Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 205827, 14 pages
http://dx.doi.org/10.1155/2015/205827

Research Article

An Efficient Algorithm for On-the-Fly Data Race Detection Using an Epoch-Based Technique

Ok-Kyoon Ha[1] and Yong-Kee Jun[2]

[1]*Engineering Research Institute, Gyeongsang National University, 501 Jinju-daero, Jinju, Gyeongsangnam-do 660-701, Republic of Korea*
[2]*Department of Informatics, Gyeongsang National University, 501 Jinju-daero, Jinju, Gyeongsangnam-do 660-701, Republic of Korea*

Correspondence should be addressed to Ok-Kyoon Ha; jassmin@gnu.ac.kr

Received 23 April 2015; Accepted 21 June 2015

Academic Editor: Rajiv M. Gupta

Data races represent the most notorious class of concurrency bugs in multithreaded programs. To detect data races precisely and efficiently during the execution of multithreaded programs, the epoch-based FastTrack technique has been employed. However, FastTrack has time and space complexities that depend on the maximum parallelism of the program to partially maintain expensive data structures, such as vector clocks. This paper presents an efficient algorithm, called *i*FT, that uses only the epochs of the access histories. Unlike FastTrack, our algorithm requires $O(1)$ operations to maintain an access history and locate data races, without any switching between epochs and vector clocks. We implement this algorithm on top of the Pin binary instrumentation framework and compare it with other on-the-fly detection algorithms, including FastTrack, which uses a state-of-the-art happens-before analysis algorithm. Empirical results using the PARSEC benchmark show that *i*FT reduces the average runtime and memory overhead to 84% and 37%, respectively, of those of FastTrack.

1. Introduction

Synchronization in parallel or multithreaded programs is an enforcing mechanism used to coordinate thread execution and manage shared data in various computational systems, including HPC (High Performance Computing). However, multithreaded programs may contain synchronization defects such as data races, which occur when two concurrent threads access a shared memory location without explicit synchronization, and at least one of them is a write. It is well known that data races are the hardest defect to handle in multithreaded programs, because of their nondeterministic interleaving of concurrent threads [1–4].

Dynamic techniques for detecting data races are usually classified into postmortem methods [4, 5], which analyze traced information or replay the program after execution, and on-the-fly methods, which use one of the following techniques: *happens-before analysis* (like FastTrack [6], SigRace [7], Dijit$^+$ [3], ThreadSanitizer [8], etc. [9–13]), *lockset analysis* (like Eraser [14]), or *hybrid analysis* (like VisualThread [15], Hegrind$^+$ [16–18], MultiRace [19], AccuLock [20], RaceTrack [21], etc. [22]).

The main drawback of dynamic detection techniques is the additional overhead of monitoring program execution and analyzing every conflicting memory operation. A *sampling* approach was introduced to solve the overhead problem of dynamic data race detection. Sampling-based techniques [23–25] can be performed efficiently when testing multithreaded programs via local thread burst-sampling [24] or a global execution time sampling strategy [23]. Although they provide significantly reduced runtime overheads, these techniques are still ineffective in detecting data races when the sampling rates are low.

FastTrack is a state-of-the-art happens-before algorithm and is an improved version of the Djit$^+$ algorithm with *vector clocks* (VCs) [26, 27]. This technique exploits the idea that full generality of VCs is often unnecessary for data race detection. The technique replaces heavyweight VCs with a lightweight identifier, called an *epoch*, that uses only the tuple of the clock value and the thread id.

Epoch-based happens-before analysis decreases the runtime and memory overhead of almost all VC operations from $O(n)$ to $O(1)$ in the detection of data races, where n designates the maximum number of simultaneously active threads during an execution. However, FastTrack requires a time and space overhead of $O(n)$ for the shared read accesses to shared memory locations. Therefore, the overhead problem still exists, because the small fraction of shared read accesses make it difficult to dynamically analyze programs with a large number of concurrent threads [13].

This paper presents an efficient algorithm, called *i*FT, that uses only epochs to detect data races. Thus, *i*FT represents an improvement over the FastTrack method. Our algorithm maintains only two epochs of earlier read accesses to shared memory locations, instead of the full VCs, using the *left-of-relation* [11]. Thus, it requires only $O(1)$ runtime and memory overhead to maintain the access history and locate data races, without any switching between epochs and VCs, unlike FastTrack. Furthermore, the technique is guaranteed to report a subset of data races detected by FastTrack.

We implement the new algorithm on top of the Pin instrumentation framework [28], which uses a just-in-time (JIT) compiler to recompile target program binaries for dynamic instrumentation. To compare the accuracy of *i*FT for on-the-fly data race detection, we also implement two other detection algorithms, Djit$^+$ and FastTrack, on top of the same framework, and employ the same optimized VC primitives. We compare the efficiency of *i*FT with Djit$^+$ and FastTrack, which use a happens-before analysis to detect data races. The experimental results on C/C++ benchmarks using Pthreads show that our algorithm reduces the runtime and memory overheads compared with the other algorithms, while soundly detecting similar data races to FastTrack.

In summary, the contributions of our work are as follows:

(i) *i*FT provides a significant improvement in efficiency, exhibiting an $O(1)$ runtime and memory overhead for each access history, whereas FastTrack requires $O(n)$ VC operations.

(ii) *i*FT matches the well-established precision of FastTrack, although it uses only two epochs instead of the full VCs for earlier read accesses to shared memory locations.

(iii) *i*FT reduces the average runtime and memory overhead to 84% and 37%, respectively, of those of FastTrack.

The remainder of this paper is organized as follows. Section 2 discusses important concepts of happens-before analysis with VCs, and Section 3 introduces the FastTrack algorithm and its limitations. We present our improved algorithm in Section 4 and evaluate it empirically in Section 5 by comparing with existing techniques for data race detection. We introduce some related work in Section 6 and conclude our argument in Section 7.

2. Background

On-the-fly methods of detecting data races typically use VCs to precisely analyze the happens-before relation. This section presents important rules for allocating VCs to the concurrent thread segments introduced in this paper and describes how VCs represent the happens-before relation during the execution of multithreaded programs.

2.1. Execution of Multithreaded Programs. In this work, we consider multithreaded programs using the POSIX thread standard (Pthread) as a model of concurrent threads. Pthread is widely used not only on C/C++ applications, but also on many Unix-like operating systems (Linux, Solaris, Mac OS, FreeBSD, etc.), because it provides various APIs and libraries for creating, manipulating, and synchronizing threads.

In a multithreaded program, a block of thread T that is partially serially executed is represented as a *thread segment*, denoted by t. Thus, a thread can be represented as a set of thread segments, denoted by $T = t_1, t_2, \ldots, t_n$ $(n \geq 1)$. A thread segment t is delimited by *thread operations* that can take one of the following forms:

(i) $init(t)$ models the creation of a thread segment t and the start of the execution of thread T.

(ii) $fork(t, u)$ models the creation of a thread segment u from the current thread segment t and the start of a new thread segment t' on the same thread T.

(iii) $join(t, u)$ models the termination of a thread segment u and the creation of a new thread segment t' on the same thread T from the current thread segment t.

A thread segment t contains a finite sequence σ that consists of at least one event e, denoted by $\sigma = e_1, e_2, \ldots, e_n$. σ_t denotes the sequence of events generated on a thread segment t. An event takes one of the following forms:

(i) *Access Events* $read(x)$ and $write(x)$. The former models the reading of a shared memory location x, and the latter simulates the updating of x.

(ii) *Mutual Exclusion Events* $acq(t, l)$ and $rel(t, l)$. The former models the acquisition of a lock l to enter a critical section. The latter models the release of a lock l to leave a critical section and the start of a new thread segment t' on the same thread T.

(iii) *Condition Variable Synchronization Events* $wait(t, v)$ and $sig(u, v)$. The former models the wait for condition variable v until another thread wakes v and the subsequent start of a new thread segment t' on the same thread T. The latter models the wake-up of a thread waiting on v and the start of a new thread segment u' on the same thread U.

(iv) *Barrier Event* $barrier(t, b)$. This models the waiting of multiple threads until the number of waiting threads is b and the start of a new thread segment on each of the waiting threads.

In this work, we consider the above thread operations and events as *synchronization primitives* rather than access events.

2.2. VC-Based Happens-Before Analysis. Happens-before analysis uses a representation of Lamport's happens-before relation [27] to determine the logical concurrency between two thread segments. According to this relation, if a thread segment t must happen at an earlier time than another thread segment u, t *happens before* u or σ_t *happens before* σ_u, denoted by $t \xrightarrow{hb} u$ or $\sigma_t \xrightarrow{hb} \sigma_u$. If neither $t \xrightarrow{hb} u$ nor $u \xrightarrow{hb} t$ is satisfied, we say that t is *concurrent with* u or σ_t is *concurrent with* σ_u, denoted by $t \parallel u$ or $\sigma_t \parallel \sigma_u$.

VCs are widely used to analyze the happens-before relation $\xrightarrow{hb}$, because they can inform the execution order of thread segments and the synchronization order of thread operations and events. A vector clock VC: $Tid \rightarrow Nat$ records a clock value c for each thread while the program is executing. Thus, thread segment t maintains a VC $C_t = \langle c_1, \ldots, c_n \rangle$, which has n entries if the maximum number of active threads in the execution of a multithreaded program is n. The VC of each thread segment is partially ordered ($\sqsubseteq$) pointwise, with a minimum element $\langle 0, \ldots, 0 \rangle$ and associated synchronization primitives that define pointwise maximums. For instance, the entry $C_t[u]$ for any thread segment u stores the latest clock value of u that happened before the current synchronization primitive of t.

During program execution, the VCs of the thread segments are maintained according to the following rules:

(i) *init*(t)

$$\forall_i : C_t[i] \longleftarrow 0;$$
$$C_t[t] \longleftarrow 1;$$

(ii) *fork*(t, u)

$$Init(u);$$
$$\forall_i : C_u[i] \longleftarrow \max\{C_t[i], C_u[i]\};$$
$$C_t[t] \longleftarrow C_t[t] + 1;$$

(iii) *join*(t, u)

$$\forall_i : C_t[i] \longleftarrow \max\{C_t[i], C_u[i]\};$$
$$C_u[u] \longleftarrow C_u[u] + 1;$$

(iv) *acq*(t, l)

$$\forall_i : C_t[i] \longleftarrow \max\{C_t[i], C_l[i]\},$$
where C_l is a vector clock for each lock l;

(v) *rel*(t, l)

$$C_t \longleftarrow C_t[t] + 1;$$
$$\forall_i : C_l[i] \longleftarrow \max\{C_t[i], C_l[i]\}.$$

The other synchronization events, *wait*(t, v), *sig*(u, v), and *barrier*(t, b), can be modeled with the *join*() operation.

Figure 1 represents a multithread execution with synchronization primitives as a directed acyclic graph, called a Partial Order Execution Graph (POEG) [10, 11]. In the POEG, a vertex is either a thread operation or a synchronization event, and an arc represents a logical thread segment started by the synchronization primitives. The dashed lines indicate the synchronization order in the execution of the program. The events r and w, represented by small disks on the arcs, denote read and write events at a shared memory location, respectively. The numbers attached to each thread segment and event name indicate an observed order, and the VCs are allocated for each thread segment by the above rules.

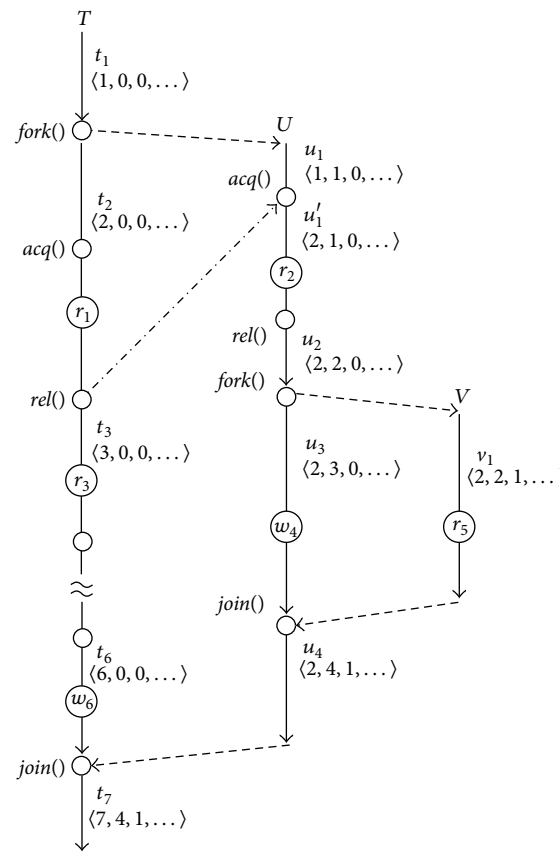


Figure 1: An example of multithread execution with synchronization primitives.

Using the VCs of each thread segment, we simply analyze the happens-before relation between any two thread segments. If the clock value of a thread segment t is less than or equal to the corresponding clock value of another thread segment u, we can conclude that t happens before u. Otherwise, t is concurrent with u. Formally,

$$t \xrightarrow{hb} u \equiv \sigma_t \xrightarrow{hb} \sigma_u \equiv C_t[t] \leq C_u[t]$$
$$t \parallel u \equiv \sigma_t \parallel \sigma_u \equiv \left(C_t[t] > C_u[t] \vee C_t[u] < C_u[u]\right). \quad (1)$$

Obviously, $C_t[t] \leq C_u[t]$ means that thread segment u was synchronized from an earlier thread segment t by one of the synchronization primitives. Then, C_t is partially ordered with C_u, denoted by $C_t \sqsubseteq C_u$, and is never involved in any race. Finally, the happens-before analysis locates a data race during the execution of a multithreaded program whenever any two events on two concurrent thread segments access a shared memory location, and at least one of the events is a write.

Definition 1. Given two access events e_t and e_u to a shared memory location from two distinct thread segments t and u, respectively, if the two events are not synchronized (i.e., neither $C_t \sqsubseteq C_u$ nor $C_u \sqsubseteq C_t$) and at least one of the events is a write, there exists a data race between e_t and e_u.

For example, in Figure 1, consider two events r_3 and w_4 on two different thread segments t_3 and u_3, respectively. The two events constitute a data race, because neither $C_{t3} \sqsubseteq C_{u3}$ nor $C_{u3} \sqsubseteq C_{t3}$ is satisfied, as $C_{t3}[t] = 3 > C_{u3}[t] = 2$ and $C_{t3}[u] = 0 < C_{u3}[u] = 3$, and therefore $t_3 \parallel u_3$.

3. FastTrack Algorithm

VC-based happens-before techniques, such as Djit$^+$ [3], obviously require $O(n)$ space to maintain the VCs for each thread segment and access history and also require $O(n)$ time for VC operations (e.g., join, copy, and comparison).

FastTrack [6], which improves on Djit$^+$, exploits the insight that the full generality of VCs is often unnecessary for data race detection. The key ideas behind this insight are as follows: (1) all writes to a shared memory location x are totally ordered by a happens-before analysis, which assumes no data races have been detected on x so far, and (2) writing to x could potentially conflict with the last read of x performed by any other thread, although reads are not totally ordered, even in race-free programs. By exploiting these results, FastTrack replaces heavyweight VCs with a lightweight identifier for a thread segment, called an *epoch*, using only the tuple of clock value c ($\equiv C_t[t]$) and thread id t, denoted by $c@t$. Thus, FastTrack reduces the runtime and space overhead of almost all VC operations from $O(n)$ to $O(1)$ in the detection of data races.

For a shared memory location x, the FastTrack algorithm defines an access history using two entries:

(i) R_x: it records a VC for all concurrent read events or an epoch for the last read event of x.

(ii) W_x: it records only an epoch for the last write event to x.

FastTrack reports data races by analyzing $\overset{hb}{\longrightarrow}$ and simply maintains epochs or VCs by updating the access histories. For the algorithm, some notions are used to analyze $\overset{hb}{\longrightarrow}$ using the epoch. The function $E(t)$ is shorthand for $c@t$, and $E(t) \preceq VC$ denotes that the epoch $E(t)$ happens before a vector clock VC, where $E(t) \preceq VC$ if and only if $c \leq VC[t]$.

When a new event e_i occurs on thread segment t, the algorithm for reporting data races and maintaining each entry is as follows.

Upon a Read Event of x by Thread t

(1) If the epoch of the current e_i is the same as that of R_x, $R_x = E(t)$, the algorithm takes no action.

(2) If $R_x \neq E(t)$, then the algorithm checks $W_x \preceq C_t$ to report a data race between an earlier write event and e_i.

(3) If $R_x \preceq C_t$ is satisfied, only $E(t)$ is kept in R_x. Otherwise, $C_t[t]$ is updated to R_x, which maintains a full VC.

Upon a Write Event to x by Thread t

(1) If the epoch of the current e_i is the same as that of W_x, $W_x = E(t)$, then the algorithm takes no action.

(2) If $W_x \neq E(t)$, then the algorithm checks $W_x \preceq C_t$ to report a data race between an earlier write event and e_i.

(3) If there exists only one epoch in R_x, then the algorithm checks $R_x \preceq C_t$ to report a data race between an earlier read event and e_i. Otherwise, the algorithm checks $R_x \sqsubseteq C_t$ for a full VC maintained in R_x.

(4) The previous epoch or VC is removed from R_x, and $E(t)$ is inserted into W_x.

Table 1: Access history states for detecting data races in Figure 1 using the FastTrack algorithm.

e_i	R_x	W_x	*Races*
r_1	$2@t_{\text{id}}$	$\perp_e$	
r_2	$\langle 2, 1, 0, \ldots \rangle$	$\perp_e$	
r_3	$\langle 3, 1, 0, \ldots \rangle$	$\perp_e$	
w_4	$\perp_e$	$3@u_{\text{id}}$	$r_3 - w_4$
r_5	$1@v_{\text{id}}$	$3@u_{\text{id}}$	$w_4 - r_5$
w_6	$\perp_e$	$6@t_{\text{id}}$	$w_4 - w_6, r_5 - w_6$

Table 1 explains how the FastTrack algorithm reports data races and manages the access history during the execution of the program shown in Figure 1. Initially, W_x starts from $\perp_e$, indicating that the shared memory location x has not yet been written. When the first read event r_1 occurs on thread segment t_2, the epoch $2@t_{\text{id}}$ is recorded in R_x instead of a full VC, where t_{id} indicates the thread id for t_2. When the second read r_2 on thread segment u_1 accesses x, r_2 shares x with the first read event r_1, because $t_2 \parallel u_1$, where we say that x is in a *Read Shared* state. In this state, as read may consist of either one or more data races with a later write event, the VCs of all shared reads of x are kept in R_x. Thus, R_x switches to a VC representation $\langle 2, 1, 0, \ldots \rangle$ to record the clocks of the last reads by the two thread segments in Table 1. With this adaptive switching between epochs and VCs in R_x, FastTrack greatly reduces the overhead of the $O(n)$ VC operations.

When read event r_3 occurs on t_3, $C_t[t] = 3$ is directly updated in the corresponding entry of R_x, although R_x maintains a VC for the Read Shared state. Thus, the updating takes $O(1)$ time. A data race $\{r_3\text{-}w_4\}$ is reported because Definition 1 is satisfied (i.e., neither $R_x \sqsubseteq C_{u3}$ nor $C_{u3} \sqsubseteq R_x$ is true) when a write event to x occurs. The VC of prior read events in R_x is removed by resetting R_x to $\perp_e$, and the epoch for w_4, $3@u_{\text{id}}$, is stored in W_x. When a read of x occurs on v_1, only the epoch of r_5 is kept in R_x, because the read event is not shared with any others, and a data race $\{w_4\text{-}r_5\}$ is reported. Finally, three concurrent events, w_4, r_5, and w_6, give rise to

two data races, $\{w_4\text{-}w_6, r_5\text{-}w_6\}$, because $E(u_3) \preceq C_{t6}$ and $E(v_1) \preceq C_{t6}$ are not satisfied.

A common problem with using VCs for happens-before analysis is the space and time overhead, which depends on the number of threads in the multithreaded programs, whereas the FastTrack algorithm provides a significant performance improvement over the lockset analysis by utilizing the lightweight epoch clock. Moreover, it suggests the design of a hybrid technique with both precision and efficiency, such as AccuLock [20]. However, there is further room for improvement, because the algorithm requires $O(n)$ VC operations to guarantee no loss of precision when shared data enters the Read Shared state, such as r_2 and r_3 in Figure 1. Therefore, the overhead problem still exists, because the shared read accesses make it difficult to dynamically analyze programs with a large number of concurrent threads.

4. Efficient Data Race Detection

FastTrack precisely reports data races with significantly improved performance, because epochs require only a constant space and a constant time for almost all VC operations. However, the algorithm still needs VC operations whenever a shared memory location has shared read events on concurrent thread segments. As this situation makes it impossible to dynamically analyze programs with a large number of concurrent threads [13], the overhead problem potentially exists, with the space overhead being more critical than the time overhead. Thus, we efficiently improve the FastTrack algorithm to reduce this overhead problem.

Our improved FastTrack (*i*FT) algorithm reports data races in a constant amount of time and space, even in the worst case, because it maintains only two epochs instead of full VCs for R_x using the *left-of-relation*. The notion of the left-of-relation was originally suggested by Mellor-Crummey [11]. Mellor-Crummey's technique maintains two concurrent read events in an access history to detect data races with a write event. Techniques based on the left-of-relation guarantee that a program is free of data races, although it maintains only two read events in each access history, because it locates at least one data race (if any exist). However, Mellor-Crummey's technique does not support synchronization primitives other than fork/join operations, such as thread locking and wait-signals. Moreover, the left-of-relation does not apply to VC-based detectors, because VCs cannot analyze the logical position of thread segments, unlike Mellor-Crummey's OS labeling [11].

We simply define a left-of-relation that is a partial ordering of two concurrent thread segments t_i and t_j for two distinct events e_i on t_i and e_j on t_j in an execution graph, such as the POEG of Figure 1, and the events are not related to $e_i \preceq e_j$. To apply the left-of-relation to the *i*FT algorithm, we use a breadth value b instead of the thread id T_{id} of the original FastTrack algorithm. The breadth value b is produced by performing a left-to-right preorder numbering or an *English Order* numbering of the EH labeling scheme [29] and is used to identify the position of a current thread considering its sibling threads. If a thread segment t_i precedes another thread segment t_j and $t_i \parallel t_j$ in an execution of a multithreaded program, b_i for t_i is less than b_j for t_j.

Thus, an epoch $E(t_i)$ of thread segment t_i is redefined as the tuple of clock value c_i and breadth value b_i, denoted by $c_i@b_i$. Now, the left-of-relation between any two thread segments is simply analyzed by comparing their breadth values from each epoch.

Definition 2. Given two read events e_i and e_j to a shared memory location on two concurrent thread segments t_i and t_j, respectively, if b_i for $E(t_i)$ is less than b_j for $E(t_j)$, one says that e_i is *left of* e_j, denoted by $e_i \prec e_j$. Formally,

$$e_i \prec e_j = \begin{cases} \mathfrak{T}(e_i) = \mathfrak{T}(e_j) = Read \wedge \\ t_i \parallel t_j \wedge b_i < b_j, \end{cases} \tag{2}$$

where $\mathfrak{T}(e_i)$ represents the event type (read or write) of e_i. By applying the left-of-relation, we employ the *leftmost* event, denoted by e_l, and *rightmost* event, denoted by e_r, concepts to maintain only two concurrent events in R_x. We use R_{lx} and R_{rx} to denote the leftmost event and rightmost event, respectively, in R_x. If the current event e_i satisfies $e_i \prec e_l \prec e_r$, e_i is the leftmost event. This event is recorded in R_{lx} instead of e_l, where the prior event e_l always satisfies the left-of-relation with e_r in R_{rx}; therefore, $e_l \prec e_r$. Similarly, the current event e_i is the rightmost event and is recorded in R_{rx} instead of e_r, if it satisfies $e_l \prec e_r \prec e_i$.

We now provide a detailed description of how *i*FT locates three kinds of data races for concurrent events: *read-write races*, *write-write races*, and *write-read races*.

Read-Write Races. Detection is possible because a write event to a shared memory location x can conflict with prior read events of x performed by any other thread. To detect read-write races, we consider two read states: (1) Exclusive state, where a read event of x is performed exclusively on a thread segment, and (2) Read Shared state, where x has read events that are shared by two or more concurrent thread segments. In the Exclusive state, because read events of x occur on the same thread, they are totally ordered, and the epoch of the last read event is recorded in R_x. Read events of x that are shared by multiple threads are unordered in a read-only manner, and each read event may consist of a data race with a later write event. Thus, if x is in the Read Shared state, two epochs of the two concurrent read events are recorded in R_x by the left-of-relation.

Using R_x, which maintains only two epochs instead of a full VC, *i*FT detects data races as well as FastTrack, because it locates one or two of the read-write data races.

Lemma 3. *If data races exist between earlier reads and a current write event w, iFT locates one or two of those located by FastTrack.*

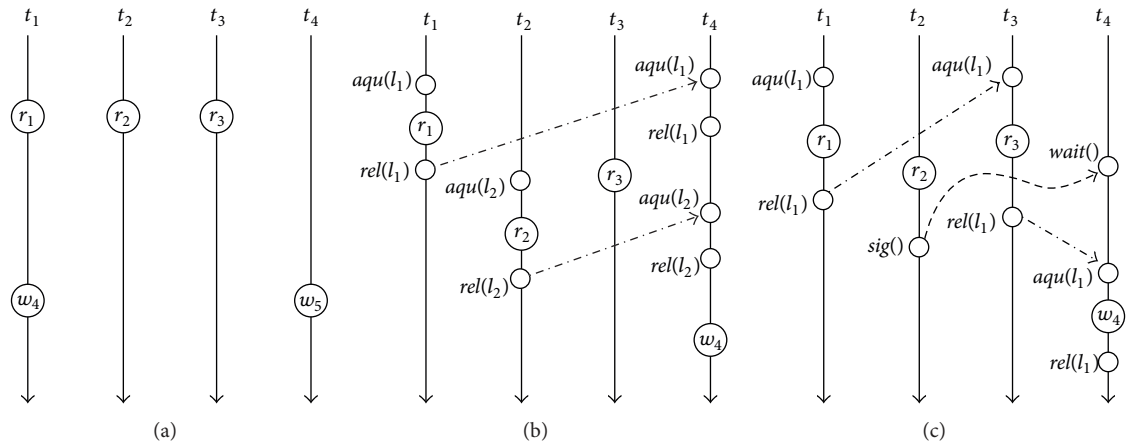


Figure 2: Examples of read-write data races.

Proof. Two distinct shared read events toward x are kept in R_{lx} and R_{rx} by the left-of-relation. Since $R_{lx} \parallel R_{rx}$, we guarantee the following:

(1) If $R_{lx} \xrightarrow{hb} w$ and $R_{rx} \parallel w$, then *i*FT reports a data race between R_{rx} and w, because $R_{lx} \preceq w$, and neither $R_{rx} \sqsubseteq w$ nor $w \sqsubseteq R_{rx}$ is satisfied.

(2) If $R_{lx} \parallel w$ and $R_{rx} \xrightarrow{hb} w$, then *i*FT reports a data race between R_{lx} and w, because $R_{rx} \preceq w$, and neither $R_{lx} \sqsubseteq w$ nor $w \sqsubseteq R_{rx}$ is satisfied.

(3) If $R_{lx} \parallel w$ and $R_{rx} \parallel w$, then *i*FT reports two data races between w and both shared read events.

(4) If $R_{lx} \xrightarrow{hb} w$ and $R_{rx} \xrightarrow{hb} w$, then *i*FT fails to report any data races. □

Figure 2 shows three examples of read-write data races during the execution of a multithreaded program with nondeterministic interleaving of concurrent threads. In Figure 2(a), three shared read events, r_1, r_2, and r_3, happen before the two write events, w_4 and w_5. The leftmost event r_1 and the rightmost event r_3 are kept in R_{lx} and R_{rx}, respectively, by the left-of-relation. Thus, *i*FT can report a data race between r_3 and w_4, because $r_1 \xrightarrow{hb} w_4$ and $r_3 \parallel w_4$. When w_5 occurs on thread segment t_4 that is concurrent with the others, *i*FT reports two data races $\{r_1\text{-}w_5, r_3\text{-}w_5\}$.

In Figure 2(b), t_1 and t_4 are synchronized by a lock variable l_1, and t_2 is also synchronized with t_4 by a lock variable l_2. For the execution of Figure 2(b), *i*FT records two read events r_1 and r_3 in R_{lx} and R_{rx}, respectively. It reports only the data race $\{r_3\text{-}w_4\}$ between R_{rx} and w_4, because $R_{lx} \preceq w_4$ is satisfied. Therefore, $r_1 \xrightarrow{hb} w_4$ by the synchronization between t_1 and t_4. *i*FT records r_2 from t_2 in R_{lx} instead of r_1 if the acquiring lock l_2 is reserved, because $r_1 \xrightarrow{hb} r_2$ by the thread interleaving $t_1 \rightarrow t_4 \rightarrow t_2$. Finally, *i*FT reports two read-write data races $\{r_2\text{-}w_4, r_3\text{-}w_4\}$ for the execution.

In Figure 2(c), there are two kinds of synchronization events, locking and a signal-wait. Because $r_1 \xrightarrow{hb} r_3$ is satisfied by lock variable l_1, *i*FT records r_3 in R_{lx} as the leftmost event, and r_2 is recorded in R_{rx}. Thus, *i*FT locates no data races, because $r_3 \xrightarrow{hb} w_4$ by the acquiring lock l_1, and $r_2 \xrightarrow{hb} w_4$ by the signal-wait event. If a pair of wait and signal events does not occur between t_2 and t_4, *i*FT obviously locates the data race $\{r_2\text{-}w_4\}$, as it analyzes that the rightmost event r_2 is concurrent with w_4.

Lemma 4. *If data races exist between R_x and a current write event, the races located by iFT are a subset of those located by FastTrack.*

Proof. Suppose that the same fixed program execution order is provided to both analyses. Let I_{race} (F_{race}) be the set of races located by *i*FT (FastTrack), and let $I_{Rx}(e)$ ($F_{Rx}(e)$) be the read events recorded in R_x by *i*FT (FastTrack). Because $I_{Rx}(e) \in F_{Rx}(e)$ in the execution order, we guarantee the following:

(1) If $F_{\text{race}} = \phi$, then $I_{\text{race}} = \phi$ is satisfied because it is impossible to satisfy $I_{\text{race}} \neq \phi$.

(2) If $F_{\text{race}} \neq \phi$, then $I_{\text{race}} \neq \phi$ is satisfied because $I_{\text{race}} = \phi$ cannot be satisfied by Lemma 3.

Therefore, $I_{\text{race}} \subseteq F_{\text{race}}$ is satisfied. □

For example, in Figure 2(a), the three data races $\{r_3\text{-}w_4, r_1\text{-}w_5, r_3\text{-}w_5\}$ located by *i*FT are a subset of the five data races $\{r_2\text{-}w_4, r_3\text{-}w_4, r_1\text{-}w_5, r_2\text{-}w_5, r_3\text{-}w_5\}$ located by FastTrack.

Write-Write Races. These involve two concurrent write events to x. All write events to x are totally ordered, with the assumption that no data races have been detected on x. Thus, *i*FT records the epoch of the write event in W_x and locates a write-write race between W_x and a later write event to x by analyzing the epoch of W_x and the current VC of the write event, $W_x \preceq C_t$.

Write-Read Races. These involve a write event to x that is concurrent with a later read event of x. *i*FT locates such a data race by analyzing $W_x \preceq C_t$.

```
(01) ReadCheck(x, t)
(02)    if E(t) = any epoch kept in R_x then return;
(03)    if W_x ⋠ C_t then Report a data race;
(04)    MaintainAH(R_x, E(t));
(05) End ReadCheck

(01) WriteCheck(x, t)
(02)    if E(t) = W_x then return;
(03)    if W_x ⋠ C_t or R_x ⋠ C_t then Report a data race;
(04)    MaintainAH(W_x, E(t));
(05)    R_x ← ϕ;
(06) End WriteCheck

(01) MaintainAH(AH, epoch)
(02)    if AH = ϕ or IsMostL(AH, epoch) or IsMostR(AH, epoch)
(03)       or IsOrdered(AH, epoch) then AH ← epoch;
(04) End MaintainAH
```

Algorithm 1: The *i*FT algorithms.

Lemma 5. *If FastTrack locates a write-write race or a write-read race during the execution of a program, iFT can locate the data race from the same fixed execution.*

Proof. Let $I_{Wx}(e)$ ($F_{Wx}(e)$) be a write event recorded in W_x by *i*FT (FastTrack). Then, $F_{\text{race}} = I_{\text{race}}$ holds, because $I_{Wx}(e) = F_{Wx}(e)$ in the execution order, and both analyses employ only $W_x \preceq C_t$ to analyze $\xrightarrow{hb}$. □

Algorithm 1 presents the pseudocode for *i*FT, which consists of three algorithms: *ReadCheck*(), *WriteCheck*(), and *MaintainAH*(). *ReadCheck*() and *WriteCheck*() mainly focus on filtering events, reporting data races, and maintaining an access history AH for a shared memory location x whenever an event e_i on thread segment t accesses x. To report data races, we use the inversion of $\preceq$, denoted by $\not\preceq$, to catch instances where the current event is concurrent with a prior event. In *ReadCheck*() and *WriteCheck*, $W_x \not\preceq C_t$ denotes that neither $W_x \preceq C_t$ nor $C_t \preceq W_x$ is satisfied. *IsOrdered*() is used by *MaintainAH*() to check the happens-before relation between the current event and prior events in AH. *MaintainAH*() manages access histories for every x and employs *IsMostL*() and *IsMostR*() to maintain only two concurrent events in R_x by applying the left-of-relation.

Table 2 shows the changing state of an access history for detecting the data races appearing in Figure 1 using the *i*FT algorithm, where we assume that the breadth values are allocated as $T = 0$, $U = 1$, and $V = 2$. In the figure, the epoch of read event r_1 on t_2, 2@0, is recorded in R_x, as the read event of x is performed exclusively. When the rightmost read r_2 occurs on u_1, x enters the Read Shared state. The epoch of r_2 (1@1) is recorded with the epoch of r_1, instead of the full VC of FastTrack in Table 1, because $b(r_1) = 0$ is less than $b(r_2) = 1$, and therefore $r_1 \preccurlyeq r_2$. Because r_3 is the last read event on thread T when the event occurs, the epoch of the prior leftmost event r_1 is updated to the epoch of r_3, 3@0. When w_4 occurs on u_3, the data race $\{r_3\text{-}w_4\}$ is reported, as for the FastTrack algorithm. However, *i*FT only compares two epochs in R_x without any VC operations. *i*FT also reports the data race $\{w_4\text{-}r_5\}$ and two data races $\{w_4\text{-}w_6, r_5\text{-}w_6\}$, as does the FastTrack algorithm, when r_5 and w_6 occur. Consequently, the results in Table 2 show that *i*FT detects apparent data races as well as FastTrack, although the new algorithm maintains only two epochs for concurrent read events in R_x.

Table 2: Access history states for detecting data races using the *i*FT algorithm.

e_i	R_x	W_x	*Races*
r_1	**2@0**	$\perp_e$	
r_2	2@0 **1@1**	$\perp_e$	
r_3	**3@0** 1@1	$\perp_e$	
w_4	$\perp_e$	**3@1**	$r_3 - w_4$
r_5	**1@2**	3@1	$w_4 - r_5$
w_6	$\perp_e$	**6@0**	$w_4 - w_6, r_5 - w_6$

Theorem 6. *iFT efficiently and soundly locates data races if it maintains only two epochs in R_x.*

Proof. The *i*FT algorithm has $O(1)$ time and space overheads for detecting data races, because it removes the switching between epochs and VCs for R_x of the FastTrack algorithm by maintaining only two concurrent epochs for the Read Shared state of x. From Lemmas 3, 4, and 5, the algorithm soundly locates data races because it reports a subset including at least one of the data races located by the FastTrack algorithm. □

5. Evaluation

We empirically evaluated the efficiency and precision of *i*FT in comparison with other dynamic detection algorithms that

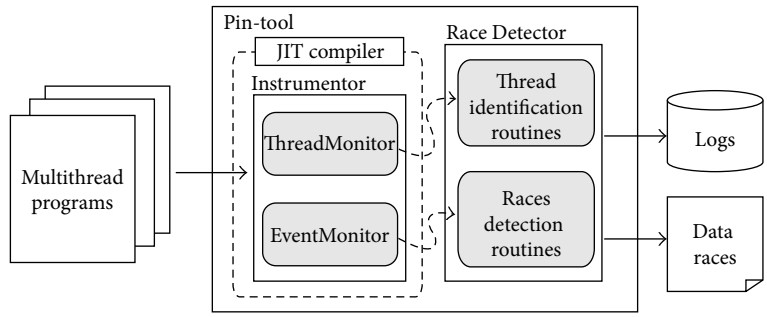


Figure 3: Overall architecture of a data Race Detector.

use the happens-before analysis. The experimental results show that our technique not only soundly reports data races, but also reduces the time and space overhead of data race detection for programs with a large number of concurrent threads.

5.1. Implementation and Experimentation. We implemented the *i*FT algorithm and two other dynamic detection algorithms on top of the Pin instrumentation framework [28], which uses a JIT compiler to recompile target program binaries for dynamic instrumentation. Building a lightweight tool for monitoring memory access is easier with Pin than with other dynamic binary instrumentation frameworks, such as Valgrind [30]. The two algorithms used for comparison are Djit$^+$ [3] (a high performance VC-based happens-before analysis algorithm) and FastTrack [6] (a state-of-the-art happens-before analysis algorithm).

Figure 3 depicts the architecture of the detectors. Each detector consists of an Instrumentor and a Race Detector to report data races during program execution. The Instrumentor consists of two modules: ThreadMonitor and EventMonitor. These, respectively, track thread operations and event instances for every shared memory location considering synchronization primitives. The Race Detector performs the thread identification routines to generate and manage VCs for each active thread segment, as well as the detection routines to report data races.

The thread identification routines employ the VC primitives discussed in Section 2. These are commonly used to analyze the happens-before relation in the detection routines of all algorithms. A lock-free algorithm was used in the detection routines to remove the centralized bottleneck of access histories. Whenever the Instrumentor catches one of the thread operations or events, it calls either the thread identifier routines or the detection routines to add instrumentation at each interesting point of the running target binaries. Because the Instrumentor and Race Detector use only the shadow binaries of the target programs, which are generated by the JIT compiler of the Pin framework, no source code annotation is required to monitor memory access events or synchronization primitives.

To supplement the correct identification of concurrent thread segments, we used a special structural table for each thread. The table consists of four important items of information, the system thread id, Pthread id, Pin thread id, and clock value. The system thread id is the thread id allocated by the operating system, and the Pthread id is allocated by Pthread functions such as `pthread_create()`. The Pin thread id is the logical identifier created in sequence whenever the Pin framework catches a thread start operation. Thus, we employed the Pin thread id as the breadth value b_i of an epoch $(c_i@b_i)$ in the *i*FT algorithm. The clock value is used to form a VC of a thread segment using synchronization primitive operations.

Our experimentation focused on comparing the soundness and the efficiency of on-the-fly data race detection in programs with a large number of concurrent threads. To evaluate the *i*FT algorithm, we compared the data races reported by each detector and measured the execution time and the memory consumed by the execution instances of a set of C/C++ benchmarks using Pthread. For this purpose, we used 12 applications from the PARSEC 2.1 benchmark suite [31]. These target different areas, including HPC, with applications such as data mining, financial analysis, and computer vision. All applications were executed with the default simulation inputs of the PARSEC benchmark suite to produce proper runtime overheads and memory consumption.

Before conducting the experiments, we investigated the benchmark applications in terms of the frequency of access events and synchronization primitives. The results of this analysis with the FastTrack algorithm are given in Table 3. We used `sim-medium` simulation inputs in the execution of each application. In the table, "Same Epoch" means that read/write events to a shared memory location x have been filtered out by FastTrack as they occurred after the first read/write event on the same thread segment. "Exclusive" indicates that only epochs were used to locate data races, because read/write events exclusively accessed x. "Shared" indicates the Read Shared state in which x has shared read events being performed by concurrent thread segments. "VC Scan" indicates that a current write event was compared with R_x when x entered the Read Shared state. Thus, two memory operations, Shared and VC Scan, require VC operations that require $O(n)$ time and space overheads in FastTrack.

From this investigation, we can see that 78.3% of all operations and events were read events and 21.6% were write events. Other operations and events accounted for less than 0.1% of the total. These results reaffirm that almost all parts of data race detection involve tracing access events to shared memory locations, because this accounts for more

Table 3: Analysis of PARSEC benchmarks using FastTrack.

Applications	# of threads	Read (78.3%)			Write (21.6%)		
		Same Epoch	Exclusive	Shared	Same Epoch	Exclusive	VC Scan
blackscholes	9	99.8%	0.2%	0%	99.0%	1.0%	0%
bodytrack	10	94.5%	2.6%	2.9%	87.9%	10.8%	1.3%
canneal	9	87.7%	7.4%	4.9%	69.3%	23.7%	7.0%
dedup	25	90.8%	8.7%	0.5%	70.2%	29.3%	0.5%
facesim	8	89.8%	10.1%	0.1%	94.2%	5.7%	0.1%
ferret	35	94.9%	3.7%	1.4%	75.7%	19.8%	4.5%
fluidanimate	9	84.8%	10.8%	4.4%	90.0%	9.9%	0.1%
raytrace	9	97.2%	2.7%	0.1%	99.6%	0.4%	0%
streamcluster	17	76.1%	23.3%	0.6%	84.0%	15.3%	0.7%
swaptions	9	99.1%	0.5%	0.4%	95.9%	2.2%	1.9%
vips	4	75.8%	24.2%	0%	1.3%	98.7%	0%
x264	64	97.8%	1.8%	0.4%	95.2%	4.8%	0%
Average		90.7%	8.0%	1.3%	80.2%	18.5%	1.3%

than 99% of operations in the benchmarks. Fortunately, the convergence of memory operations is again removed, as there is a possibility that this will affect the tracing of events for data race detection. For example, in the table, 90.7% of read events and 80.2% of write events occurred in the same epoch. VC operations are rarely needed, accounting for an average of only 1.3% of all read/write events. Thus, the switching approach in FastTrack is quite effective in improving the performance of happens-before analysis.

The implementation and experimentation were carried out on a system with two 2.4 GHz Intel Xeon quad-core processors and 32 GB of memory under Linux Kernel 2.6. We installed the most recent version of the Pin framework (Version 2.12), and the applications were compiled with gcc 4.4.4 for all detectors. We used a programmed logging method to measure the execution time and memory consumption of each application. This method uses system files in the `proc` directory, which provides real-time information on the system, including `meminfo`, `iomem`, and `cpuinfo`. The average runtime and memory overheads of all applications were measured for ten executions under each detector. Figure 4 shows the resulting analyzed information, such as thread creation, detected data races, execution time, and memory consumption, during an execution of the `x264` application using our implemented *i*FT detector.

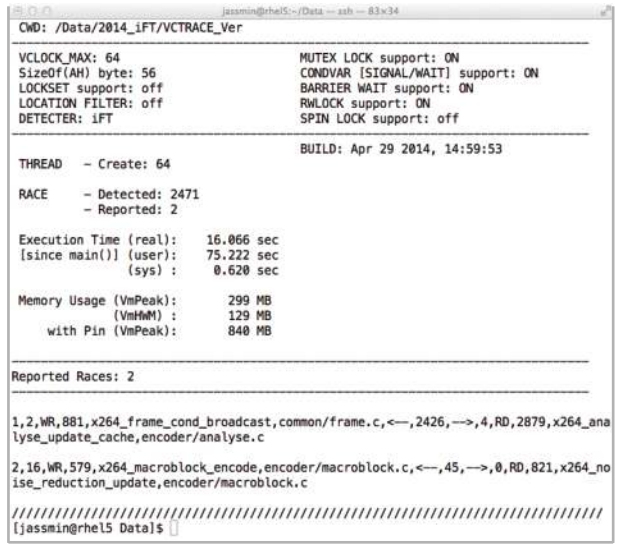


Figure 4: Execution result of the implemented *i*FT detector.

5.2. Results and Analysis

5.2.1. Precision. We acquired the reported data race results to evaluate the precision of *i*FT. Three detectors were applied on the same Pin framework for fair experimentation. All applications of PARSEC benchmark were run with `sim-medium` simulation inputs, and two real applications were run with both of server program and several client programs. The two real applications used for the experimentation are `MySQL` (an open source DBMS) and `Cherokee` (an open source web and server application). These applications were repeatedly tested until each detector had fixed all warnings. The number of data races located by the three detectors is given in Table 4.

All of the detectors reported that there were no data races in six of the applications in the PARSEC benchmarks, `blackscholes`, `dedup`, `facesim`, `raytrace`, `swaptions`, and `vips`. This agrees with prior research [32], which considered an implementation of FastTrack on top of the DynamoRIO instrumentation framework. Djit$^+$ and FastTrack reported exactly the same data races for all applications, as found in [6, 20], because these two detectors are based on identical precision. Similarly, *i*FT reported the same data races as FastTrack, with the exception of the `bodytrack` and `x264` applications.

All the detectors located a data race in `canneal` and `fluidanimate`, which run into user-defined synchronization functions, such as *atomic*() and *barrier_wait*(). They reported two data races in `ferret`; these were caused by a shared counter variable and a shared Boolean flag for a queue in the application. The three detectors reported

Table 4: Number of data races located on the PARSEC benchmark and real applications.

Applications	# of threads	Detected races		
		Djit$^+$	FastTrack	*i*FT
PARSEC				
blackscholes	9	0	0	0
bodytrack	10	8	8	7
canneal	9	1	1	1
dedup	25	0	0	0
facesim	8	0	0	0
ferret	35	2	2	2
fluidanimate	9	1	1	1
raytrace	9	0	0	0
streamcluster	17	4	4	4
swaptions	9	0	0	0
vips	4	0	0	0
x264	64	3	3	2
Real				
MySQL	78	8	8	8
Cherokee	126	7	7	7

four data races for `streamcluster`. These were caused by using the same user-defined synchronization, *barrier_wait*(), and object pointers to a shared structure without explicit synchronization. All of the detectors reported eight data races in `MySQL` due to object pointers to a shared structure without any proper synchronization and shared flags for thread termination. The three detectors located seven data races in `Cherokee`. A data race in `Cherokee` was the result of log corruption similar to a well-known bug in Apache's logging code (Apache bug #25520).

For `bodytrack`, all detectors found six data races, which were caused by the initialization of objects in shared structures without synchronization and the misuse of condition variables. Djit$^+$ and FastTrack also reported two data races involving two kinds of unprotected counter variables for a user-defined *wait-notify* operation, whereas *i*FT reported only one of the data races. *i*FT located two data races for `x264`, caused by two pointers in different functions that were referring to a shared structure and its members. The pointers allowed the shared memory locations to be concurrently accessed by read/write events from each function without any proper synchronization. The other detectors reported three data races, including two detected by *i*FT; the other one was caused by the same bug via a pointer to the same shared structure.

In `bodytrack` and `x264`, shared read events that are not the leftmost or rightmost events can be exempted from relevant events of the data race detection process by our *i*FT algorithm. Hence, *i*FT reported fewer data races for these two applications, and the reported data races were a subset of those given by FastTrack. For example, in the result of `x264`, a prior read access of a shared structure in a file (`frame.c`) was removed from R_x of an *AH*, since a new read access of the same shared structure in another file (`analyse.c`) occurred on the leftmost thread. *i*FT reported only a data race between the leftmost read access and a later write access to the same shared structure in a file (`encorder.c`), whereas FastTrack reported two data races between these read accesses and the later write. However, *i*FT located the missed data race after we had fixed the previously reported data race by using a local pointer variable.

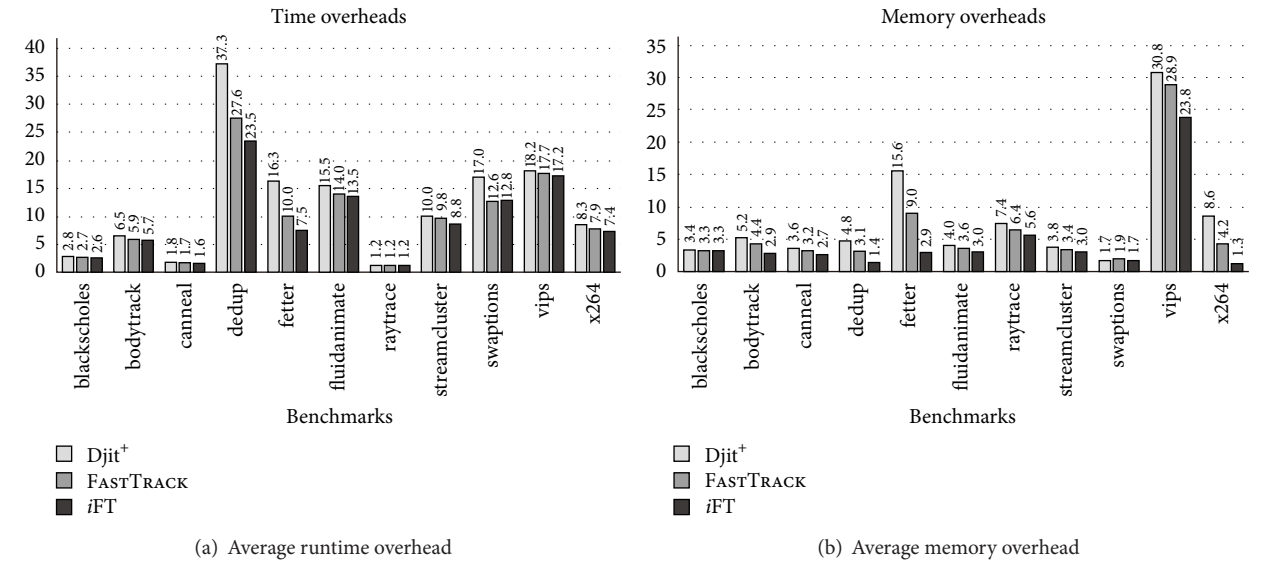
From this experiment, we can conclude that *i*FT is sound, because the precision of the *i*FT algorithm is fixed relevant to the well-established precision of FastTrack.

5.2.2. Efficiency. We measured the runtime and memory consumption of the benchmarks over three detectors to evaluate the efficiency of *i*FT. Figure 5 depicts the measured runtime and memory overhead results for 11 applications of PARSEC with `sim-medium` simulation inputs. The graph shows the average runtime and memory overheads for each of the detectors as a proportion of the original run. Because `facesim` is a representative long-running application that uses a small number of concurrent threads and naturally requires quite high runtime and memory overheads for on-the-fly data race detection, the application was excluded from the efficiency test.

From Figure 5(a), almost all of the *i*FT results are lower than those of the other detectors. *i*FT incurred an average runtime overhead of 8.5x, whereas FastTrack and Djit$^+$ required average runtime overheads of 9.2x and 11.2x, respectively. In particular, *i*FT required explicitly lower runtime overheads for two applications, `dedup` and `ferret`, which use more than 20 active threads during program execution. For instance, *i*FT incurred an average runtime overhead of 23.5x for `dedup`, whereas FastTrack and Djit$^+$ incurred average runtime overheads of 27.6x and 37.3x, respectively. In the case of `ferret`, the incurred runtime overhead of *i*FT was 7.5x, while FastTrack and Djit$^+$ incurred average runtime overheads of 10x and 16x, respectively. Several applications, such as `blackscholes`, `canneal`, and `raytrace`, have lower overheads than the others because of their model of parallelism (e.g., fork-join parallelism).

In Figure 5(b), we see that *i*FT incurred an average memory overhead of 4.3x, whereas FastTrack incurred an average memory overhead of 6.0x. This means that *i*FT reduced the average memory overhead to 58% of that of Djit$^+$ and 72% of that of FastTrack for 11 applications. If we consider the three applications that use several ten dynamic threads, *i*FT incurred an average memory overhead of 1.9x, while FastTrack required an average memory overhead of 5.4x. Thus, the proposed *i*FT reduced the average memory overhead to 37% of that recorded by FastTrack.

We measured average memory consumption for two real applications under our Pin framework. The results of the measurement appear in Figure 6. For the experiments, `MySQL` used 78 multiple threads during 60 seconds for an execution, and 126 threads were used for `Cherokee`. We employed four monitoring steps, Native, Pin-only, Monitoring, and Detecting, to show how many additional overheads were incurred by instrumentation work under Pin framework. Native means the original execution without our Pin



(a) Average runtime overhead

(b) Average memory overhead

Figure 5: Measured runtime and memory overhead results for 11 benchmarks of PARSEC.

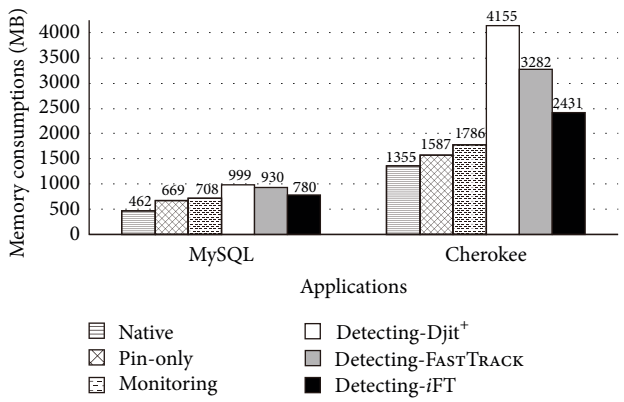


Figure 6: Measured memory consumption for two real applications.

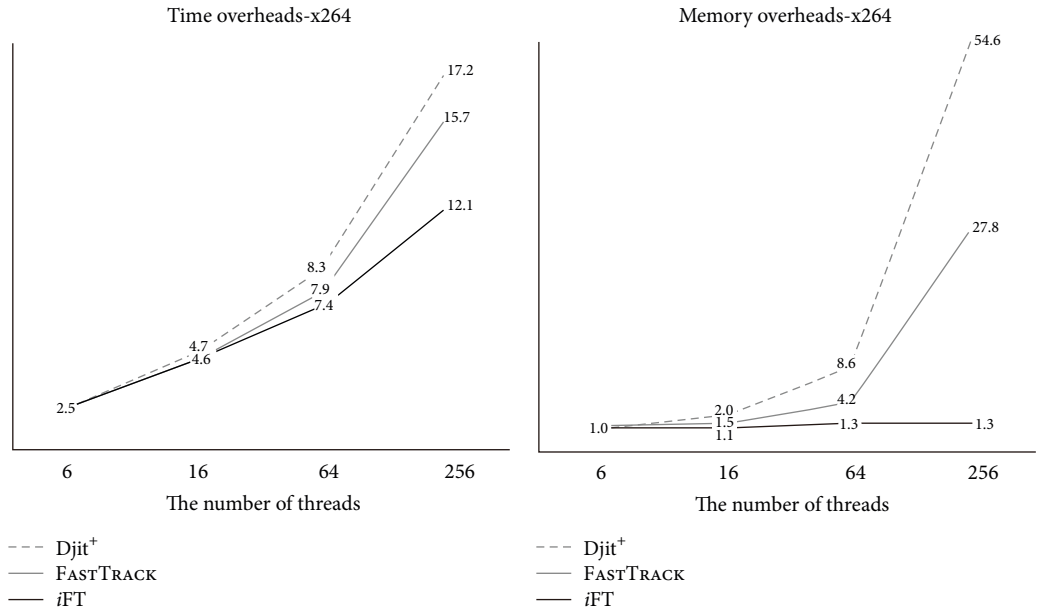
framework, and Pin-only indicates the measured results that the applications were run on the Pin framework without monitoring and instrumentation work. Monitoring means that only the thread executions and memory accesses were traced under the Pin framework. Detecting means that we measured the memory consumption of the execution of the applications under the three detectors that were implemented on top of the Pin framework.

In Figure 6, we see that Pin-only incurred an average memory consumption of 2.2x and Monitoring incurred an average memory consumption of 2.6x. *i*FT incurred an average memory consumption of 2.8x, whereas FastTrack incurred an average memory consumption of 3.6x. This means that *i*FT reduced the average memory consumption to 62% of that of Djit$^+$ and 76% of that of FastTrack for two applications. If we exclude Pin-only step that incurred 1,128 MB in the average case, *i*FT incurred an average memory consumption of 1.7x, while FastTrack required an average memory consumption of 2.3x. For the two real applications, *i*FT reduced the average memory consumption to 49% of FastTrack.

We chose the `x264` application from the PARSEC benchmark for additional comparison, because it employs a different number of concurrent threads to process the virtual pipelined stages for each input frame. In contrast, the other applications use a fixed number of threads, although they use different inputs. The comparison used all six simulation inputs provided by the PARSEC suite, because these lead to an increasing thread size in each input frame.

Figure 7 depicts the measured runtime and memory overhead results for the `x264` application. In the experiment, *i*FT incurred an average runtime overhead of 6.6x, whereas the other detectors averaged more than 8x slowdown. In particular, in the executions with the `sim-large` input (256 threads), *i*FT reduces the runtime overhead to 74% of that of the other detectors. *i*FT performs well in reducing the memory overhead, averaging just 1.3x, whereas the memory overhead of the other detectors increased by a factor of more than 95% relative to that of *i*FT. Under *i*FT, the application ran with `native` input using 1,024 concurrent threads, but the other detectors ran out of memory with the `native` input because of the 32 GB limitation of our system. In this case, *i*FT required a runtime overhead of 11.5x and a memory overhead of 1.6x to locate two data races. It is noteworthy that the distinguished performance of *i*FT is caused by the elimination of the VC operations used in the FastTrack algorithm.

The results in Figure 7 show that *i*FT reduced the memory overhead by 11.4x and gave a speedup of 1.3x compared to the other dynamic detectors. The overheads of *i*FT were similar to those of the other algorithms for small-size inputs, as `x264` uses fewer than 20 threads for these inputs. However, with the larger inputs, *i*FT reduced the runtime and memory overheads compared to the other detectors. For example, *i*FT required just 82% of the runtime and 8% of the memory



(a) Average runtime overheads

(b) Average memory overheads

Figure 7: Measured runtime and memory overhead results for x264 application.

overhead of FastTrack for these larger inputs. The results emphasize again that *i*FT is practically useful for detecting data races on-the-fly in programs with a large number of concurrent threads.

The empirical results from Table 4 to Figure 7 show that our *i*FT algorithm is a sound and practical method for on-the-fly data race detection, because it reduces the average runtime and memory overhead to 84% and 37%, respectively, of those recorded by FastTrack.

6. Related Work

Most prior dynamic techniques have focused on detecting data races more precisely or efficiently. Since FastTrack was introduced, several detectors have been designed to combine lockset analysis with happens-before analysis by leveraging the lightweight nature of epochs.

AccuLock [20] was the first solution to use this combined approach, achieving comparable performance to FastTrack and limited false positives. This detector applies a new, efficient lockset algorithm to FastTrack to enforce a thread locking discipline. This uses the notion of potential data races, called ∅-races, in which any two concurrent read/write events access a shared memory location without a common lock. The detector considers the sensitivity to thread interleaving using thread locking, as it excludes the subset of happens-before relations found with lock acquirements and releases from VCs. However, AccuLock still requires $O(n)$ operations to maintain an access history and locate data races, similar to FastTrack.

ThreadSanitizer [8] is another hybrid detector based on the same combination approach. This detector provides improved precision in the detection of data races by adapting the fastidious aspect of thread synchronizations and race patterns appearing in C/C++ applications. However, unlike AccuLock, it uses VCs to analyze the happens-before relation and multiple locksets for concurrent writes. Thus, the detector offers the same time and memory overhead as earlier hybrid detectors such as MultiRace [3]. Recently, a new version of ThreadSanitizer was released (but not reported officially). This included the FastTrack algorithm and epochs instead of the VCs of the old version.

In our prior work [33], we presented an on-the-fly Race Detector for OpenMP programs. This detector uses a thread identifying technique to analyze the happens-before relation and a data race detection protocol that utilizes the lockset analysis. A significant improvement in efficiency was obtained because the left-of-relation was also applied to the protocol, and it is able to precisely report data races for OpenMP programs with a large number of concurrent threads. However, our prior detector may lose its soundness or efficiency when handling general threading models, like Pthread, because it only considers the structured fork-join parallel program model, such as OpenMP.

7. Conclusion

There is a trade-off between efficiency and precision in the detection of data races using the happens-before or lockset analysis. FastTrack is the fastest happens-before analysis algorithm to provide comparable performance to the lockset analysis. However, there is still room for improvement, as the algorithm requires some VC operations. In this paper, we presented an improved FastTrack algorithm, called *i*FT, that uses only the epochs in each access history by applying the left-of-relation. This algorithm is practically sound, needing

only an $O(1)$ runtime and memory overhead to maintain an access history and providing similar performance to the well-established FastTrack algorithm.

We implemented our algorithm as a Pin-tool on top of the Pin instrumentation framework and compared it empirically with other detection algorithms, including FastTrack. Empirical results from a set of C/C++ benchmarks showed that our *i*FT algorithm is a practical and sound method for on-the-fly data race detection, reducing the average runtime and memory overhead to 84% and 37%, respectively, of those required by FastTrack. This low overhead of the *i*FT algorithm is significant, because it can be used for on-the-fly detection based on both happens-before analysis and a hybrid technique, as presented here for an empirical comparison of efficiency. Thus, we believe that the light weight of *i*FT algorithm can apply to production algorithms which include fault tolerance techniques and testing tools for developing dependable software as well as safety critical software such as avionics and nuclear power systems. Future work will focus on improving the *i*FT algorithm via a hybrid detection technique, similar to that of AccuLock but without the false positive problem, and the enhancement of precision to handle more variant synchronization primitives, as in ThreadSanitizer.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2060082).

References

[1] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, "A theory of data race detection," in *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD '06)*, pp. 69–78, ACM, New York, NY, USA, 2006.

[2] R. H. B. Netzer 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.

[3] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded c++ programs," in *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*, pp. 179–190, ACM, June 2003.

[4] X. Zhou, K. Lu, X. Wang, and X. Li, "Exploiting parallelism in deterministic shared memory multiprocessing," *Journal of Parallel and Distributed Computing*, vol. 72, no. 5, pp. 716–727, 2012.

[5] M. Ronsse and K. De Bosschere, "RecPlay: a fully integrated practical record/replay system," *ACM Transactions on Computer Systems*, vol. 17, no. 2, pp. 133–152, 1999.

[6] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pp. 121–133, ACM, June 2009.

[7] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas, "Sigrace: signaturebased data race detection," *SIGARCH's Computer Architecture News*, vol. 37, no. 3, pp. 337–348, 2009.

[8] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer—data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*, pp. 62–71, ACM, New York, NY, USA, December 2009.

[9] M. Christiaens, M. Ronsse, and K. De Bosschere, "Bounding the number of segment histories during data race detection," *Parallel Computing*, vol. 28, no. 9, pp. 1221–1238, 2002.

[10] A. Dinning and E. Schonberg, "Detecting access anomalies in programs with critical sections," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD '91)*, pp. 85–96, ACM, New York, NY, USA, May 1991.

[11] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Proceedings of the ACM/IEEE conference on Supercomputing (Supercomputing '91)*, pp. 24–33, ACM, New York, NY, USA, November 1991.

[12] D. Perkovic and P. J. Keleher, "A protocol-centric approach to on-the-fly race detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 10, pp. 1058–1072, 2000.

[13] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Efficient data race detection for async-finish parallelism," *Formal Methods in System Design*, vol. 41, no. 3, pp. 321–347, 2012.

[14] S. Savage, 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.

[15] J. J. Harrow, "Runtime checking of multithreaded applications with visual threads," in *SPIN Model Checking and Software Verification: 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000. Proceedings*, vol. 1885 of *Lecture Notes in Computer Science*, pp. 331–342, Springer, Berlin, Germany, 2000.

[16] A. Jannesari, B. Kaibin, V. Pankratius, and W. F. Tichy, "Helgrind+: an efficient dynamic race detector," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09)*, pp. 1–13, IEEE Computer Society, Rome, Italy, May 2009.

[17] A. Jannesari and W. F. Tichy, "On-the-fly race detection in multi-threaded programs," in *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD '08)*, ACM, New York, NY, USA, July 2007.

[18] A. Jannesari and W. F. Tichy, "Identifying ad-hoc synchronization for enhanced race detection," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS '10)*, pp. 1–10, April 2010.

[19] E. Pozniansky and A. Schuster, "MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs: research articles," *Concurrency and Computation: Practice & Experience*, vol. 19, no. 3, pp. 327–340, 2007.

[20] X. Xie and J. Xue, "Acculock: accurate and efficient detection of data races," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*, pp. 201–212, IEEE Computer Society, Chamonix, France, April 2011.

[21] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: efficient detection of data race conditions via adaptive tracking," in

Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05), pp. 221–234, ACM, October 2005.

[22] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*, pp. 167–178, ACM, New York, NY, USA, June 2003.

[23] M. D. Bond, 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] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: effective sampling for lightweight data-race detection," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 134–143, 2009.

[25] K. Zhai, B. Xu, W. K. Chan, and T. H. Tse, "CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications," in *Proceedings of the 21st International Symposium on Software Testing and Analysis (ISSTA '12)*, pp. 221–231, ACM, New York, NY, USA, July 2012.

[26] R. Baldoni and M. Raynal, "Fundamentals of distributed computing: a practical tour of vector clock systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, 2002.

[27] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[28] M. Bach, M. Charney, R. Cohn et al., "Analyzing parallel programs with pin," *Computer*, vol. 43, no. 3, Article ID 5427374, pp. 34–41, 2010.

[29] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson, "On-the-fly maintenance of series-parallel relationships in Fork-Join multithreaded programs," in *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04)*, pp. 133–144, ACM, New York, NY, USA, June 2004.

[30] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," in *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*, pp. 65–74, ACM, June 2007.

[31] C. Bienia and K. Li, "Parsec 2.0: a new benchmark suite for chipmultiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[32] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. Amarasinghe, "Aikido: accelerating shared data dynamic analyses," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 173–184, 2012.

[33] O.-K. Ha, I.-B. Kuh, G. M. Tchamgoue, and Y.-K. Jun, "On-the-fly detection of data races in openmp programs," in *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD '12)*, pp. 1–10, ACM, New York, NY, USA, 2012.