Minimal Context-Switching Data Race Detection with Dataflow Tracking

Zheng Long, Li Yang, Xin Jie, Liu Hai–Feng, Zheng Ran, Liao Xiao–Fei, Jin Hai

View online: http://doi.org/10.1007/s11390–023–1569–7

Zheng L, Li Y, Xin J *et al.* Minimal context-switching data race detection with dataflow tracking. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 39(1): 211-226 Jan. 2024. DOI: 10.1007/s11390-023-1569-7

Minimal Context-Switching Data Race Detection with Dataflow Tracking

Long Zheng (郑　龙), *Member, CCF, ACM, IEEE*, Yang Li (李　洋), *Student Member, CCF*
Jie Xin (辛　杰), *Student Member, CCF*, Hai-Feng Liu (刘海峰), *Student Member, CCF*
Ran Zheng (郑　然), *Member, CCF, ACM, IEEE*
Xiao-Fei Liao* (廖小飞), *Senior Member, CCF, Member, IEEE*
and Hai Jin (金　海), *Fellow, CCF, IEEE, Life Member, ACM*

National Engineering Research Center for Big Data Technology and System, School of Computer Science and Technology Huazhong University of Science and Technology, Wuhan 430074, China

Services Computing Technology and System Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

Cluster and Grid Computing Laboratory, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

E-mail: longzh@hust.edu.cn; ly_winter@hust.edu.cn; jxin@hust.edu.cn; hfliu@hust.edu.cn; zhraner@hust.edu.cn
xfliao@hust.edu.cn; hjin@hust.edu.cn

Received May 7, 2021; accepted December 23, 2023.

Abstract Data race is one of the most important concurrent anomalies in multi-threaded programs. Emerging constraint-based techniques are leveraged into race detection, which is able to find all the races that can be found by any other sound race detector. However, this constraint-based approach has serious limitations on helping programmers analyze and understand data races. First, it may report a large number of false positives due to the unrecognized dataflow propagation of the program. Second, it recommends a wide range of thread context switches to schedule the reported race (including the false one) whenever this race is exposed during the constraint-solving process. This ad hoc recommendation imposes too many context switches, which complicates the data race analysis. To address these two limitations in the state-of-the-art constraint-based race detection, this paper proposes DFTracker, an improved constraint-based race detector to recommend each data race with minimal thread context switches. Specifically, we reduce the false positives by analyzing and tracking the dataflow in the program. By this means, DFTracker thus reduces the unnecessary analysis of false race schedules. We further propose a novel algorithm to recommend an effective race schedule with minimal thread context switches for each data race. Our experimental results on the real applications demonstrate that 1) without removing any true data race, DFTracker effectively prunes false positives by 68% in comparison with the state-of-the-art constraint-based race detector; 2) DFTracker recommends as low as 2.6–8.3 (4.7 on average) thread context switches per data race in the real world, which is 81.6% fewer context switches per data race than the state-of-the-art constraint based race detector. Therefore, DFTracker can be used as an effective tool to understand the data race for programmers.

Keywords data race, satisfiability modulo theory, multi-threaded program, dynamic detection

1 Introduction

A data race happens when multiple threads access the same memory location without appropriate synchronization, and at least one of them updates the value[1]. For programmers, data races reported by a race detector are generally used for verification and further understanding so that data races can be correctly fixed. For the sake of programmer productivity, it is particularly important for race detection in terms

Regular Paper

This work is supported by the National Key Research and Development Program of China under Grant No. 2023YFB4503400, and the National Natural Science Foundation of China under Grant Nos. 62322205, 62072195, and 61825202.

*Corresponding Author

of precision[2, 3] and the number of thread context switches exposed on the data race[4, 5]. Precision means that the race detector reports true data races without false positives so that programmers can concentrate on the real ones. The number of thread context switches represents the complexity of scheduling a data race. The larger the number of thread context switches is, the more difficult it is for programmers to understand this data race. If both aspects are well done by the race detector, much debugging effort of data races will be saved for programmers. An ideal race detection tool should report true data races and recommend the minimal context switches for each data race[6]. In this paper, we investigate whether and how we can improve the precision and reduce the number of context switches for each recommended data race.

Constraint-based analysis has been an effective means for race detection[7–9] (hereinafter referred to as CRD). [7–9] identify data races by formulating race detection as a constraint-solving problem. Specifically, on the basis of specific program execution, they construct a series of constraints to represent the program semantics and further generate all feasible program schedules by solving these constructed constraints via a satisfiability modulo theories (SMT) solver[10]. By this means, CRD is able to find all the races that can be found by any other sound race detector, and the generated schedules for the identified data races (abbreviated as race schedules) can be also used to reproduce how data races are scheduled.

Nevertheless, in practice, constraint-based race detectors[7–9] still have two major problems, limiting their application and productivity for programmers. CRD perceives only the semantics of basic program components (e.g., lock, read/write, and fork/join) and does not detect the complex and implicit control flow arising from the dataflow propagation of programs. CRD often leads to the incompleteness of constraint generation and mishandles the possible thread interleavings of lock-free structures (as discussed in Subsection 2.2).

As a result, [7–9] may generate excessive unnecessary race schedules for a large number of false positives (i.e., false races). Note that this weakness can be common in practice. As shown in the previous study[11], more than 88.7 % data races are related to the dataflow of the program (including ad hoc synchronization), especially with the pointer alias and reference variable. For a given data race, there may involve a crowd of race schedules to expose it. Nevertheless, CRD generates the race schedules for each data race whenever this data race is exposed during constraint solving. As a result, this ad hoc recommendation can impose too many context switches, which complicates the data race analysis for programmers. In summary, the state-of-the-art constraint-based race detection[7] is not friendly to programmers, which fails to detect true races only, and offers a minimal number of thread context switches for the debugging of a data race.

To address the two limitations in the state-of-the-art constraint-based race detection, this paper proposes DFTracker, an improved constraint-based race detector, which attempts to schedule the data races with a minimal number of thread context switches while removing unnecessary false positives. In order to eliminate the false dataflow-related races, we enhance CRD with dataflow tracking. DFTracker only focuses on tracking the dataflow related to the data races so as to reduce the analysis overhead. To be specific, the core process of dataflow detection is as follows. First, DFTtracker takes the data race candidates reported by the CRD approach as a dataflow analysis base. Second, DFTracker checks whether there exists a dataflow propagation path between two conflicting accesses of this data race candidate. Therefore, we maintain a happens-before (HB) order between them, and this data race candidate is classified as a false positive. Otherwise, this race candidate reported by CRD can be regarded as a true race.

In order to recommend the race schedule with the minimal number of thread context switches (abbreviated as minimal schedule), we propose a novel offline detection algorithm, which takes all generated feasible race schedules for each reported data race as input. Among these race schedules per data race, it analyzes all numbers of context switches for each race and then selects the minimal schedule for this race.

We evaluate the efficiency and effectiveness of DFTracker on five real programs including two server applications— Apache① and MySQL②, and three desktop applications—pbzip2③, TransmissionBT④, and Handbrake⑤. The experimental results demonstrate

①http://www.apache.org/, Jan. 2024.
②http://www.mysql.com/, Jan. 2024.
③http://compression.ca/pbzip2/, Jan. 2024.
④http://www.transmissionbt.com/, Jan. 2024.
⑤http://handbrake.fr/, Jan. 2024.

that 1) without removing any true data race, DFTracker effectively prunes false positives by 68% in comparison with the state-of-the-art CRD[7]; 2) DFTracker recommends as low as 2.6–8.3 (4.7 on average) thread context switches per data race for those real applications, which is 81.6% fewer context switches per data race than the state-of-the-art CRD. Therefore, DFTracker can be used as an effective tool to better understand and analyze data races for programmers.

The rest of this paper is organized as follows. Section 2 outlines the background and motivation of our work. We present the dataflow propagation detection in Section 3, and Section 4 elaborates how to recommend effective race schedules with the minimal number of thread context switches. Section 5 shows the experimental results. We survey related work in Section 6 and conclude this work in Section 7.

2 Background and Motivation

In this section, we first give a brief introduction to CRD[7]. Next, we give motivating examples to show two issues of CRD, including reporting false positives and recommending ineffective race schedules for race debugging. Motivated by the discussed examples, we finally present the overview of our proposal DFTracker to solve these problems for more effective and programmer-friendly race detection. To facilitate the descriptions, we define several notations listed in Table 1.

2.1 Constraint-Based Race Detection (CRD)

For a given program schedule, most existing dynamic analysis techniques (e.g., [3]) take the strict happens-before order[2] for the program analysis. Using different program schedules, they generally produce a different set of data races, that is, each program schedule is taken to report a set of data races (i.e., the race set in Fig.1), where the red star represents false data races.

Table 1. Notations in the Constraint Model

Notation	Description
e	Event in an execution trace
$\cdot$	Operator to obtain the attribute of an event, e.g., $e.type$
π	Set of events in a program schedule
$\mathsf{Race}(i, j)$	Data race between line i and line j in different threads
X_π	X constraints operated on the events in π
R_v^i	Value of read access to the shared variable v at line i
W_v^i	Value of write access to the shared variable v at line i
O_v^i	Partial order of v at line i in a program schedule

Thus a large number of unobserved data races in other schedules may be missed, as shown in Fig.1(a). If we want to get all race sets, we have to test all program schedules. In contrast, CRD breaks this limitation by using constraint-based analysis. It achieves the objective of getting all race sets without actually requiring all program schedules. Instead, it just takes only one representative program schedule as input.

Specifically, it records a specific program execution into a trace or a minimal set of traces[7]. Through analyzing the concrete program semantics (such as thread paths, reads/writes, and synchronization order) in the trace, CRD then abstracts a series of constraints to represent program schedules. They encode the constraint order of these events of a program schedule as follows:

$$\rho_\pi \wedge \iota_\pi \wedge \gamma_\pi,$$

where ρ_π denotes the partial order constraints which represent the must-happen-before relation between events (i.e., two events have the hard program order, such as fork/join, signal/wait, and program control flows). ι_π denotes the locking constraints, and γ_π de-

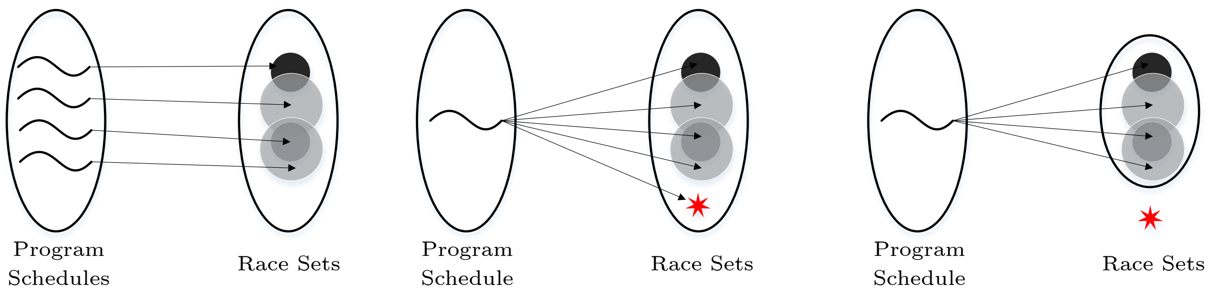


Fig.1. Different detection approaches. (a) HB-based detector. (b) CRD. (c) A precise CRD.

notes the read-write constraints. Finally, CRD invokes an SMT solver[10] to solve these constraints and generates all feasible program schedules. To further identify data races, CRD also encodes race constraints such that the order of two conflicting events a and b should be consecutive as $O_a^i = O_b^j + 1$, meaning that event b immediately happens after event a. Combining program constraints and race constraints, CRD then can generate all feasible race schedules. By this means, taking only one program schedule, CRD can identify all race sets in all feasible program schedules, as shown in Fig.1(b).

Unfortunately, due to the incomplete constraint generation, only the semantics of basic program components (e.g., lock, read/write, and fork/join) is guaranteed in their work. It lacks the corresponding strategies to track the implicit control flows arising from the dataflow propagation of the program. This weakness is also acknowledged by the authors in their paper[7], which does not provide a solution. Thus, the CRD approach may generate false positives and further induce to conduct some unnecessary constraint analysis. For a given data race, there may involve many race schedules to expose it.

Nevertheless, CRD generates the race schedules for each data race whenever this data race is exposed during constraint solving. As a result, this ad hoc recommendation can impose too many context switches, which complicates the data race analysis for programmers. In summary, the state-of-the-art constraint-based race detection[7] is not friendly to programmers, which fails to detect true data races only, and offers a minimal number of thread context switches for the debugging of a data race. Next, we will introduce motivating examples to illustrate these two issues of the CRD approach.

2.2 Motivating Examples

Dataflow propagation is very common in multi-threaded programs. It generally means that a thread requires shared data that is produced by another thread. As discussed previously, program components (e.g., lock, read/write, and fork/join) can be tracked precisely by the CRD approach. However, numerous program semantics of the lock-free structures, which can be scheduled with many possible thread interleavings for a multi-threaded program, may lead to the imprecision of the CRD approaches. The implicit order of these lock-free structures is generally propagated in the form of dataflow information, which is missed in the CRD approach. One common use case of dataflow propagation is producer-consumer communication, e.g., push/pop, enqueue/dequeue, write/read, and insert/delete. For the push/pop pair, only when the data is pushed into the stack, and later this data can be used with the pop operation.

Fig.2 depicts a simplified code from pbzip2. It employs the producer-consumer mechanism to compress the file in parallel: the main process produces the blocks into the queue q by reading the file and the consumer process compresses these blocks from the queue q in parallel. In this example, though the mutual semantics of the lock structure can be preserved by the existing CRD approach, the scheduling order of two critical sections is flexible and can be alternatively executed in different orders, depending on the runtime scheduling. In this case, the lock-free structures, i.e., the code snippets in S_1 and S_2, may be executed simultaneously. The CRD approach reports the shared data on *block* between S_1 and S_2 as a data race Race(S_1, S_2), since it argues that two blocks may be operated on the same shared memory location. In fact, when two blocks point to the same block, the order of $S_1 \rightarrow S_2$ is guaranteed through the manipulation of enqueue and dequeue. The parallelism of S_1 and S_2 is only valid when they are operated on different blocks.

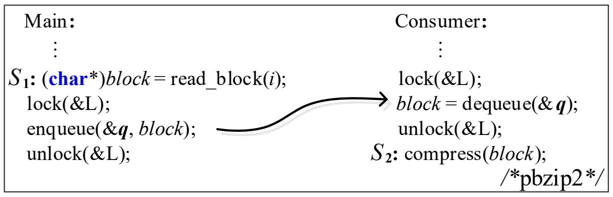

```
Main:                                      Consumer:
  ⋮                                          ⋮
S1: (char*)block = read_block(i);          lock(&L);
  lock(&L);                                block = dequeue(&q);
  enqueue(&q, block);                      unlock(&L);
  unlock(&L);                            S2: compress(block);
                                                        /*pbzip2*/
```

Fig.2. The shaded memory accesses are reported as a data race by CRD. In fact, it is a false positive because the order of $S_1 \rightarrow S_2$ is ensured by queue maintenance when both blocks in two threads are operated on the same shared memory address.

Race(S_1, S_2) is a false positive reported by CRD, because CRD is unaware of the dataflow propagation of the program, thus missing the potential happens-before order arising from the unrecognized dataflows. Still, it is a nontrivial task for CRD to recognize the dataflow propagation paths in real applications. First, two conflicting memory addresses of a data race may not be exactly the shared data used in the programs. In general, as shown in Fig.2, they may be the local memory that uses the same shared data computed or transferred from the values of other multiple objects. Second, the dataflow propagation of a conflicting memory address may be triggered in various forms, e.g., write-to-read pair, library call, and address passing. Both complex programming designs/implementations complicate the dataflow analysis of the program.

The memory order model is another main reason for the hard reproducibility of data races. A more subtle and critical point is that many relaxed memory models allow the compiler or CPU to reorder the shared memory accesses to different memory addresses. In Fig.3(b), only if line 4 and line 5 are reordered, $\mathsf{Race}(4, 10)$ can be exposed by CRD.

Apart from the above-discussed problem, by solving the program constraint with the race constraint via the SMT solver, we then get an enormous number of program schedules for each true data race. However, it is still difficult for programmers to effectively understand all these identified data races with these massively-generated schedules. As observed in real programs, the CRD approach recommends a schedule with an excessively randomized number of context switches, which is rather difficult and tedious for programmers to understand these data races. On the other hand, we can observe the opportunities to find the minimal number of context switches for each recommended schedule. Fig.3 shows a code snippet with two threads accessing two shared variables (x and y). If $T_1.4$ and $T_1.5$ can be reordered, there are two latent data races (i.e., $\mathsf{Race}(T_1.2, T_2.3)$ and $\mathsf{Race}(T_1.4, T_2.5)$) in this code. For $\mathsf{Race}(T_1.4, T_2.5)$, CRD may recommend many possible race schedules. For instance, the race schedule $T_1.1 \rightarrow T_2.1 \rightarrow T_1.2 \rightarrow T_2.2 \rightarrow T_1.3 \rightarrow T_2.3 \rightarrow T_1.5 \rightarrow T_2.4 \rightarrow T_1.4 \rightarrow T_2.5$ is relatively hard to understand because 10 thread switches are invoked for these 10 lines of code (i.e., LOC). On the other hand, the race schedule $T_2.1 \rightarrow T_2.2 \rightarrow T_2.3 \rightarrow T_1.1 \rightarrow T_1.2 \rightarrow T_1.3 \rightarrow T_1.5 \rightarrow T_2.4 \rightarrow T_2.5 \rightarrow T_1.4$ involves only three thread switches. This race schedule is more intuitive. The CRD approach may recommend those race schedules in an ad hoc manner.

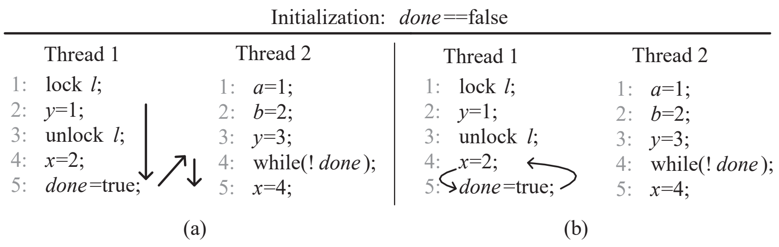
A few studies[12, 13] attempt to reduce the impact arising from context switching by producing simplified interleavings to find a root cause schedule for concurrency failures. In this earlier research, failures are assumed to be assertion violations. However, data races have far more complex concurrency behaviors[14]. They cannot always be formalized as an invariant assertion problem[13]. RaceDebugger complements to use dynamic slicing to tailor the original schedules generated by constraint-solving systems[8]. However, each exposed race schedule in RaceDebugger needs to be re-executed at runtime with some manual checkings to locate root causes precisely, limiting its practicability. DFTracker is different from but orthogonal to these earlier root cause researches[7–9]. DFTracker enables finding a minimal context-switching schedule which can be further used to explore the easy-to-understand root causes of concurrency bugs.

In addition, DFTracker works purely upon constraint-solving systems in a self-contained and automatic fashion without any manual interruption, improving the productivity of data race analysis.

2.3 Overview of DFTracker

To address the limitation of the CRD approach, we design DFTracker, an improved constraint-based race detector. Fig.4 shows the overview of DFTracker.

The basic idea of DFTracker is that it 1) uses dataflow tracking to eliminate the false positives reported by the CRD approach, and 2) recommends the


Initialization: *done*==false

(a)

```
Thread 1              Thread 2
1: lock l;            1: a=1;
2: y=1;               2: b=2;
3: unlock l;          3: y=3;
4: x=2;               4: while(!done);
5: done=true;         5: x=4;
```

(b)

```
Thread 1              Thread 2
1: lock l;            1: a=1;
2: y=1;               2: b=2;
3: unlock l;          3: y=3;
4: x=2;               4: while(!done);
5: done=true;         5: x=4;
```

Fig.3. Code snippet on different memory models. (a) $\mathsf{Race}(T_1.4, T_2.5)$ is undetectable on sequential consistent memory. (b) $\mathsf{Race}(T_1.4, T_2.5)$ is detectable if $T_1.4$ and $T_1.5$ are reordered.

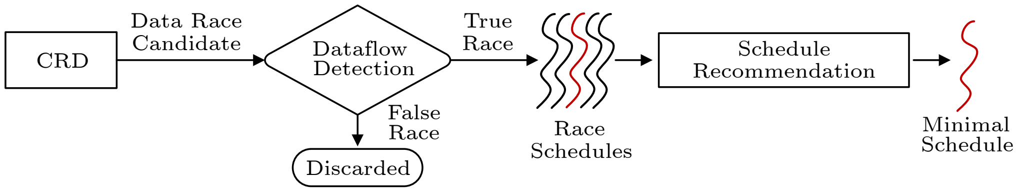


Fig.4. Overview of DFTracker.

race schedule with a minimal number of thread context switches. By this means, DFTracker improves the producibility of race detection in understanding and debugging data races.

DFTracker takes each data race candidate reported by the CRD approach as input, and augments the CRD approach in two major steps: dataflow detection and schedule recommendation. First, it invokes a dataflow checker to detect whether this data race candidate is caused by the unrecognized dataflow of the program (Section 3). If it is, DFTracker classifies this race candidate as a false race and discards it. Otherwise, we consider it as a true data race and DFTracker next generates all feasible race schedules that can expose this race. Among these generated race schedules, DFTracker invokes the schedule recommender to find out the final schedule with the minimal number of thread context switches (Section 4). Next, we will introduce each of them in detail.

3 Dataflow Propagation Detection

Since the existing CRD approach[7] reports false positives arising from the dataflow propagation of the program (discussed in Subsection 2.2), we propose to augment it with dataflow propagation tracking in order to suppress false positives of reported results. In this section, we present the design and implementation details for our dataflow tracking enhancement.

3.1 Overview

Fig.5 depicts the detailed dataflow detection process. In this work, the dataflow propagation of the program mainly refers to the observation that one access of data race candidate requires shared data that is updated by the other access.

DFTracker first performs the symbol parsing to track the dataflow dependency for two conflicting accesses of the data race candidate (i.e., S_1 and S_2). For the sake of symbol parsing, we develop an efficient approach to express and parse the dataflow of the program (①). Next, considering a data trace candidate suggested by CRD (denoted as $\mathsf{Race}(S_1, S_2)$), DFTracker then identifies the dependency point of S_2 (②) and the impact point of S_1 (③). The dependency point means the program point that S_2 depends on, whereas the impact point means the program point that S_1 impacts. The analysis of the dependency point and impact point is to ascertain the real dataflow path of the data race candidate. Recall that, in Subsection 2.2, we have shown the case studies where the dataflow propagation results in false positives in CRD. Therefore, after getting the dependency point and the impact point, DFTracker finally checks (④) whether there exists a dataflow propagation between the dependency point and the impact point. If so, it means that an HB order between them occurs, and this identified data race should be classified as a false positive. Otherwise, a true data race is reported. However, taking an identified data race candidate with two conflicting accesses, we still need to confirm which access happens first (abbreviated as the first order access, i.e., S_1), because this is critical to distinguishing the impact point from the dependency point. More details about how we solve it can be found in Subsection 3.6.

3.2 Dataflow Parsing

A multi-threaded program can run in a huge num-

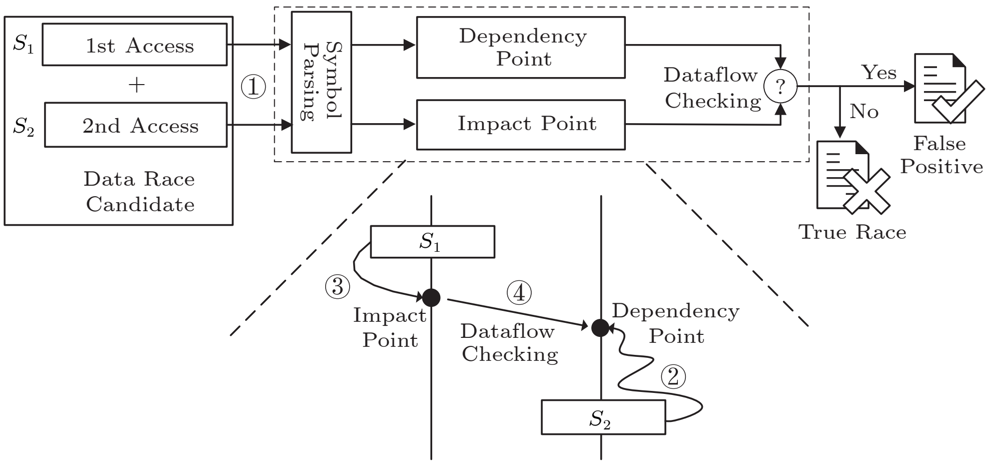


Fig.5. Dataflow propagation detection scheduling.

ber of ways because of well-known inter-thread interleavings. In addition, different program paths can further worsen the situation of the interleaving space significantly (a.k.a. the path explosion problem), leading to exponential space complexity. Therefore, it is generally difficult to capture a complete set of dataflows for a large multi-threaded program[15, 16]. However, analyzing the dataflow relation on this large exploration space also slows down the analysis efficiency significantly. Fortunately, race detection techniques focus on exposing data races from a specific execution trace. We, therefore, require only to explore the thread scheduling (e.g., lock interleavings[17]). Path explosion problem can be mitigated and also the analysis efficiency can be improved significantly. In particular, when parsing the input trace, the following information is recorded: 1) the number of threads launched, 2) executed instructions in each thread, and 3) the branch conditions triggering this execution. Based on the recorded information, the program execution can be transformed to be a sequence of statements.

In order to express the dataflow of the program, we adopt the symbolic expression and assignments for the program statements[6]. We can further reduce the overhead of program statements for dataflow analysis. Any reads or writes of a variable could be either a concrete value or a symbolic value computed from other symbolic value variables. To represent program statements, we define two basic statements as follows:

- expression: $x \mid x \ \mathsf{op}\ y \mid \mathsf{op}\ x$,
- assignment: $x = y \mid x = y \ \mathsf{op}\ z \mid x = \ \mathsf{op}\ y$,

where x, y, and z are the variables in the format of R_v^i and W_v^i in Table 1. op is the operator between two variables. For instance, we represent a branch expression $x > y$ at line 1 of a program as the symbolic equation $R_x^1 > R_y^1$. The assignment statement $x = 1$ at line 2 is denoted as the symbolic equation $W_x^2 = 1$. For other compound statements, such as return, goto, and call, we directly execute them with the symbolic variable until all statements can be expressed with basic statements. For instance, for the statement $\mathsf{a} = \mathsf{foo}()$, we first call foo, then record the basic statements in foo, and finally assign its returned value to a.

After getting the symbolic equations of program statements, we parse the dataflow propagation of the program by analyzing these symbolic equations. With parsing, we can obtain the dataflow propagation path for variables of an arbitrary code line pair. For instance, for the statements "$a = x;\ y = a + 1;$", we can infer that the shared memory address y depends on the variable a which further relies on the shared access x. Thus, we can get the dataflow propagation path of these two statements as $x \rightarrow y$.

3.3 Dependency Point Analysis

To preserve the precision of dataflow analysis, all possible dependency points of S_2 access must be detected. However, it is notoriously difficult, if not possible, to find a complete set of dependency points. Not all statements declare the variables they will use clearly. For example, the third-party library is often self-enclosed and little information is available from its public manual. In addition, the dependency relationship between two variables is not necessarily direct. They may be potentially dependent through one (or more) dependency relationship(s) from other variable(s).

Given a data trace candidate suggested by CRD ($\mathsf{Race}(S_1, S_2)$), DFTracker first parses the symbolic equation of S_2 and finds all variables it depends on (i.e., dependency variables). Then it searches the program statements that happen before the S_2 access in the same thread and locates the program statements assigning the value to those dependency variables. During the search process, we consider the following four cases by addressing the challenges raised above.

Constant. It means that the statement does not have a dependency point. For the statement $x = 1$, the shared memory x is no longer dependent on any other variable.

Local Memory. In this case, we still do not find the final dependency point, because this local variable may point to (i.e., depend on) the other shared variable that is then propagated to another shared variable, e.g., the address exchange between two shared accesses using the temporary pointer reference. As a consequence, DFTracker next finds the latest change of this local variable by further searching the statements that happens before in the same thread and repeats the process.

Shared Memory. In this case, a dependency point candidate is found, but still, we do not find all potential ones because this dependency point may still depend on other shared variables. In order to find the remaining possible dependency points, DFTracker then sets this shared memory as a new source to be analyzed and finds its dataflow dependency by repeating the local search.

Function Calls. For an internal function call (e.g., a customized function), DFTracker identifies its dataflow by tracking the program statements of the function. For an external function call (e.g., a library call and system call), it is hard to obtain the concrete symbolic equation of the program statement in the external function. However, this is not a problem for the dataflow analysis since the dataflow dependency can be still tracked by observing the input/output of the function at runtime. By and large, we can obtain the dataflow dependency of external functions with a general pattern of input→output.

Finally, DFTracker ends the search process when either of the following conditions is satisfied: 1) no more statements are available in the local thread; 2) a constant is identified (the first case in the above); 3) a certain function call is found, which does not take the shared memory address as input. If the head/tail of a thread is reached, it means that no more dependency points exist. Besides, in our observations, almost all dataflow cases end in the format of the direct constant assignment (or indirect constant expression). Otherwise, the potential dataflow exists, such as local memory and shared memory. For instance, if a function call does not take the shared memory address as input, any returned value of this function may not depend on the previous dataflow, e.g., malloc(sizes). In this case, an indirect constant is calculated.

3.4 Impact Point Analysis

After getting all possible dependency point candidates for the data trace candidate suggested by CRD ($\mathsf{Race}(S_1, S_2)$), DFTracker attempts to identify the corresponding impact point of the S_1 access. Each impact point produces the shared data required by a dependency point. The CRD suffers from the substantial unnecessary analysis arising from false positives, since it misses the implicit dataflow propagation relationship. That is, false positives of CRD arise from the dataflow propagation pattern: the dependency point reads the shared memory address whose address is updated by the impact point. In this context, we have an insight that the false positive reduction can be transformed into a problem of finding impact points.

Based on this insight, we identify the impact point of S_1 as follows. First, DFTracker searches the program statements that happen after the S_1 access in the same thread. Second, DFTracker tries to find the latest write operation which uses the conflicting address of the data race candidate to update a shared variable. If such a write operation is found, DFTracker then treats the corresponding writing site as the result impact point. Otherwise, DFTracker keeps on searching until no more statements are left. If no such impact point is found, we consider S_1 as the result impact point.

Similar to the dependency point analysis, if the impact point analysis encounters the external function calls, DFTracker just validates whether the input and output of this call are the conflicting address of the reported race and other different shared variables. If so, DFTracker determines this program point as the result impact point. Otherwise, the local search continues.

3.5 Dataflow Checking

Given a pair of a dependency point and an impact point, DFTracker then checks whether there exists a dataflow between the dependency point and the impact point, i.e., whether the impact point writes shared data that is read by the dependency point. In this context, we know that the impact point must happen after the dependency point. That is, we must explicitly know the scheduling order of the impact and dependency points. Unlike the logic clock that requires computing the relative order of any two events, we simply use the physical time when an instruction is executed as the timestamp without the overhead of maintaining the relative orders. Since the shared variable can induce a dataflow propagation, we propose to reduce the timestamp cost by recording only the timestamp for shared variables in a minimized timestamp recording overhead.

DFTracker first compares the timestamp of the impact point with that of each dependency point candidate. We record the timestamp and dynamic address of the shared memory accesses when the program trace is being recorded (more details in Subsection 3.6). According to the timestamps of the impact point and the dependency point at runtime, we then discard those dependency point candidates that have a smaller timestamp than the impact point because a valid dependency point oughts to happen after the impact point if they indeed involve a dataflow dependency.

Next, DFTracker determines a dataflow propagation path between the impact point and the dependency point. Particularly, we consider the following three conditions: 1) the impact point and each dependency point share the same memory address variable $addr$, 2) the impact point writes $addr$, and 3) the dependency point reads $addr$. If all the three conditions above are satisfied, a dataflow propagation path between the impact point and the dependency point exists. In this case, two conflicting accesses (S_1 and S_2) have an HB order, and DFTracker classifies it as a false positive. Otherwise, DFTracker considers CRD reports the true race. We note that the dataflow checking procedure above can be very fast since the checking on the memory address simply uses comparison operations.

3.6 Implementation Issues

Race Access Order Confirmation with Timestamp. Each data race candidate is paired with two accesses, i.e., S_1 and S_2 in Fig.5. To expose this race, CRD may construct the race constraint as $O_{S_1} = O_{S_2} + 1$ or $O_{S_2} = O_{S_1} + 1$. If $\mathsf{Race}(S_1, S_2)$ is a real race, both race constraints above show the race behavior. However, if $\mathsf{Race}(S_1, S_2)$ is a false positive on account of the dataflow dependency $S_1 \rightarrow S_2$, the case of $O_{S_1} = O_{S_2} + 1$ will invalidate our dataflow propagation detection in Section 3. That is because, in this case, DFTracker will perform the impact point analysis for S_2 (instead of S_1) and the dependency point analysis for S_1 (instead of S_2). This leads to the opposite (wrong) analysis and violates the real dataflow propagation path. Therefore, given a data race candidate, we need to confirm which access is the first order access, i.e., S_1, for the correct dataflow analysis. To tackle this problem, we record the timestamp of the memory access of data race candidates when the program trace is being recorded. For a given data race candidate $\mathsf{Race}(x, y)$, if x has a smaller timestamp than y at runtime, DFTracker uses the race constraint $O_y = O_x + 1$ to expose this race. That is, x is the first order access (i.e., S_1) for the impact point analysis, and y is the second order access (i.e., S_2) for the dependency point analysis.

Symbolic Address Profiling. A known issue for the symbolic execution is that the runtime information of memory addresses is lost, e.g., pointer references. KLEE[18] does not provide any address tracking for the pointer analysis. For the sake of the precise analysis of pointers, we track each data when the program is being performed, and maintain an ordered list of memory addresses for each symbolic object. If a data race is detected, DFTracker will check the dynamic memory addresses of two accesses in the profiling list. If they are the same, we accept this reported race as the data race candidate for further dataflow detection. Otherwise, we ignore this report. Through symbolic address profiling, we can significantly reduce the large number of data race candidates arising from the pointer alias as the input of DFTracker, thereby making the dataflow detection more effective.

4 Schedule Recommendation

In this section, we present a novel offline approach, which recommends the optimal race schedules with minimal thread context switches for programmers.

4.1 Minimal Thread Context Switches: Research Method

To better help programmers understand how the data race occurs, it is necessary to recommend the easy-to-follow minimal schedule for programmers. In fact, most real-world concurrency bugs generally can be exposed with a few thread context switches[5].

To find the minimal schedule, one intuitive method is to bound the number of context switches for the recommended race schedules during the constraint solving. Specifically, the context switch threshold starts from zero. For each round of constraint solving, if the SMT solver fails to return a race schedule at some threshold, the threshold will be added. Then the next round of solving process with the added threshold is performed. Once a solution is found, the current threshold of context switches is minimal. However, this online approach serializes the solving process due to the bottleneck of the continuously-increasing context switch threshold, thus providing inefficient constraint solving. Next, we introduce a novel offline approach to solve this problem more efficiently.

4.2 An Offline Algorithm

In our work, we perform constraint solving based on a specific execution trace. Hence, DFTracker does not suffer from the path explosion as discussed in Subsection 3.2. This is also the very reason for the online bounded method above to enable reducing the search space for program analysis. More importantly, DFTracker further narrows the search space of thread schedules with the main focus on a few data race candidates. Hence, instead of solving race schedules one by one, we can enumerate all race schedules in a finite number and perform an offline approach to free the capacity of the SMT solver without serialization searching.

To find the minimal schedule, we first collect all feasible race schedules for all identified data races. Then, we divide all these collected race schedules into different race schedule sets by each data race, that is, each data race has a set of race schedules that can expose itself. Ultimately, the problem is transformed to this question: in each race schedule set, how to find out the minimal schedule to expose this race?

We, therefore, propose Algorithm 1, which mainly aims to answer the above-proposed question. To facilitate the descriptions, we define the partial order of the i-th event in a race schedule S as $S(i)$. For instance, $S_1(1)$ in Fig.6 represents $O_b^5(T_2)$ which means the program order of the data b at code line 5 in thread T_2. Hence, the race schedule order of S_1 can be rewritten as $S_1(1) < S_1(2) < \ldots < S_1(7)$. To further obtain the attribute of an event, we can make it through the operator $\cdot$ defined in Table 1, e.g., $S_1(1).threadnum$ denotes the thread identifier of the 1st event in S_1. As illustrated in Algorithm 1, we take two major steps.

Algorithm 1. Minimal Thread Context Switches

```
 Input: the schedule set $SS = \{S_1, \ldots, S_m\}$
 Output: $RecS$, the recommended schedule $MSN$,
 minimal switch number
1:  $MSN \leftarrow \infty$;
2:  for $i \leftarrow 1$ to $m$ do
3:      if $SwitchNum(S_i) < MSN$ then
4:          $MSN \leftarrow SwitchNum(S_i)$;
5:          $RecS \leftarrow S_i$;
6:  return $\langle RecS, MSN\rangle$
    SwitchNum Function
    Input: $S = \{S(1) < S(2) < \ldots < S(n)\}$
7:  $SwitchNum \leftarrow 0$;
8:  $i \leftarrow 1$;
    /* Check the type of data race */
9:  while: $S(i) < S(i+1)$
      is not the data race candidate do
        /* Check thread identifier. If not
           equal, the context is switched */
10:     if: $S(i).threadnum \neq S(i+1).threadnum$ then
11:         $SwitchNum \leftarrow SiwtchNum + 1$;
12:     $i \leftarrow i + 1$;
13: return $SwitchNum$
```

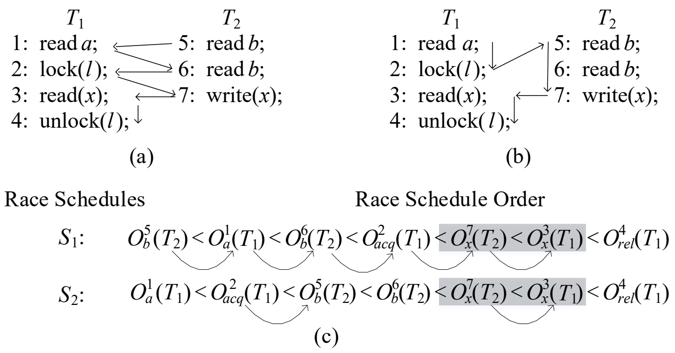

T_1
T_2
1: read a;
2: lock(l);
3: read(x);
4: unlock(l);
5: read b;
6: read b;
7: write(x);
(a)
(b)
Race Schedules
Race Schedule Order
S_1: $O_b^5(T_2) < O_a^1(T_1) < O_b^6(T_2) < O_{acq}^2(T_1) < O_x^7(T_2) < O_x^3(T_1) < O_{rel}^4(T_1)$
S_2: $O_a^1(T_1) < O_{acq}^2(T_1) < O_b^5(T_2) < O_b^6(T_2) < O_x^7(T_2) < O_x^3(T_1) < O_{rel}^4(T_1)$
(c)

Fig.6. Thread context switches with two cases, where the arrow across threads denotes the thread context switch. (a) A case with five thread switches. (b) Another case with two thread switches. (c) Race schedules where the data race is shaded in the race schedule order.

1) In a certain race schedule, it checks the thread identifier of each pair of two adjacent events as depicted by lines 9–11 in Algorithm 1. If they are performed by different threads, the thread context is switched. Otherwise, no context switch happens. By this means, we can calculate the context switch of each race schedule in a race schedule set for a given race.

2) After the results are collected in the first step, we then compare the number of context switches of all race schedules in a sequential order as depicted by lines 2–5 in Algorithm 1. In the end, we recommend the race schedule with minimal thread context switches.

As depicted in Fig.6, there are two race schedules that can expose Race(3, 7), such as $\mathcal{S}_1$ and $\mathcal{S}_2$. Algorithm 1 collects that $\mathcal{S}_1$ invokes five context switches while S_2 has two context switches. From these two results, race schedule $\mathcal{S}_2$ is recommended by Algorithm 1 as the minimal schedule.

In addition, it should be noted that two loop bodies in Algorithm 1 do not contain any data dependency. As a result, our offline approach also provides the opportunity to drastically accelerate the look-up of the minimal schedule using parallel solving. A more detailed discussion can be found in Section 6.

5 Evaluation

In this section, we evaluate the effectiveness and efficiency of DFTracker against the state-of-the-art constraint-based race detection.

5.1 Methodology

We evaluate DFTracker with five common real-world applications with different complexities (including two server applications—Apache and MySQL, and three desktop applications—pbzip2, TransmissionBT, and Handbrake). We use the underlying constraint-solving kernel of RaceDebugger[8] (with its dynamic slicing component disabled) as the representative of CRD. To support the symbolic computation for C/C++ programs, we also extend to use the underlying constraint system as in Symbiosis[12] on top of LLVM[19] and KLEE[18]. We use Yices SMT solver[⑥] as our constraint solver.

To facilitate usability, we have integrated CRD in the DFTracker framework. We can enable the functionality of CRD by using DFTracker with the option of −without-dataflow. To compare CRD with DFTracker, we first record one execution trace of each application and then perform two tools to collect the results using the same trace. The previous study[11] has shown that almost all data races are guaranteed to manifest themselves with two threads. Therefore, in our tests, all applications run with two threads.

All experiments are on a machine with four Intel® Hexa-core Xeon® CPU E5-2620 v2@2.10 GHz processors, 126 GB memory, and 1 TB SATA hard disk. The running operating system is CentOS 6.5 (x86_64) with Linux kernel 2.6.32.

We next present the effectiveness of how DFTracker can prune false positives reported by CRD. Then, we study the number of context switches for the recommended schedule by DFTracker in comparison with CRD. Finally, we report the runtime overhead of DFTracker to show that DFTracker is an efficient tool to understand races.

5.2 Precision of DFTracker

To evaluate the capability of false positive pruning of DFTracker, three scenarios are considered. We analyze: 1) the program trace with a given input 10 times using HB relation; 2) one selected execution trace once using CRD; and 3) one selected execution trace once using DFTracker.

False Race Pruning. Table 2 shows the number of reported data races with three techniques in the real world. For completeness, we also study the HB-based approach[⑦]. The results show that CRD detects more data races than the HB-based technique for all benchmarks. This is consistent with the previous study[7]. DFTracker prunes as many as 68% of false dataflow-related data races reported by CRD (↓68%). For instance, CRD detects 126 races for Apache, and DFTracker detects 86 of them as false positives. This demonstrates that while CRD offers a more sound solution to enhance the detection capability of the HB-based technique, it introduces a quite significant number of false data races due to the unawareness of program dataflows.

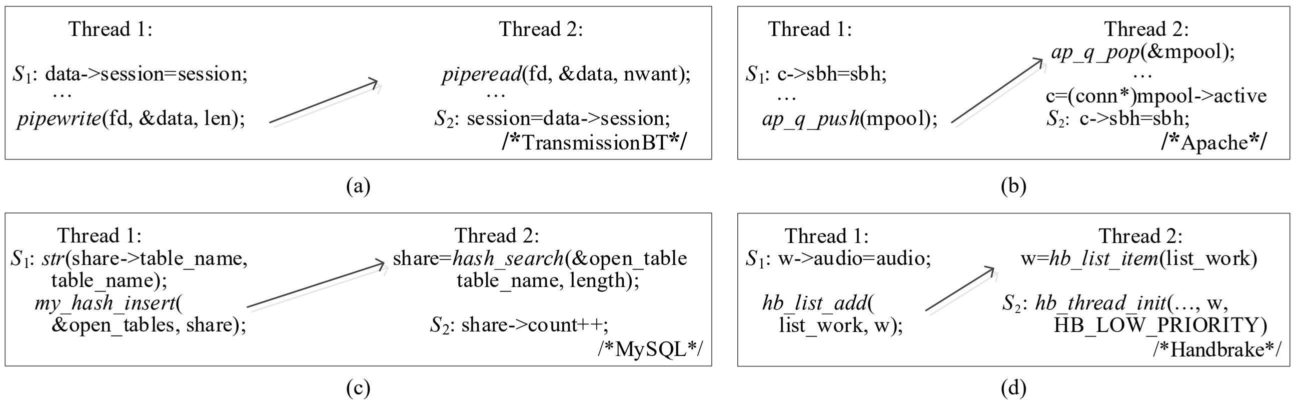
To further verify that DFTracker effectively prunes false dataflow-related alarms reported by CRD without removing any true data races, we have manually checked each reported race in Table 2 by CRD. The HB-based detector does not report the false data race related to the dataflow, as HB relation enforces the hard ordering of the program dataflow. In our observations, we find that 1) all races identified by HB are also included in the race set of CRD/DFTracker; 2) all false positive races identified by DFTracker are truly false races (caused by dataflow propagations). We use case studies to demonstrate those findings from our manual investigation.

Case Studies. We have presented a case study for pbzip2 in Fig.2. Here, we list several real examples identified by DFTracker in the other four real applications in Fig.7. All cases show a dataflow propagation of the program between S_1 and S_2, and CRD reports the false data races between S_1 and S_2. For instance, for TransmissionBT, the address of data in thread 1 is propagated from thread 1 to thread 2 through the shared pipe identifier `fd`, and then used

Table 2. Effectiveness of DFTracker in Pruning the False Data Races Arising from the Unrecognized Dataflow for CRD

App	LOC (k)	Code Size (M)	Number of Data Races			
			HB	CRD	DFTracker	Reduction
Apache	392	6	31	126	40	86
MySQL	1 132	22	44	211	63	148
pbzip2	5	1	3	28	9	19
TransmissionBT	79	4	8	45	19	26
Handbrake	1 070	3	6	34	11	23
Total	-	-	92	444	142	302 (↓68%)

[⑥]http://yices.csl.sri.com/, Jan. 2024.
[⑦]https://software.intel.com/en-us/intel-inspector-xe, Jan. 2024.



```
Thread 1:                                   Thread 2:
S1: data->session=session;                  piperead(fd, &data, nwant);
…                                           …
pipewrite(fd, &data, len);                  S2: session=data->session;
                                                    /*TransmissionBT*/
```
(a)

```
Thread 1:                     Thread 2:
                              ap_q_pop(&mpool);
S1: c->sbh=sbh;               …
…                             c=(conn*)mpool->active
ap_q_push(mpool);             S2: c->sbh=sbh;
                                      /*Apache*/
```
(b)

```
Thread 1:                               Thread 2:
S1: str(share->table_name,              share=hash_search(&open_table
    table_name);                              table_name, length);
  my_hash_insert(
    &open_tables, share);               S2: share->count++;
                                                  /*MySQL*/
```
(c)

```
Thread 1:                     Thread 2:
S1: w->audio=audio;           w=hb_list_item(list_work)

hb_list_add(                  S2: hb_thread_init(…, w,
  list_work, w);                    HB_LOW_PRIORITY)
                                            /*Handbrake*/
```
(d)

Fig.7. False data races caused by the unrecognized dataflows in the four real-world programs. The two statements indicated by the arrow are the impact point and the dependency point, respectively.

to invoke a task in thread 2. Since there is a happens-before order between *pipewrite* and *piperead*, the two shared accesses between S_1 and S_2 will be ordered as well and never conflict with each other. However, CRD fails to recognize this implicit control flow, without tracking the dataflow. Likewise, the cases from Apache, MySQL, and Handbrake also have similar dataflow propagations, and CRD identifies them as data races (in fact, these reports are wrong).

5.3 Effective Schedule Recommendations

To evaluate the effectiveness of DFTracker for the understanding and debugging of data races, we mainly measure two metrics: 1) schedule reduction with/without the dataflow detection strategy discussed in Section 3, and 2) minimal thread-context-switching schedule recommendation with/without our Algorithm 1 proposed in Section 4. The detailed results are listed in Table 3.

Schedule Reduction with Dataflow Detection. In order to illustrate how the data race happens, a large number of race schedules have been generated via constraint solving. In Table 3, we can find that one data race generally involves a large number of race schedules. For instance, for Apache, CRD reports 126 data races, and 89 210 race schedules have been generated. In contrast, DFTracker reports 40 true data races, and only 7 021 race schedules are generated, thus reducing 92.1% of race schedules in comparison with CRD. On average, through pruning the false positives with the dataflow detection, the number of race schedules is reduced significantly by 92.4% on average (↓92.4%). As a side product, this reduction also increases the efficiency of race schedule analysis for the optimal schedule recommendation.

Schedule Recommendation with Minimal Thread Context Switches. Programs can have very different data races, which require different numbers of context switches to expose themselves. As shown in Table 3, DFTracker recommends 2.6–8.3 (4.7 on average) thread switches for each race while CRD reports 11.7–53.9 (25.5 on average) thread switches. That is because CRD recommends the current thread schedule in an ad hoc manner.

Overall, DFTracker reduces the number of context switches per recommended trace by 81.6 % on average (↓81.6%). For instance, Fig.8 illustrates a real race example from MySQL. We have tested this example 10 times using CRD and DFTracker, respectively. The results show that CRD produces the uncertain recommendation of thread switches ranging from 14 to 26; while DFTracker always recommends 10 context switches for this race. We highlight that

Table 3. Effectiveness Evaluation of Race Schedules in the Real World

App	Schedule Reduction		Schedule Recommendation	
	CRD	DFTracker	CRD	DFTracker
Apache	89 210	7 021	34.2	6.5
MySQL	195 916	9 198	53.9	8.3
pbzip2	4 201	872	11.7	2.6
TransmissionBT	17 194	5 891	13.1	3.4
Handbrake	11 352	1 087	14.6	4.0
Average	63 575	4 814 (↓92.4%)	25.5	4.7 (↓81.6%)

```
Thread 1:
buf_pool_check_no_pending_io(void){
 bool ret=true;
 if(buf_pool->n_pend_reads){
         ret=false;
 }
 return ret;
}
Thread 2:
buf_page_ip_completed(buf_page_t* bpage){
 //processing read requests
 buf_pool->n_pend_read--;
}                                    /*MySQL*/
```

Fig.8. True data race from MySQL.

DFTracker is deterministic while the CRD approach is nondeterministic. Therefore, DFTracker saves much debugging effort for programmers to understand the complex thread context switching.

5.4 Runtime Overhead

To evaluate the runtime overhead, we compare the elapsed time of constraint solving between CRD and DFTracker using the five real applications above.

Fig.9 illustrates the elapsed time of constraint solving for the race schedules between DFTracker and CRD. DFTracker has almost the same performance as CRD, introducing only 5.4% runtime overhead. That means DFTracker is able to amend the two weak points of CRD with almost negligible runtime overhead reintroduced. There are two main reasons for such high efficiency. First, we leverage the symbolic expression for the program statements, and then perform symbolic execution to simulate the program execution. This makes the dataflow analysis of the program much easier through only parsing the symbolic equations as discussed in Subsection 3.2. Second, the

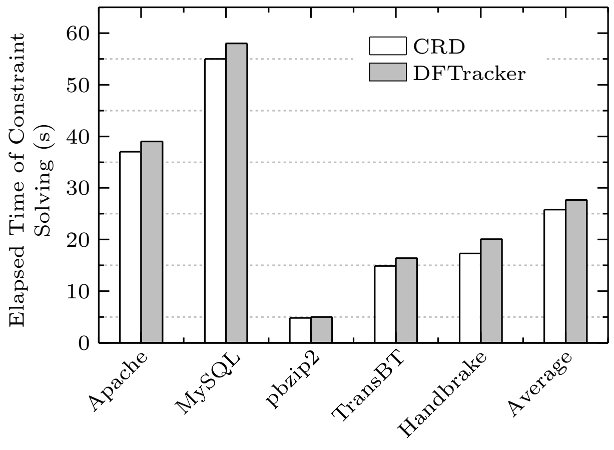


Fig.9. Elapsed time of constraint solving between CRD and DFTracker.

impact point of the first-order access and the dependency point of the second-order access tend to have a locality in real applications. For instance, S_1 and S_2 of all cases in Fig.7 have a few lines of the distance to the impact point and the dependency point. As a result, DFTracker can locate the impact point and the dependency point fast in practice.

6 Related Work

Dynamic Race Detection. Dynamic race detection[3, 14, 20, 21] is an important and fruitful research area in the literature. It is mainly used to precisely track some inclusive data races observed from a specific program execution. Most existing dynamic detectors are rooted in the happens-before (HB) relation[2, 22], which holds the hard ordering for lock semantics, and thus limits the detection capability. As a consequence, the lockset algorithm[23] and causally-precede relation[24] were proposed to improve the detection coverage. However, these approaches are still either unsound or missing some races. Constraint analysis techniques (CRDs) are used to significantly improve the detection coverage[7, 25]. The CRD approaches are sound and can generate all feasible program executions for race detection. However, CRD still suffers from false positives and the excessive number of context switches for the recommended data race. DFTracker augments the existing CRD tool with the awareness of dataflow and resolves the two issues of CRD.

Dataflow Analysis. Dataflow analysis is a technique for gathering information about the possible set of values calculated at various program points[26–29]. Procrustes[26] uses dataflow graphs to accelerate the sparse deep neural network training by characterizing access patterns. Wongsuphasawat *et al.* used the dataflow to improve the graph visualization in TensorFlow[27]. However, these techniques cannot be applied to our problem, since DFTracker only takes one execution trace performed by a specific input. As a consequence, DFTracker does not need to track the whole branch/path (i.e., control flow) information of the program. Also, the dataflow propagation of the program is identified by backtracking the related-event order in a recorded trace[30]. To pinpoint the data dependencies of program executions, they have to record a volume of runtime information into the trace, e.g., instructions, register values, memory addresses, and their changes every time. This makes the dataflow detection with prohibitively high cost. Besides, this technique also strictly depends on the event order in a specific execution trace, which is not helpful for race exploration in other execution traces. In contrast, DFTracker simplifies the dataflow analysis through symbolic parsing (without CFG (Control Flow Graph)). Furthermore, DFTracker generalizes the dataflow problem with the constraint analysis technique, which removes the restricted event order recorded in the trace.

Context Switch Bounded Analysis. The context switch bounded analysis is originally used to transform an NP-hard problem to a solvable polynomial problem for the complete analysis of concurrent programs in theory[4–6, 31]. Later, it is used for programmers to understand how the bugs happen. For instance, Inverso *et al.*[32] found a large number of bugs with a fixed bound of context switches in a parallel and distributed manner. Other work[4–6] extends the analysis of context switch bounded problems by giving priority to schedules with fewer preemptions. As we have discussed in Section 4, context switch bounded analysis may lead to inefficient constraint solving. Even worse, for those races with a relatively large number of context switches, the existing context switch bounded analysis has to discard them for efficiency. In this study, we formulate the problem as a special case of a few identified data races. DFTracker collects a finite number of race schedules. As a result, we are able to enumerate all race schedules to find minimal context switches for each data race, with a low runtime overhead.

7 Conclusions

In this paper, we proposed DFTracker, which enables recommending each race with minimal thread context switches while ensuring no false positives. DFTracker uses the dataflow propagation of programs to determine false positives for the constraint-based approach. A novel algorithm was further developed to recommend effective race schedules for debugging. This facilitates programmers to understand a data race reported better. Experimental results and case studies on real applications showed DFTracker removes false positives with fewer context switches significantly over the state-of-the-art approaches. In the future, it would be interesting to apply DFTracker to debug data races that exist in more than two threads, which involves further dataflow analysis across multiple threads.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Netzer R H B, Miller B P. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1992, 1(1): 74–88. DOI: 10.1145/130616.130623.

[2] Lamport L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978, 21(7): 558–565. DOI: 10.1145/359545.359563.

[3] Flanagan C, Freund S N. FastTrack: Efficient and precise dynamic race detection. In *Proc. the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2009, pp.121–133. DOI: 10.1145/1542476.1542490.

[4] Tessler C, Fisher N. BUNDLEP: Prioritizing conflict free regions in multi-threaded programs to improve cache reuse. In *Proc. the 2018 IEEE Real-Time Systems Symposium*, Dec. 2018, pp.325–337. DOI: 10.1109/RTSS.2018.00048.

[5] Davis R I, Altmeyer S, Burns A. Mixed criticality systems with varying context switch costs. In *Proc. the 2018 IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr. 2018, pp.140–151. DOI: 10.1109/RTAS.2018.00024.

[6] Huang J, Zhang C, Dolby J. CLAP: Recording local executions to reproduce concurrency failures. In *Proc. the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2013, pp.141–152. DOI: 10.1145/2491956.2462167.

[7] Huang J, Meredith P O N, Rosu G. Maximal sound predictive race detection with control flow abstraction. In *Proc. the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2014, pp.337–348. DOI: 10.1145/2594291.2594315.

[8] Zheng L, Liao X F, Jin H, He B S, Xue J L, Liu H K. Towards concurrency race debugging: An integrated approach for constraint solving and dynamic slicing. In *Proc. the 27th International Conference on Parallel Architectures and Compilation Techniques*, Nov. 2018, Article No. 26. DOI: 10.1145/3243176.3243206.

[9] Pereira J C, Machado N, Pinto J S. Testing for race conditions in distributed systems via SMT solving. In *Proc. the 14th International Conference on Tests and Proofs*, Jun. 2020, pp.122–140. DOI: 10.1007/978-3-030-50995-8_7.

[10] De Moura L, Bjørner N. Z3: An efficient SMT solver. In *Proc. the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Mar. 2008, pp.337–340. DOI: 10.1007/978-3-540-78800-3_24.

[11] Lu S, Park S, Seo E, Zhou Y Y. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. the 13th International Confer-*

ence on Architectural Support for Programming Languages and Operating Systems, Mar. 2008, pp.329–339. DOI: 10.1145/1346281.1346323.

[12] Machado N, Lucia B, Rodrigues L. Concurrency debugging with differential schedule projections. In *Proc. the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2015, pp.586–595. DOI: 10.1145/2737924.2737973.

[13] Machado N, Lucia B, Rodrigues L. Production-guided concurrency debugging. In *Proc. the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2016, Article No. 29. DOI: 10.1145/2851141.2851149.

[14] Mathur U, Pavlogiannis A, Viswanathan M. The complexity of dynamic data race prediction. In *Proc. the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, Jul. 2020, pp.713–727. DOI: 10.1145/3373718.3394783.

[15] Zhang X Y, Gupta R. Whole execution traces. In *Proc. the 37th International Symposium on Microarchitecture*, Dec. 2004, pp.105–116. DOI: 10.1109/MICRO.2004.37.

[16] Qin F, Wang C, Li Z M, Kim H S, Zhou Y Y, Wu Y F. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2006, pp.135–148. DOI: 10.1109/MICRO.2006.29.

[17] Zheng L, Liao X F, He B S, Wu S, Jin H. On performance debugging of unnecessary lock contentions on multicore processors: A replay-based approach. In *Proc. the 2015 IEEE/ACM International Symposium on Code Generation and Optimization*, Feb. 2015, pp.56–67. DOI: 10.1109/CGO.2015.7054187.

[18] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. the 8th USENIX Conference on Operating Systems Design and Implementation*, Dec. 2008, pp.209–224.

[19] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. the International Symposium on Code Generation and Optimization*, Mar. 2004, pp.75–86. DOI: 10.1109/CGO.2004.1281665.

[20] Xu M, Kashyap S, Zhao H Q, Kim T. Krace: Data race fuzzing for kernel file systems. In *Proc. the 2020 IEEE Symposium on Security and Privacy*, May 2020, pp.1643–1660. DOI: 10.1109/SP40000.2020.00078.

[21] Endo A T, Møller A. NodeRacer: Event race detection for Node. js applications. In *Proc. the 13th IEEE International Conference on Software Testing, Validation and Verification*, Oct. 2020, pp.120–130. DOI: 10.1109/ICST46399.2020.00022.

[22] Mathur U, Kini D, Viswanathan M. What happens-after the first race? Enhancing the predictive power of happens-before based dynamic race detection. *Proceedings of the ACM on Programming Languages*, 2018, 2(OOPSLA): Article No. 145. DOI: 10.1145/3276515.

[23] Xie X W, Xue J L. Acculock: Accurate and efficient detection of data races. In *Proc. the International Symposium on Code Generation and Optimization*, Apr. 2011, pp.201–212. DOI: 10.1109/CGO.2011.5764688.

[24] Genç K, Roemer J, Xu Y F, Bond M D . Dependence-aware, unbounded sound predictive race detection. *Proceedings of the ACM on Programming Languages*, 2019, 3(OOPSLA): Article No. 179. DOI: 10.1145/3360605.

[25] Roemer J, Genç K, Bond M D. SmartTrack: Efficient predictive race detection. In *Proc. the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2020, pp.747–762. DOI: 10.1145/3385412.3385993.

[26] Yang D Q, Ghasemazar A, Ren X W, Golub M, Lemieux G, Lis M. Procrustes: A dataflow and accelerator for sparse deep neural network training. In *Proc. the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2020, pp.711–724. DOI: 10.1109/MICRO50266.2020.00064.

[27] Wongsuphasawat K, Smilkov D, Wexler J, Wilson J, Mané D, Fritz D, Krishnan D, Viégas F B, Wattenberg M. Visualizing dataflow graphs of deep learning models in tensorFlow. *IEEE Trans. Visualization and Computer Graphics*, 2018, 24(1): 1–12. DOI: 10.1109/TVCG.2017.2744878.

[28] Lai L B, Qing Z, Yang Z Y, Jin X, Lai Z M, Wang R, Hao K Z, Lin X M, Qin L, Zhang W J, Zhang Y, Qian Z P, Zhou J R. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment*, 2019, 12(10): 1099–1112. DOI: 10.14778/3339490.3339494.

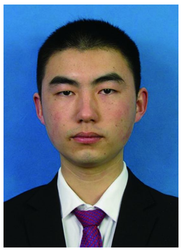
[29] Chen R, Li S S, Li Z. From monolith to microservices: A dataflow-driven approach. In *Proc. the 24th Asia-Pacific Software Engineering Conference*, Dec. 2017, pp.466–475. DOI: 10.1109/APSEC.2017.53.

[30] Zhang J Q, Xiong W W, Liu Y, Park S, Zhou Y Y, Ma Z Q. ATDetector: Improving the accuracy of a commercial data race detector by identifying address transfer. In *Proc. the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2011, pp.206–215. DOI: 10.1145/2155620.2155645.

[31] Abdulla P A, Arora J, Atig M F, Krishna S. Verification of programs under the release-acquire semantics. In *Proc. the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2019, pp.1117–1132. DOI: 10.1145/3314221.3314649.

[32] Inverso O, Trubiani C. Parallel and distributed bounded model checking of multi-threaded programs. In *Proc. the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2020, pp.202–216. DOI: 10.1145/3332466.3374529.

Long Zheng is an associate professor in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. He received his Ph.D. degree in computer science from HUST, Wuhan, in 2016. His research interests include program analysis, runtime systems, and configurable computer architecture with a particular focus on graph processing.


Yang Li is a Master student in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. He received his B.E. degree in computer science from HUST, Wuhan, in 2018. His research interests include graph processing and multi-threaded programs.

Jie Xin is a Master student in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. He received his B.E. degree in computer science from HUST, Wuhan, in 2019. His current research interests include graph processing and in-memory computing.

Hai-Feng Liu is currently a Master student in the School of Computer Science and Technology at Huazhong University of Science and Technology (HUST), Wuhan. He received his B.E. degree in computer science from Wuhan University, Wuhan, in 2020. His research interests include in-memory computing and resistive random-access memory.

Ran Zheng received her M.S. and Ph.D. degrees in computer science from Huazhong University of Science and Technology (HUST), Wuhan, in 2002 and 2006, respectively. She is an associate professor of computer science and engineering at HUST, Wuhan. Her research interests include distributed computing, cloud computing, high-performance computing, and their applications.

Xiao-Fei Liao received his Ph.D. degree in computer science and engineering from Huazhong University of Science and Technology (HUST), Wuhan, in 2005. He has served as a reviewer for many conferences and journal papers. His research interests are in the areas of system software, P2P systems, cluster computing, and streaming services. He is a senior member of CCF and a member of IEEE.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST), Wuhan. Jin received his Ph.D. degree in computer engineering from HUST, Wuhan, in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz, Chemnitz. Jin worked at University of Hong Kong, Hong Kong, between 1998 and 2000, and as a visiting scholar at University of Southern California, Los Angeles, between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. Jin is a fellow of CCF and IEEE, and a life member of ACM. He has co-authored 22 books and published over 900 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.