of executed traces. Therefore, in our implementation, performance of trace instrumentation is significantly better than instruction instrumentation. Another advantage of using trace instrumentation in our implementation is that, it is much easier to identify segments in segment-based hybrid algorithm. As discussed in Chapter 4, each trace is mapped to a segment in the implementation of hybrid algorithm.

6.1.3. Experimental Results

In this section, our experimental results are examined. Similar to most of the data race detection implementations such as [15, 20, 22] the results are compared from two different perspectives, execution time overhead and potential data races found. It is not appropriate to compare our experimental results with other work's experimental results since there are plenty of different parameters that might affect the experimental results. Differences in one of the parameters might yield completely divergent results. Some of the parameters can be listed as below.

- Different DBI tools.
- Different API function calls from the same DBI.
- Handling of data races in arrays.

- Handling of dynamic memory allocation/deallocation(malloc / new / calloc / realloc/ free / delete).
- Instrumentation coverage (Only main application or main application + library function calls).
- Synchronization operations instrumentation (Locks / Condition Variables / Barriers / Semaphores / Timed Synchronization).

6.1.3.1. Execution Overhead. When the results of the three implementations are compared with respect to their execution times, it would be reasonable to measure the overhead due to the binary instrumentation. Therefore, we created an *Empty* implementation, which does the same instrumentation with all three implementations, however, it does not make any computations or analysis related to data race detection. In Table 6.3, the values show the slow downs calculated with respect to the *Empty* instrumentation. For most of the applications and on the average, the slow downs are arranged as $HB > Hybrid > Lockset$, which is expected.

Table 6.3. Benchmark performance slow downs.

Application	Empty	Lockset	Hybrid	Happens-Before
x264	1.00x	13.80x	20.26x	25.20x
freqmine	1.00x	117.93x	332.41x	261.56x
vips	1.00x	103.69x	250.21x	315.43x
swaptions	1.00x	72.43x	134.98x	202.15x
bodytrack	1.00x	18.67x	31.24x	50.37x
fluidanimate	1.00x	89.42x	85.10x	113.52x
streamcluster	1.00x	75.28x	133.05x	178.56x
canneal	1.00x	16.41x	20.88x	26.71x
pbzip2	1.00x	41.92x	44.27x	50.30x
httpd	1.00x	13.32x	24.11x	26.75x
average	1.00x	56.26x	107.71x	125.04x

Lockset algorithm's slowdown is less than half of the happens-before (HB) and *Hybrid* implementations. This is acceptable since heavy memory and execution overhead of vector clocks are not present in the *Lockset* algorithm. On the average, happens-before algorithm is about 15% slower than the *Hybrid* implementation. Detailed comparisons of the two algorithms are presented in Section 6.4.

6.1.3.2. Potential Data Races. In this section, we show our experimental results in terms of detected number of potential data races in benchmarks.

Table 6.4. Detected potential number of data races in benchmarks.

Application	Lockset	Hybrid	Happens-Before
x264	184	27	1
freqmine	76	23	7
vips	174	64	0
swaptions	0	0	0
bodytrack	47	5	4
fluidanimate	281	35	0
streamcluster	19	14	0
canneal	1	1	0
pbzip2	29	23	0
httpd	1777	1151	0
Firefox	81	7	0
average	240	121	1

On the average detected number of potential data races are ordered as $Lockset > Hybrid > HB$, which is shown in Table 6.5. These results met our expectations. As discussed in Chapter 3, lockset detectors produce too many false positives. On the contrary, happens-before detectors do not produce any false positives. Moreover, they even miss some of the potential data races. Hybrid algorithm poses characteristics from both approaches.

Table 6.5. Number of Thread Counts For Different Experiments.

Application	Original Execution	Experiment-1	Experiment-2	Experiment-3
x264	15	18	20	32
freqmine	15	18	20	32
fluidanimate	4	6	8	12
canneal	4	6	8	12
pbzip2	16	12	20	32

For some of the applications, it is possible to change the number of the threads that run concurrently without changing the inputs. Such applications include *x264*, *freqmine*, *fluidanimate*, *canneal* and *pbzip2*. For these applications, we performed three different experiments with different number of threads as shown in Table 6.5. Though execution time depends on the thread count, the total number of potential data races detected are not affected.

6.2. Results of Optimizations on Segment-Based Hybrid Algorithm

We implemented some optimizations on segment-based hybrid algorithm which are described in Chapter 5. In this section, we are going to explore the implementation results.

6.2.1. Optimization 1: Storing Vector Clock Comparison History Cache

As discussed in Chapter 5, the maximum number of history kept for each vector clock has an impact on the performance. The algorithm in Figure 5.4 displays our approach for vector clock history optimization and Table 6.6 displays the effect of changing the maximum number of vector clock history on the execution time. As the cache size increases, the rate of successful searches in the cached history increases too. However, since the search space and thus the search time increases, the total performance does not vary much. On the average, when a single vector clock history cache is maintained for each vector clock, the execution time decreases 5%. On the average, the maximum improvement is gained when the history size is 50, the execution

times decrease 7%.

Table 6.6. Vector clock history size vs execution time decrease.

History Size	0	1	2	5	10	20	50	100	250	1000
x264	1.0x	1.05x	0.93x	0.93x	0.93x	0.95x	0.96x	0.96x	1.04x	0.96x
freqmine	1.0x	0.88x	0.9x	0.89x	0.9x	0.88x	0.89x	0.88x	0.86x	0.88x
vips	1.0x	0.9x	0.96x	0.95x	0.86x	0.92x	0.97x	0.88x	0.97x	0.97x
swaptions	1.0x	1.34x	1.41x	1.42x	1.38x	1.36x	1.19x	1.44x	1.39x	1.07x
bodytrack	1.0x	0.93x	0.92x	0.9x	0.92x	0.92x	0.89x	0.94x	0.95x	0.95x
fluidanimate	1.0x	0.83x	0.84x	0.83x	0.81x	0.81x	0.82x	0.81x	0.83x	0.83x
streamcluster	1.0x	0.78x	0.8x	0.78x	0.78x	0.78x	0.78x	0.77x	0.78x	0.77x
canneal	1.0x	0.97x	0.96x	0.98x	1.01x	0.99x	0.96x	0.97x	0.99x	0.97x
pbzip2	1.0x	1.0x	0.99x	1.0x	1.01x	1.01x	1.0x	1.01x	1.0x	0.99x
httpd	1.0x	0.86x	0.84x	0.87x	0.85x	0.86x	0.84x	0.87x	0.84x	0.86x
average	1.0x	0.95x	0.96x	0.95x	0.94x	0.95x	0.93x	0.95x	0.97x	0.93x

For some of the applications such as *swaptions*, keeping vector clock history slows down the execution, independent of the history size. The explanations that lead to the slow down is the unsuccessful searches in the vector clock history utilize CPU so much that the gain of successful searches cannot compensate the unsuccessful ones.

6.2.2. Optimization 2: Multiple Accesses of a Single Variable in a Segment

As discussed in Chapter 5, whenever the same variable is accessed more than once in a segment, the accesses after the first access have no effect and can be omitted. Table 6.7 shows the performance improvement of applying the algorithm in Figure 5.4 for the benchmarks.

First of all, the total number potential data races found do not vary for any of the applications by applying the optimization. The observation from the performance perspective is that some of the applications such as *streamcluster*, *x*264 and *httpd* is processed 10% to 30% faster. On the contrary, the gain of the optimization for some applications such as *bodytrack* and *swaptions* is not adequate to compensate the overhead of the optimization. The slow down for these applications is approximately 10%. On the average, the optimization improves performance by 4%-5%.

Table 6.7. Multiple accesses of a single variable in a segment.

application	Not Enabled	Enabled
x264	1.00x	0.90x
freqmine	1.00x	0.97x
vips	1.00x	0.99x
swaptions	1.00x	1.13x
bodytrack	1.00x	1.10x
fluidanimate	1.00x	0.96x
streamcluster	1.00x	0.89x
canneal	1.00x	0.98x
pbzip2	1.00x	1.0x
httpd	1.00x	0.70x
average	1.00x	0.96x

6.2.3. Optimization 3: Limiting Total Number of Segments

As discussed in Chapter 5, many of the segments are unrelated to any of the potential data races. Our approach for limiting the segments are described in the algorithm in Figure 5.6. By limiting the total number of segments in the execution, it is possible to discard some of the segments that appear in none of the races. However, this optimization may lead to missing some of the potential data races.

In Table 6.8, the effect of limiting the total segment count on the execution time and the potential data races is displayed. Although each application executes different number of segments during execution, the parameter that is altered is the percentage of maximum segments with respect to the all segments in the original execution. For instance, 50% means that the total segment count is limited to half of the original execution for that application.

Table 6.8. Total segment count limitation. (a) Between 100% - 18%, (b) between 12% - 1%.

(a)

	100%		50%		32%		18%	
Application	Time	Race	Time	Race	Time	Race	Time	Race
x264	1.0x	27	0.99x	27	0.93x	27	0.96x	27
freqmine	1.0x	23	0.47x	23	0.45x	23	0.41x	23
vips	1.0x	64	0.96x	63	0.96x	59	0.95x	55
swaptions	1.0x	0	1.0x	0	0.94x	0	0.64x	0
bodytrack	1.0x	5	0.88x	3	0.78x	0	0.8x	0
fluidanimate	1.0x	35	0.89x	34	0.83x	34	0.79x	34
streamcluster	1.0x	14	0.84x	14	0.77x	14	0.71x	14
canneal	1.0x	1	1.03x	1	0.94x	1	0.91x	1
pbzip2	1.0x	23	0.99x	23	0.99x	23	0.99x	20
httpd	1.0x	1151	0.91x	1121	0.84	1113	0.85x	1113
average	1.00x	134	0.9x	130	0.84x	129	0.8x	128

(b)

	12%		4%		1%	
Application	Time	Race	Time	Race	Time	Race
x264	0.94x	27	0.89x	26	0.76x	4
freqmine	0.4x	23	0.37x	19	0.94x	19
vips	0.87x	53	0.85x	38	0.72x	0
swaptions	0.64x	0	0.59x	0	0.9x	0
bodytrack	0.85x	0	0.77x	0	0.68x	0
fluidanimate	0.72x	34	0.81x	33	0.91x	22
streamcluster	0.68x	14	0.63x	14	0.54	14
canneal	0.94x	1	0.89x	1	0.97x	1
pbzip2	0.98x	17	0.98x	17	0.94	0
httpd	0.84x	1097	0.81x	1097	0.79x	56
average	0.79x	126	0.76x	124	0.81x	11

Evaluating the results from the perspective of data races, our expectations are met. That is, even the total segment count is limited to 4% of the original execution, most of the potential data races are detected. However, the execution time decrease is not as high as desired. The operation of destroying a segment requires too much computation since the segment must be removed from all of the segment sets that it is element of. This computation overhead decreases the gain of applying the optimization.

6.2.4. Optimization 4: Proportional Detection of Data Races

As discussed in Chapter 5, proportional detection of data races are useful for deployed software since the overhead of data race detection can be adjusted by adjusting the proportion. The algorithm in Figure 5.7 displays our proportional approach in segment-based hybrid algorithm.

Results in Table 6.9 show that execution times and detected data races roughly converge to the sampling rate on the average, up to 16% sampling rate. When the sampling rate of memory accesses is decreased more, due to the overhead of creating and managing segments, performance does not converges to the sample rate.

For *pbzip*, the execution time is not converging to the sampling rate. The reason is that *pbzip* applications makes too many file operations, which is the bottleneck. Thus, sampling memory accesses cannot yield the expected result.

6.3. Multiple Optimizations

In this section we display experimental results of applying multiple optimizations. We do not apply optimization 3 and optimization 4 together. The reason is that both of the optimizations miss some of the potential data races and affect the performance. Therefore, it would be difficult to infer the effect of each optimization on the results.

We firstly show the results of applying optimization 1 and optimization 2 together. None of these optimizations affect the potential data races detected. Thus, we can observe the highest performance with lossless results in terms of potential data races detected.

Next, we add optimization 3 and optimization 4 to the previously executed optimization separately. These two multiple optimizations aim to provide a trade-off between the number of potential data races detected and the performance of data race detection.

Table 6.9. Proportional detection of data races. (a) Between 100% - 50%, (b) between 16% - 2%

(a)

Sample Rate	100%		71%		50%	
Application	Time	Race	Time	Race	Time	Race
x264	1.0x	27	0.76x	23	0.57x	19
freqmine	1.0x	23	0.32x	23	0.17x	23
vips	1.0x	64	0.76x	53	0.5x	35
swaptions	1.0x	0	0.44x	0	0.48x	0
bodytrack	1.0x	5	0.88x	1	0.53x	1
fluidanimate	1.0x	35	0.7x	28	0.63x	27
streamcluster	1.0x	14	0.66x	13	0.53x	12
canneal	1.0x	1	0.81x	1	0.65x	1
pbzip2	1.0x	23	0.98x	13	0.87x	6
httpd	1.0x	1151	0.65x	797	0.43x	478
average	1.0x	134	0.70x	95	0.54x	60

(b)

Sample Rate	16%		5%		2%	
Application	Time	Race	Time	Race	Time	Race
x264	0.27x	12	0.16x	9	0.14x	3
freqmine	0.06x	12	0.02x	11	0.02x	4
vips	0.18x	17	0.11x	1	0.05x	0
swaptions	0.13x	0	0.13x	0	0.1x	0
bodytrack	0.18x	0	0.09x	0	0.07x	0
fluidanimate	0.18x	12	0.08x	2	0.06x	2
streamcluster	0.15x	5	0.03x	2	0.01x	0
canneal	0.39x	0	0.29x	0	0.27x	0
pbzip2	0.75x	1	0.76x	0	0.77x	0
httpd	0.09x	111	0.15x	23	0.03x	15
average	0.24x	17	0.18x	5	0.15x	2

6.3.1. Multiple Optimization 1: Optimization 1 and Optimization 2

Our first multiple optimization aims to increase the performance without changing the potential data races detected. Therefore, we utilized the optimizations that do not affect the precision of hybrid data race detection: optimization 1 and optimization 2.

Table 6.10. Optimization 1 and Optimization 2.

History Size	0	1	2	5	10	20	50	100	250	1000
x264	0.92x	0.95x	0.88x	0.87x	0.88x	0.90x	0.89x	0.89x	0.99x	0.88x
freqmine	0.98x	0.85x	0.87x	0.87x	0.86x	0.85x	0.83x	0.84x	0.84x	0.85x
vips	0.98x	0.9x	0.96x	0.95x	0.86x	0.92x	0.97x	0.88x	0.97x	0.97x
swaptions	1.16x	1.44x	1.48x	1.47x	1.42x	1.41x	1.33x	1.49x	1.42x	1.17x
bodytrack	1.09x	0.99x	0.98x	0.99x	0.95x	0.99x	0.98x	1.02x	1.03x	1.02x
fluidanimate	0.96x	0.78x	0.80x	0.79x	0.77x	0.77x	0.78x	0.77x	0.79x	0.81x
streamcluster	0.91x	0.68x	0.7x	0.68x	0.68x	0.68x	0.68x	0.67x	0.68x	0.69x
canneal	0.99x	0.95x	0.95x	0.96x	0.99x	0.97x	0.94x	0.93x	0.97x	0.95x
pbzip2	1.02x	1.0x	0.98x	0.99x	0.99x	0.99x	1.0x	1.00x	0.98x	0.98x
httpd	0.71x	0.61x	0.61x	0.63x	0.62x	0.66x	0.67x	0.64x	0.64x	0.69x
average	0.97x	0.91x	0.92x	0.92x	0.90x	0.91x	0.91x	0.91x	0.93x	0.91x

Table 6.10 shows the performance variation relative to vector clock history size when optimization 2 is enabled. The results display that the increase in the performance is almost equal to the addition of applying each optimization separately.

The multiple optimization discussed here aims to achieve best performance without sacrificing the precision. On the average, when the vector clock history size is set to 10, the highest performance is achieved.

6.3.2. Multiple Optimization 2: Optimization 1, Optimization 2 and Optimization 3

In this section we display the effect of optimization 3 while *Multiple Optimization 1* is already applied with vector clock history is set to 10.

Table 6.11 displays the results of applying optimization 1, optimization 2 and optimization 3 together. Time values are relative to the execution of segment-based hybrid algorithm with no optimization applied. For instance, when the total number of segments are limited to 50%, the performance is increased 17% while only 4 of the potential data races are missed.

Table 6.11. Results of Optimization 1, Optimization 2 and and Optimization 3. (a) Limit Between 100% - 18%, (b) limit between 12% - 1%.

(a)

Limit.	100%		50%		32%		18%	
Application	Time	Race	Time	Race	Time	Race	Time	Race
x264	0.91x	27	0.90x	27	0.86x	27	0.86x	27
freqmine	0.88x	23	0.41x	23	0.40x	23	0.36x	23
vips	0.87x	64	0.88x	63	0.88x	59	0.87x	55
swaptions	1.44x	0	1.40x	0	1.34x	0	0.98x	0
bodytrack	0.97x	5	0.83x	3	0.73x	0	0.77x	0
fluidanimate	0.81x	35	0.67x	34	0.68x	34	0.66x	34
streamcluster	0.72x	14	0.61x	14	0.59x	14	0.57x	14
canneal	1.0x	1	1.01x	1	0.92x	1	0.90x	1
pbzip2	0.99x	23	0.98x	23	1.00x	23	0.98x	20
httpd	0.68x	1151	0.62x	1121	0.58	1113	0.61x	1113
average	0.93x	134	0.83x	130	0.80x	129	0.76x	128

(b)

Limit.	12%		4%		1%	
Application	Time	Race	Time	Race	Time	Race
x264	0.88x	27	0.80x	26	0.66x	4
freqmine	0.31x	23	0.29x	19	0.88x	19
vips	0.79x	53	0.77x	38	0.61x	0
swaptions	0.95x	0	0.87x	0	1.32x	0
bodytrack	0.80x	0	0.71x	0	0.64x	0
fluidanimate	0.61x	34	0.59x	33	0.79x	22
streamcluster	0.55x	14	0.44x	14	0.50	14
canneal	0.95x	1	0.91x	1	0.98x	1
pbzip2	0.98x	17	0.97x	17	0.96	0
httpd	0.58x	1097	0.57x	1097	0.68x	56
average	0.74x	126	0.69x	124	0.80x	11

6.3.3. Multiple Optimization 3: Optimization 1, Optimization 2 and Optimization 4

With the same motivation as *Multiple Optimization 2*, we display the effect of optimization 4 while *Multiple Optimization 1* is already applied with vector clock history is set to 10.

Table 6.12 displays the results of applying optimization 1, optimization 2 and optimization 4 together. Time values are relative to the execution of segment-based hybrid algorithm with no optimization applied. The results are compatible with the results of optimization 4. On the average, the results are 4% to 8% faster and none of the potential data races are missed compared to optimization 4.

Table 6.12. Results of Optimization 1, Optimization 2 and and Optimization 4. (a) Sample rate between 100% - 50%, (b) sample rate between 16% - 2%.

(a)

Sample Rate	100%		71%		50%	
Application	Time	Race	Time	Race	Time	Race
x264	0.92x	27	0.66x	23	0.49x	19
freqmine	0.88x	23	0.21x	23	0.13x	23
vips	0.87x	64	0.67x	53	0.49x	35
swaptions	1.54x	0	0.74x	0	0.67x	0
bodytrack	1.01x	5	0.91x	0	0.52x	1
fluidanimate	0.82x	35	0.56x	28	0.44x	27
streamcluster	0.74x	14	0.42x	13	0.33x	12
canneal	1.02x	1	0.82x	1	0.65x	1
pbzip2	1.04x	23	0.97x	13	0.81x	6
httpd	0.71x	1151	0.41x	797	0.29x	478
average	0.96x	134	0.64x	95	0.48x	60

(b)

Sample Race	16%		5%		2%	
Application	Time	Race	Time	Race	Time	Race
x264	0.17x	12	0.11x	9	0.10x	3
freqmine	0.04x	12	0.02x	11	0.06x	4
vips	0.12x	17	0.07x	1	0.06x	0
swaptions	0.23x	0	0.21x	0	0.19x	0
bodytrack	0.16x	0	0.08x	0	0.07x	0
fluidanimate	0.08x	12	0.06x	2	0.06x	2
streamcluster	0.12x	5	0.04x	2	0.05x	0
canneal	0.37x	0	0.28x	0	0.31x	0
pbzip2	0.74x	1	0.72x	0	0.75x	0
httpd	0.08x	111	0.12x	23	0.04x	15
average	0.21x	17	0.17x	4	0.17x	2

6.4. Performance Comparison of Happens Before Algorithm and Segment-Based Hybrid Algorithm

In this section some of the performance metrics of hybrid algorithm and happens-before algorithm are compared. Firstly, the average number of vector clock operations per memory access is going to be compared. Then, memory requirements of the two algorithms are compared.

6.4.1. Vector Clock Operation Performance

Firstly, we briefly revisit happens-before algorithm. In happens-before algorithm, in order to identify conflicting accesses, there should exist two vector clocks for each

memory address, $x_r.VC$ and $x_w.VC$. For any memory address x, $x_r.VC(i)$ identifies the last read of x by thread t_i. In the same manner, $x_w.VC(i)$ identifies the last write of x by thread t_i. Race check on each memory access is done as follows:

- If the access is write, make two happens-before comparison with the thread's vector clock, one with $x_r.VC$ and the other with $x_w.VC$.
- If the access is read, make one happens-before comparison with the thread's vector clock, with $x_w.VC$.

The above information indicates that, on each memory access, happens-before data race detector is expected to make one or two vector clock comparisons. The average is expected to be in the range of 1 to 2. As the proportion of reads in the execution increases, the average number of vector clock comparisons per memory access is expected to decrease. For the segment-based hybrid approach, the calculation is somewhat different. As discussed in Chapter 3, the comparisons are done in two different points:

- Before adding a segment to a segment set, check if the segment happens-before the other segments in the set.
- Checking for a race among the elements of reader segment set and writer segment set of a memory address.

The number of vector clock operations depends on the number of segments in the segment set on each memory access. Thus, the average number of vector clock operations could not be estimated statically. In fact, even for happens-before case, the exact average number of vector clock comparison could not be estimated statically. The calculation is done as follows for both of the algorithms. Declare two variables *memory_access_count* and *vector_clock_comparison_count*.

- On each memory access, increment the value of *memory_access_count*.
- On each vector clock comparison, increment the value of *vector_clock_comparison_count*.

Then, after application finishes, calculate the average, *vector_clock_compasion_per_access*, as follows

$$vector\_clock\_compasion\_per\_access = \\ vector\_clock\_comparison\_count / \\ memory\_access\_count \tag{6.1}$$

Table 6.13 shows the calculated average number for both segment-based hybrid approach and happens-before approach. As expected, the results of happens-before approach is between 1 and 2 for all applications. They are mostly close to 1 since the proportion of the read accesses are generally much more than the write accesses, thus the averages are converging to 1. On the segment-based hybrid approach, the average number of vector clock operations per memory access ranges from 0.34 to 22.80 for different applications. Lets examine the expected values for some the sample conditions.

- When the memory is accessed for the first time, there is no need to make any vector clock comparisons, since segment sets are empty, the segment is added to one of the reader or writer segment set depending on the memory access type.
- If segment size is already one, there are four possibilities:
  (i) If reader segment set size equals to one and the access is read access: Only 1 vector lock comparison is done while updating the reader segment set.
  (ii) If reader segment set size equals to one and the access is write access: 1 vector clock comparison is done for updating the reader segment set. Also, 0 or 1 vector clock comparison is done for checking the race condition, depending on the reader segment set size after updating the set. Thus, in total 1 or 2 vector clock operations are done.
  (iii) If writer segment set size equals to one and the access is read access: No need to do vector clock operation for updating the segment sets. 1 vector clock operation is needed for checking the race condition. Thus, in total 1

vector clock operation is needed.

(iv) If writer segment set size equals to one and the access is write access: 1 vector clock comparison is done for updating the writer segment set.

(v) Thus, on average, the vector clock comparison count is in between 1 and 2 if segment size is 1 in any of the segment sets.

- If both reader and writer segment set sizes are equal or greater than one, the average number of vector clock operations would be greater than 2.

Table 6.13. Average number of vector clock comparison per memory access.

Application	Hybrid	Happens Before
x264	2.12	1.07
freqmine	14.87	1.16
vips	7.99	1.11
swaptations	4.78	1.04
bodytrack	2.85	1.13
fluidanimate	1.85	1.08
streamcluster	2.58	1.07
canneal	2.54	1.34
Firefox	0.34	1.07
pbzip2	0.99	1.07
httpd	22.80	1.37
average	5.70	1.13

As the number of accesses to a single memory address increases, the average number is expected to increase. However, this does not mean that the average number has to increase. The increase depends on the number of concurrent access to the memory address. If the accesses are concurrent, the size of segment sets get bigger, and the average number of vector clock operations per memory access increases. To sum up, the average number of vector clock comparisons is 1.13 for happens-before approach, whereas, the value becomes 5.70 for segment-based hybrid approach. Therefore, we can conclude that happens-before approach, on the average, performs better in terms

of the average number of vector clock operations per memory access.

6.4.2. Memory Performance

In this section, memory requirements of lockset, happens-before and segment-based hybrid algorithms are examined. Table 6.14 shows memory requirements for each application. The results show that hybrid detector's memory requirement is almost 40% less than happens-before detector and 30% more than lockset detector. These results are expected since utilization of vector clocks increases memory requirements remarkably. Happens-before detectors are entirely based on vector clocks, lockset detectors do not utilize vector clocks and hybrid detectors are partially based on vector clocks.

Table 6.14. Memory requirements for different algorithms (MB).

Application	Locket	Hybrid	Happens Before
x264	794	884	2785
freqmine	1444	2201	3873
vips	1502	2220	3564
swaptations	304	574	744
bodytrack	752	1408	2062
fluidanimate	674	886	1240
streamcluster	513	590	574
canneal	750	904	1766
pbzip2	721	910	1104
Firefox	1066	1271	1971
httpd	1562	1990	3244
average	916	1258	2084

7. CONCLUSIONS AND FUTURE WORK

In this study, we present a comprehensive study on dynamic data race detection algorithms. First of all, we investigated three dynamic data race detection algorithms: happens-before, lockset and hybrid algorithms. Specifically, segment-based hybrid algorithm is the main concern of our study.

We implemented these three algorithms using dynamic binary instrumentation. Experiments are performed with 8 applications from PARSEC benchmark, Firefox browser, Apache httpd web server and pbzip2,a parallel compression tool. Performance of happens-before detector is the worst in terms of both memory requirements and execution time overhead. On the average, execution slow down is 125x for happens-before detector, 107x for hybrid detector and 56x for lockset detector. Memory requirements are 2084M for happens-before detector, 1258M for hybrid detector and 916M for lockset detector. Potential number of data races detected by happens-before, hybrid and lockset detectors are 1, 121, 240 respectively.

Although there exits plenty of work in the literature that concern dynamic data race detection algorithms, the concept of segment and segment-based hybrid algorithms are not examined in detail. The main contribution of this thesis is the detailed examination of segment-based hybrid algorithm. We proposed four different improvements on segment-based hybrid algorithm. These improvements are:

- Storing Vector Clock Comparison History Cache
- Multiple Accesses of a Single Variable in a Segment
- Limiting Total Number of Segments throughout Execution
- Proportional Detection of Data Races

The first two improvements do not alter the number of data races that are detected in any of the benchmark applications. However, the execution time is decreased 7% and 4%, respectively, on the average by applying the improvements. The last two

improvements alter both the execution time and data races detected relative to specification of the input parameters. For instance, for the third improvement when the input parameter is set to 50%, execution time decreases 90% where detected number of potential races decreases to 98%.

We applied the combinations of the improvements. The combinations can be categorized in two types. First type of combinations increase the performance without scarifying the precision. The performance can be increased 10% for this type of combinations. Secondly, we offer a trade-off between the performance and the total number of data races detected. We can detect 97% of the potential data races with 16% increase in the performance.

We also compared happens-before algorithm and segment-based hybrid algorithm in terms of memory utilization and total number of vector clock comparisons. The comparison reveals that memory requirements of happens-before algorithm is more than segment-based hybrid algorithm, whereas, hybrid algorithm executes more vector clock comparisons on the average.

As another future work, the segment method can be applied to happens-before and FastTrack [22] algorithms. It is likely that their segment-based variations would yield better results in terms of performance without sacrificing the precision.

Another direction for future work could be a new viewpoint for instrumentation granularity. Currently, dynamic data race detection algorithms instrument every memory access. However, the number of memory accesses are considerably more than the number of segments. Thus, a new approach may ignore memory accesses and instruments only segments. Segments that are concurrent or not protected by common locks can be detected. Then, these detected segments could be further evaluated for conflicting memory accesses. Therefore, execution overhead of data race detection could be decreased sharply.

Recently, there have been several proposals for hardware-assisted data race detection such as SigRace [32] and HARD [33]. These algorithms detect potential data races with minimal overhead compared to software based detectors. Segment-based hybrid algorithm can be modified so that it can be applied to hardware-assisted data race detection for further increase in the performance.

REFERENCES

1. Savage, S., M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs", *ACM Transactions on Computer Systems*, Vol. 15, No. 4, pp. 391–411, 1997.

2. Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation", *ACM SIGPLAN Notices*, Vol. 40, No. 6, pp. 190–200, 2005.

3. Sutter, H., "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software", *Dr. Dobb's Journal*, Vol. 30, No. 3, pp. 202–210, 2005.

4. Lu, S., S. Park, E. Seo and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics", *SIGOPS Operating Systems Review*, Vol. 42, No. 2, pp. 329–339, 2008.

5. Netzer, R. H. B. and B. P. Miller, "What Are Race Conditions?: Some Issues and Formalizations", *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 1, pp. 74–88, 1992.

6. Kahlon, V., Y. Yang, S. Sankaranarayanan and A. Gupta, "Fast and Accurate Static Data-race Detection for Concurrent Programs", *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pp. 226–239, Springer-Verlag, Berlin, Heidelberg, 2007.

7. Voung, J. W., R. Jhala and S. Lerner, "RELAY: Static Race Detection on Millions of Lines of Code", *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pp. 205–214, ACM, New York, NY, USA, 2007.

8. Pratikakis, P., J. S. Foster and M. Hicks, "LOCKSMITH: Context-sensitive Correlation Analysis for Race Detection", *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pp. 320–331, ACM, New York, NY, USA, 2006.

9. Bala, V., E. Duesterwald and S. Banerjia, "Dynamo: A Transparent Dynamic Optimization System", *ACM SIGPLAN Notices*, Vol. 46, No. 4, pp. 41–52, 2011.

10. Nethercote, N. and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation", *ACM SIGPLAN Notices*, Vol. 42, No. 6, pp. 89–100, 2007.

11. Patil, H., C. Pereira, M. Stallcup, G. Lueck and J. Cownie, "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs", *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pp. 2–11, ACM, New York, NY, USA, 2010.

12. Ronsse, M. and K. De Bosschere, "RecPlay: A Fully Integrated Practical Record/Replay System", *ACM Transactions on Computer Systems (TOCS)*, Vol. 17, No. 2, pp. 133–152, 1999.

13. Ernst, M. D., "Invited Talk Static and Dynamic Analysis: Synergy and Duality", *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pp. 35–35, ACM, New York, NY, USA, 2004.

14. Pozniansky, E. and A. Schuster, "Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs", *ACM SIGPLAN Notices*, Vol. 38, No. 10, pp. 179–190, 2003.

15. Serebryany, K. and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice", *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pp. 62–71, ACM, New York, NY, USA, 2009.

16. *Pthread Library*, 2013, `http://pthread.org`, accessed at May 2014.

17. Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565, 1978.

18. Banerjee, U., B. Bliss, Z. Ma and P. Petersen, "A Theory of Data Race Detection", *Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '06, pp. 69–78, ACM, New York, NY, USA, 2006.

19. Marino, D., M. Musuvathi and S. Narayanasamy, "LiteRace: Effective Sampling for Lightweight Data-race Detection", *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pp. 134–143, ACM, New York, NY, USA, 2009.

20. O'Callahan, R. and J.-D. Choi, "Hybrid Dynamic Data Race Detection", *ACM SIGPLAN Notices*, Vol. 38, No. 10, pp. 167–178, 2003.

21. Christiaens, M. and K. De Bosschere, "TRaDe, a Topological Approach to On-the-fly Race Detection in Java Programs", *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pp. 15–15, USENIX Association, Berkeley, CA, USA, 2001.

22. Flanagan, C. and S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection", *Communications of the ACM*, Vol. 53, No. 11, pp. 93–101, 2010.

23. Bond, M. D., K. E. Coons and K. S. McKinley, "PACER: Proportional Detection of Data Races", *ACM SIGPLAN Notices*, Vol. 45, No. 6, pp. 255–268, 2010.

24. Smaragdakis, Y., J. Evans, C. Sadowski, J. Yi and C. Flanagan, "Sound Predictive Race Detection in Polynomial Time", *ACM SIGPLAN Notices*, Vol. 47, No. 1, pp. 387–400, 2012.

25. Jannesari, A., K. Bao, V. Pankratius and W. F. Tichy, "Helgrind+: An Efficient

Dynamic Race Detector", *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '09, pp. 1–13, IEEE Computer Society, Washington, DC, USA, 2009.

26. Downey, A. B., *The Little Book of Semaphores*, Green Tea Press, 2 edn., 2005, `http://greenteapress.com/semaphores/`.

27. Matsumoto, M. and T. Nishimura, "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator", *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, pp. 3–30, Jan.

28. Bienia, C., *Benchmarking Modern Multiprocessors*, Ph.D. Thesis, Princeton University, 2011.

29. *Mozilla Firefox 28.0*, 2014, `http://www.mozilla.org/`, accessed at May 2014.

30. Gilchrist, J., "Parallel Data Compression with bzip2", *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS '04, pp. 559–564, ACM, MIT Cambridge, USA, 2004.

31. *The Apache HTTP Server Project - 2.2.22*, 2014, `http://www.apache.org/`, accessed at May 2014.

32. Muzahid, A., D. Suárez, S. Qi and J. Torrellas, "SigRace: Signature-based Data Race Detection", *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 337–348, ACM, New York, NY, USA, 2009.

33. Zhou, P., R. Teodorescu and Y. Zhou, "HARD: Hardware-Assisted Lockset-based Race Detection", *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pp. 121–132, IEEE Computer Society, Washington, DC, USA, 2007.