LOFT: Redundant Synchronization Event Removal for Data Race Detection†

Yan Cai  
Department of Computer Science  
City University of Hong Kong  
Tat Chee Avenue, Hong Kong  
yancai2@student.cityu.edu.hk

W.K. Chan  
Department of Computer Science  
City University of Hong Kong  
Tat Chee Avenue, Hong Kong  
wkchan@cs.cityu.edu.hk

Abstract—Many happens-before based techniques for multithreaded programs implement vector clocks to track incrementally the causal relations among the synchronization operations acting on threads and locks. In these detectors, every such operation results in a vector-based assignment to a vector clock, even though the assigned value is the same as the value of the vector clock right before the assignment. The cost of such vector-based operations however grows with the number of threads and the amount of such operations. It is unclear to what extent redundant assignments can be removed. Whether two consecutive assignments to the same vector clock of a thread result in the same content critically depends on the operations on the locks occurred in between these assignments. In this paper, we systematically explore the said insight and quantify a sufficient condition that can soundly remove such operations without affecting the precision of such tracking. We applied our approach on FastTrack to formulate LOFT. We evaluate LOFT using the PARSEC benchmarking suite. The result shows that, on average, LOFT removes 58.0% of all such operations incurred by FastTrack, and runs 16.2% faster than the latter in tracking the causal relations among these operations.

Keywords—data race detection; redundant operation optimization

I. INTRODUCTION

The advent of multi-core processors motivates programmers to develop multithreaded programs to improve the efficiency of their programs through parallel computations. However, any improper synchronization among threads in a multithreaded program may lead the program to produce a failure, such as a wrong output or crash. Unfortunately, debugging a multithreaded program can be intricate because a concurrency bug may merely manifest itself into a failure in some but not all interleaving sequences of threads even for the same input.

A data race is said to occur if two or more threads accessing the same memory location in an undetermined order, and at least one of these accesses is a write operation [9]. If such a data race can lead the program to produce unexpected behavior, the race is said to be harmful. Detecting (harmful) data races in programs is one of the preventive measures to assure the reliability of multithreaded programs.

In particular, many dynamic detectors have been proposed. Examples include Goldilock [7], FastTrack [9], LiteRace [16], DJIT+ [19], Helgrind™[11], AccuLock [23], RaceTrack [24], and Eraser [22]. These detectors can be sub-classified into lockset based algorithms (e.g., [7][22]), happens-before based algorithms (e.g., [9][16][19]), and the hybrid of the former two (e.g., [11][23][24]). Lockset based algorithms in general run faster than, but are much less precise than the other two kinds of dynamic detectors. FastTrack, being a kind of happens-before based algorithm, recently demonstrated that this class of algorithm can be efficiently implemented. For instance, the mean efficiency of FastTrack can be comparable with that of lockset based algorithms (e.g., Eraser) in an empirical experiment using a suite of Java programs as subjects [9].

We observe that almost all happens-before based dynamic detectors commonly implement vector clocks [14] to track the casual relationships among memory accesses, lock operations, and thread manipulation. In these detectors, every operation that tracks such relationships among lock operations and thread management must produce a vector join operation over two vector clocks, and the result of the join may further be assigned to at least one vector clock. For instance, although Pacer [3] applies a sampling approach to collect memory access operations to reduce the overhead for data race detection, yet it still needs to track the casual relationships among threads and locks in full, irrespective to whether or not these events occur in sampling periods.

In this paper, we study the problem of precise reduction of vector clock updates for threads and locks in the on-the-fly tracking of the happens-before relations on an execution trace. We apply our result to dynamic happens-before based data race detectors. It is worth noting that our solution is general rather than restrictive to such detectors.

The size of each vector clock in such a detector grows as the number of threads in a program increases. Hence, the time costs of the said join and assignment operations also change (linearly) with the number of threads in a program.

Is it always necessary for a dynamic happens-before based detector to assign a new instance to a vector clock whenever an operation of the above kind is observed? If the answer is negative, how can a technique soundly identify those redundant operations? Moreover, to what extent can such a technique remove the involved redundant operations? There are wide applications of this kind of technique. For instance, if such a technique can remove a large amount of such redundant operations, the size of a corresponding operation log for post-mortem analyses or execution replay techniques can be reduced significantly. To the best of our knowledge, the above research questions have not been explored.

† This work is supported in part by the General Research Fund of the Research Grant Council of Hong Kong (project no. 111410).
We observe that whether two consecutive assignments to the same vector clock of a thread result in the same content critically depends on the operations for those lock occurred in between the two assignments. Let us consider the following example: Suppose that a thread $t$ releases a lock $m$ followed by acquiring it. Further suppose that in between this pair of operations, no thread acquires $m$ or releases it. In this situation, a detector needs not to assign any value to $m$’s vector clock to reflect the lock acquisition operation. It is because the original value kept by $m$’s vector clock is still sufficient to reflect the latest causal relationship between $t$ and $m$, and the visible timestamps of other threads from the viewpoint of $t$. To ease our presentation, we refer to such a “suppressed” vector-based operation (such as a comparison or an assignment) as a redundant operation.

In this paper, we explore the above insight. We systematically and exhaustively analyze and characterize the above kinds of scenarios, and formulate the conditions that can soundly remove such redundant operations. We apply our approach to formulate an algorithm called LOFT, standing for Lock-Optimized FastTrack. We base our approach on FastTrack [9] because FastTrack represents the state of the art for dynamic happens-before based algorithms.

In the experiment, we evaluated, via LOFT, to what extent our approach can eliminate vector clock updates for synchronization operations acting on threads and locks without compromising the precision of data race detection of FastTrack on the PARSEC benchmarking suite [2][6][11]. The experimental result showed that on average, LOFT removed 58.0% of all such operations needed by FastTrack without any loss in detection precision, and ran 16.2% faster than FastTrack in tracking all causal relationships among the synchronization operations acting on threads and locks in the executions of these subjects.

The main contribution of this paper is threefold: (1) We identify and characterize a class of thread-centric scenarios that each involves a consecutive pair of lock operations. We formulate the first sufficient condition that redundant operations can be soundly removed without affecting the precision of the causal relations being tracked on the fly. Our solution on the elimination of the redundant vector clock updates is general and not restrictive to a particular correctness criterion (e.g., data race freedom) used in pair with our solution. (2) It proposes a data race detector (LOFT), which implements our approach. (3) It reports an experiment that validates the feasibility of our approach, and compares LOFT with FastTrack. The experimental result show a significant amount of vector clock updates induced by synchronization operations can be removed. The result also shows that the time cost to maintain the data structure for the above sufficient condition and the checking itself are well-compensated by the reduced amount of vector clock updates.

The rest of the paper is organized as follows. Section II presents a motivating example. Section III elaborates the preliminaries of happens-before based data race detection. Section IV presents our analysis and LOFT. Section V reports an experiment that validates our approach. Section VI reviews related work. Section VII concludes the paper.

II. MOTIVATING EXAMPLE

Figure 1(a) shows a motivating example adapted from the classic Producer and Consumer Problem. It shows a shared location pool, which is protected by a shared lock $m$, and two threads (Producer and Consumer). The Producer thread repetitively produces a datum, and puts it into pool. The Consumer thread repetitively fetches a datum from pool.

Figure 1(b), from top to bottom, shows a possible execution that interleaves between the two threads, as indicated by the rightmost and the leftmost columns of Figure 1(b). The Producer thread firstly acquires and releases the lock $m$ twice. Then, the Consumer thread also acquires and releases the lock twice. Finally, the Producer thread acquires and releases the lock. In total, the execution involves five lock acquire and five lock release.

Let us use FastTrack [9] on the above execution to illustrate our point. The algorithm firstly sets up three vector clocks for $m$, Producer, and Consumer, respectively, as shown in Figure 1(b) under the vector clock column. To track each lock acquire or release operation on the fly, the algorithm needs to perform two vector-based operations, one for comparing two vector clock instances and another for updating the vector clock of $m$, Producer, or Consumer to

---

**Shared variables**

<table>
<thead>
<tr>
<th>int pool[1000];</th>
<th>Lock $m$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool isPoolEmpty;</td>
<td>bool isPoolFull;</td>
</tr>
</tbody>
</table>

**Consumer**

```c
while(true) {
    while(isPoolEmpty) wait(100);
    Acquire($m$);
    //fetch a datum from pool
    ...
    Release($m$);
}
```

**Producer**

```c
while(true) {
    while(isPoolFull) wait(100);
    Acquire($m$);
    //add a datum to pool
    ...
    Release($m$);
}
```

(a) The code

**Possible interleaving**

(for brevity, we only show acquire and release operations)

<table>
<thead>
<tr>
<th>Consumer</th>
<th>vector clock</th>
<th>Producer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer</td>
<td>$&lt;1,1&gt;$</td>
<td>$&lt;0,0&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;0,0&gt;$</td>
<td>$&lt;1,1&gt;$</td>
</tr>
<tr>
<td>m</td>
<td>$&lt;0,0&gt;$</td>
<td>$&lt;1,1&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;0,1&gt;$</td>
<td>$&lt;1,1&gt;$</td>
</tr>
<tr>
<td>Producer</td>
<td>$&lt;1,1&gt;$</td>
<td>$&lt;1,2&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;1,1&gt;$</td>
<td>$&lt;1,2&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;1,2&gt;$</td>
<td>$&lt;1,3&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;1,2&gt;$</td>
<td>$&lt;1,3&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;2,2&gt;$</td>
<td>$&lt;1,2&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;2,2&gt;$</td>
<td>$&lt;1,3&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;3,2&gt;$</td>
<td>$&lt;1,2&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;3,2&gt;$</td>
<td>$&lt;1,3&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;2,3&gt;$</td>
<td>$&lt;2,4&gt;$</td>
</tr>
<tr>
<td></td>
<td>$&lt;2,3&gt;$</td>
<td>$&lt;2,4&gt;$</td>
</tr>
</tbody>
</table>

(b) Analysis on vector clock instances on a possible execution

Figure 1. An example consumer and producer example
keep another instance. Therefore, FastTrack needs in total 10 such (vector clock) operations. Every such operation takes \(O(n)\) time, where \(n\) is the size of a vector clock, which is also the number of threads in the example (i.e., 2). The values of the three vector clocks are also shown in Figure 1(b).

We observe that many such lock acquire (release, respectively) operations marked with the star “*” symbols (“-”), respectively in Figure 1(b) need either no vector-based operation at all or merely an assignment of a value to one entry of one vector clock. In the figure, they are shown as shaded vector clocks and shaded entries, respectively. The underlying reason is as follows: the lock is consecutively acquired or released by the same thread. Because the lock is only used by the same thread (say Producer), updating the vector clock of the thread to collect the timestamp of another thread is unnecessary.

Owing to the above reason, the involved vector-based operations for these “*” and “-” operations can be either safely removed or replaced by an assignment with a scalar value, which only takes \(O(1)\) time. Consequently, in an ideal case, only the operations marked with the plus “+” need to take vector-based operations. Hence, to track the causal relationships as illustrated in Figure 1(b), a good algorithm can use three vector clock operations to complete the tracking of all these vector clock instances. In summary, seven operations are redundant, which can be substituted by scalar operations, such as updating the value in the initialized vector clock instance of the lock \(m\) from “1” to “2” on the second release of the Producer thread. Our approach explores this insight.

III. PRELIMINARIES

A. Events

A data race detector typically monitors a set of critical operations, such as read (rd) from or write (wr) to a memory location \(v\); acquire (acq) or release (rel) a lock \(m\); fork or join a thread \(t\). Like many existing work, our model does not consider nested locks because the handling of such locks or reentrance locks can be extended. Following [9], for brevity, we only present how our model handles the above set of six critical operations. We assume that we can obtain the standard execution information such as the identity of each thread and the related program statement associated with each operation.

A trace \(\sigma\) is the projection of an execution of a program on this set of critical operations. We assume that the program being monitored is sequentially consistent [17]. Moreover, we assume that a lock can only be acquired by at most one thread at a time.

B. Happens-before Relations and Data Race

A happens-before relation, denoted by \(\xrightarrow{h_b}\), is a partial order relation among events in a multi-threaded program or concurrent system [14]. It is defined by the following three rules: (a) Program order: If \(\alpha\) and \(\beta\) are two events (i.e., two critical operations described above) performed by the same thread, and \(\alpha\) precedes \(\beta\), then we write \(\alpha \xrightarrow{h_b} \beta\). (b) Release and acquire: If \(\alpha\) is a release operation of a lock \(m\), and \(\beta\) is an acquire operation of the same lock \(m\) performed by a thread different from the one performing \(\alpha\), and \(\alpha\) precedes \(\beta\), then we write \(\alpha \xrightarrow{h_b} \beta\). (c) Transitivity: if \(\alpha \xrightarrow{h_b} \beta\) and \(\beta \xrightarrow{h_b} \gamma\), then \(\alpha \xrightarrow{h_b} \gamma\).

A data race is formally defined as follows: Suppose that two events \(\alpha\) and \(\beta\) accessing the same shared location \(v\) in a trace, and at least one of them is a write. If neither \(\alpha \xrightarrow{h_b} \beta\) nor \(\beta \xrightarrow{h_b} \alpha\), then \((\alpha, \beta)\) forms a racing pair. We consider that both \((\alpha, \beta)\) and \((\beta, \alpha)\) refer to the same racing pair. Similarly, we consider that \((\text{statement}(\alpha), \text{statement}(\beta))\) is also the same racing pair as \((\alpha, \beta)\), where \(\text{statement}(x)\) is the program statement that is associated with the event \(x\). The shared location \(v\) is said to be in race.

An algorithm for dynamic data race detection outputs a set of event pairs or a set of locations based on a set of traces. An event pair is said to be a false positive if the reported pair is not a racing pair. Similarly, a location \(v\) is said to be a false positive if \(v\) is not in race on any such trace, and yet the algorithm does not include the location in its output. An algorithm is said to be precise if any reported racing pair or location is not a false positive.

C. Vector Clock and DJIT+

We use DJIT+ [19] to illustrate data race detections.

A timestamp is a number. A vector clock is a finite array of timestamps. DJIT+ assigns one vector clock \(C_i\) to each thread \(t\). This vector clock logs the thread’s current timestamp as well as the other threads’ timestamps visible to the thread \(t\). DJIT+ also assigns one vector clock \(L_m\) to each lock \(m\). For each memory location \(v\), it assigns two vector clocks \(W_v\) and

```
Algorithm: DJIT+

On initialization:
1. For each thread \(t\), \(C_i[t]=1\), where \(i\) is from 1 to \(n\).
2. For each memory location \(v\), \(W_v[i]=R_v[i]=0\), where \(i\) is from 1 to \(n\).
3. For each lock \(m\), \(L_m[i]=0\), where \(i\) is from 1 to \(n\).

On acquiring a lock \(m\) for thread \(t\):
4. \(C_i(t) = \max\{C_i[t], L_m[i]\}\), where \(i\) is from 1 to \(n\).

On releasing a lock \(m\) for thread \(t\):
5. \(C_i(t) = C_i[t] + 1\).
6. \(L_m[i] = \max\{C_i[t], L_m[i]\}\), where \(i\) is from 1 to \(n\).

On the first read to a memory location \(v\) in the current timestamp for thread \(t\):
7. \(R_v[i] = C_i[t]\).
8. For each thread \(i\) (where \(#i\)≠), if \(C_i[t] \leq W_v[i]\), report a write-read data race, where \(i\) is from 1 to \(n\).

On the first write to a memory location \(v\) in the current timestamp for thread \(t\):
9. \(W_v[i] = C_i[t]\).
10. For each thread \(i\) (where \(#i\)≠), if \(C_i[t] \leq W_v[i]\), report a write-write data race, where \(i\) is from 1 to \(n\).
11. For each thread \(i\) (where \(#i\)≠), if \(C_i[t] \leq R_v[i]\), report a read-write data race, where \(i\) is from 1 to \(n\).
```

Figure 2. The DJIT+ algorithm
Each thread $t$ has its own timestamp variable that is incremented on each release operation performed by $t$. $C_t$ records the current timestamps of the thread $t$ and others threads gotten from $L_m$ on acquiring the lock $m$ by $t$. $L_m$ records a snapshot of $C_t$ when $t$ releases the lock $m$.

To maintain its data structure, DJIT+ uses the following strategies. For every acquire operation on the lock $m$ performed by the thread $t$, DJIT+ updates $C_t$ to be a vector instance, in which each entry is the maximum of the corresponding entries in $L_m$ and $C_t$ (i.e., $C_t = C_t \cup L_m$, see [8][9]). For every release operation of a lock $m$, DJIT+ increments the timestamp kept at $C_t[m]$ by one (while all the other values kept by $C_t$ remain unchanged), followed by updating $L_m$ to be a vector instance, in which each entry is the maximum of the corresponding entries in $L_m$ and $C_t$ (i.e., $L_m = C_t \cup L_m$).

Moreover, for every write (read, respectively) operation to a memory location $v$ performed by a thread $t$, DJIT+ updates $W_v[t]$ ($R_v[t]$, respectively) to be the contents of $t$'s vector clock (i.e., $W_v[t] := C_t[t]$). Immediately after each read operation (by a thread $i$) from the location $v$, DJIT+ compares $C_t$ with $W_v[i]$ to determine whether any thread, say $i$ (where $i \neq t$), recorded in these two vector clock instances violates the following condition: $C_t[i] \leq W_v[i]$. If this is the case, a write-read data race is said to have been detected. Similarly, immediately after each write operation to $v$, in addition to the above comparison for the purpose of detecting write-write data races, DJIT+ further compares $C_t$ with $R_v$ to determine whether any thread, say $i$ (where $i \neq t$), recorded in these two vector clock instances violates the following condition: $C_t[i] \leq R_v[i]$. If this is the case, a read-write data race is said to have been detected.

Figure 2 shows the DJIT+ algorithm (where $n$ is the number of threads). DJIT+ slows down a program execution significantly, especially if the execution involves many concurrently running threads. This is because DJIT+ needs one vector-to-vector comparison for every read or write operation on every memory location and every lock acquire or release operation, which is $O(n)$ in time complexity each.

IV. OUR ANALYSIS AND LOFT

Figure 3 depicts an overview of our technique LOFT. The component with a solid frame differentiates LOFT from other data race detectors, in which LOFT implemented our analysis result to remove some possible operations on vector clock instances for threads manipulation and lock events.

FastTrack reduces the amount of vector creations and usages incurred by DJIT+ to record the time that an execution accesses memory locations (i.e., steps 7–11 in Figure 2). To track their happens-before relations, these algorithms (including FastTrack) commonly update the vector clock instances for threads and locks by assigning them with other vector clock instances whenever an event for thread

$$R_v$$ for the write and read operations on $v$, respectively.

management or lock operations is observed. As we have illustrated in the motivating example, every such event results in at least one vector-based comparison or assignment (i.e., the steps 4–6 in Figure 2) [9][16][19].

In this section, we analyze the scenarios for the steps 4–6 in Figure 2. As we have described in Section I, our approach is generic. Specifically, we characterize lock acquire and release operations by exhausting all possible scenarios in between a pair of such consecutive operations performed by the same thread. We present them as six cases as depicted by Figure 4.

We firstly present some auxiliary functions to ease our subsequent presentation. Suppose that $V_1$ and $V_2$ are two vector clock instances, and the number of elements in either instance is $n$. If $V_1[i] \leq V_2[i]$ for $1 \leq i \leq n$, we denote this condition by $V_1 \leq V_2$. Similarly, if $V_1[i] = V_2[i]$ for $1 \leq i \leq n$, we denote this condition by $V_1 = V_2$. We also define $V_1 \cup V_2$ to be a vector clock instance $V_3$ such that $V_3[i] = \max(V_1[i], V_2[i])$ for $1 \leq i \leq n$, and the number of elements in the instance is also $n$. We use $e_i$ (for $i=1, 2, \ldots$) to denote critical operations. We also define two functions: lastLock($t$) represents the most recent lock that the thread $t$ has released, and lastThread($t$) represents the most recent thread that has released the lock $m$. For instance, in the motivating example, when the first occurrence of the acquire($m$) event in the Consumer column of Figure 1(b) occurs, the Consumer thread did not acquire (hence did not release) any other lock. So, for this event, lastLock (Consumer) is null. At this moment, $m$ has been most recently released by the Producer thread via the second occurrence of the release($m$) event of Producer. So, with respect to the above acquire($m$) event, lastThread($m$) is Producer.

We are going to present six cases. In each case, the condition refers to the condition when $e_x$ in the case occurs, which is also the highlighted event for the corresponding case in Figure 4.

When $e_x$ is Acquire ($t, m$)

Case 1. [when lastThread($m$) = $t$].

Let $e_t$ be an event in a trace that $t$ releases $m$ such that lastThread($m$) = $t$, and $e_x$ be an event in the same trace that $t$ acquires $m$.

Consider the trace ($\ldots, e_t, \ldots, e_x, \ldots$). When $e_t$ occurs, we must have $L_m = C_t$ (as shown as the first arrow in
Case 1 of Figure 4). Moreover, when \( e \) occurs, \( L_m \) must still remain unchanged. However, the values in \( C_t \) may or may not be incremented because \( t \) may acquire some other lock(s) in between \( e_1 \) and \( e_3 \); otherwise, \( C_t \) must remain unchanged. In either situation, we have \( L_m \subseteq C_t \) when \( e \) occurs. So, for the tracking of \( e \), assigning the values from \( C_t \cup L_m \) to \( C_t \) does not change the values kept in \( C_t \). Therefore, there is no need to perform any comparison between \( L_m \) and \( C_t \). Hence, the above assignment can be removed (which is shown as a dotted arrow in Case 1 of Figure 4) when \( e \) occurs.

Case 2. \([\text{when } \text{lastThread}(m) \neq t]\).

Let \( e \) be an event that \( t \) acquires \( m \).

Consider the trace \( \langle \ldots, e, \ldots \rangle \). When \( e \) occurs, because we have \( \text{lastThread}(m) \neq t \), there are two sub-cases to consider: \( m \) must either have been released by a thread \( t' \) (where \( t' \neq t \)) or have not been updated since it was initialized. In the former case, \( L_m \) must once contain a value the same as that of \( C_t \) because all locks are initialized as all 0s, whereas all threads are initialized as all 1s (see steps 1 and 3 of Figure 2). Therefore, without further checking, we cannot decide whether \( L_m \subseteq C_t \) holds when \( e \) occurs. In this situation, when \( e \) occurs, such a comparison and its associated potential assignment from \( C_t \cup L_m \) to \( C_t \) are necessary, and cannot be removed (which is depicted by the second arrow in Case 2 of Figure 4).

When \( e \) is Release \((t, m)\)

Case 3. \([\text{when } \text{lastThread}(m) = t \text{ and } \text{lastLock}(t) = m]\).

Let \( e \) be an event in a trace that \( t \) releases \( m \) such that \( \text{lastThread}(m) = t \) and \( \text{lastLock}(t) = m \), \( e \) be an event that \( t \) releases \( m \), and \( e_2 \) be the corresponding acquire operation by \( t \) with respect to \( e \).

Consider the trace \( \langle \ldots, e_1, \ldots, e_2, \ldots, \rangle \), which is depicted as Case 3 in Figure 4. The analysis for Case 3 is straightforward: When \( e_1 \) occurs, we have \( L_m = C_t \). In between \( e_1 \) and \( e_2 \), as well in between \( e_2 \) and \( e_3 \), the condition \( \text{lastLock}(t) = m \) implies that \( t \) has not acquired or released any other lock after \( e_1 \). Moreover, the condition \( \text{lastThread}(m) = t \) implies that \( m \) has not been acquired or released by any other thread after \( e_1 \). These two conditions respectively imply that \( C_t \) and \( L_m \) remain unchanged when \( e \) occurs. Therefore, we have \( L_m = C_t \) when \( e_2 \) occurs. Consequently, we have \( C_t \cup L_m \) carrying the same value as that kept by \( L_m \). Hence, there is no need to update \( L_m \). When \( e \) occurs, the involved vector-based comparison between \( C_t \) and \( L_m \) and the assignment to \( L_m \) can be removed (which is depicted by the third (dotted) arrow in Case 3 of Figure 4).

Case 4. \([\text{when } \text{lastThread}(m) = t \text{ and } \text{lastLock}(t) \neq m]\).

Let \( e \) be an event that \( t \) releases \( m \) such that \( \text{lastThread}(m) = t \) and \( \text{lastLock}(t) \neq m \).

Consider the trace \( \langle \ldots, e_1, \ldots, e_2, \ldots, \rangle \). When \( e \) occurs, the condition \( \text{lastThread}(m) = t \) implies that \( m \) has not been acquired and released by any other thread in between \( e_1 \) and \( e_2 \). Hence, \( L_m \) remains unchanged since the occurrence of \( e_1 \). (Note also because of this condition, \( L_m \) must have been updated at least once, and hence, \( L_m \) cannot stay at its initialized value.) However, when \( e \) occurs, the condition \( \text{lastLock}(t) \neq m \) implies that \( C_t \) has been updated due to the thread’s acquisition of some other lock(s) in between \( e_1 \) and \( e_2 \) (as illustrated by the \( \text{acq}(l) \) operation in the example). This operation may have incremented some timestamps kept by \( C_t \) with respect to the same vector clock at the time when \( e \) occurs. Therefore, the assignment from \( C_t \cup L_m \) to \( L_m \) cannot be removed when \( e \) occurs.

Case 5. \([\text{when } \text{lastThread}(m) \neq t \text{ and } \text{lastLock}(t) = m]\).

Let \( e \) be an event that \( t \) releases \( m \) such that \( \text{lastLock}(t) = m \), \( e \) be an event that \( t \) releases \( m \), and \( e_2 \) be the corresponding lock acquire operation of \( e \). Similar to Case 4, these three events are depicted as the first and the last two operations in Case 5 of Figure 4.

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
<th>Case 5</th>
<th>Case 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>( m )</td>
<td>( t )</td>
<td>( m )</td>
<td>( s )</td>
<td>( t )</td>
</tr>
<tr>
<td>\text{rel}(m)</td>
<td>\text{acq}(m)</td>
<td>\text{acq}(m)</td>
<td>\text{rel}(m)</td>
<td>\text{acq}(m)</td>
<td>\text{rel}(m)</td>
</tr>
</tbody>
</table>

Figure 4. Example scenarios of the six cases on acquiring or releasing a lock (where \( t \) and \( s \) are threads; and \( m \) and \( l \) are locks; \( \text{rel}(m) \) and \( \text{acq}(m) \) represent release(m) and acquire(m), respectively. “***” is the direction for vector clock assignment between the corresponding thread and lock: (1) an arrow with a dot at the end of the arrow shows an \( \text{acq}(m) \) or a \( \text{rel}(m) \) operation that satisfies the LRDB or LRG relation, (2) a dotted arrow shows a corresponding \( \text{acq}(m) \) or \( \text{rel}(m) \) operation that violates the LRDB or LRG relation, (3) a gray arrow is just for the reference. “****” means that there may be additional pairs of \( \text{acq}(x) \) and \( \text{rel}(x) \) where \( x \) is a lock).
Consider the trace \( \langle e_1, e_2, \ldots, e_n \rangle \). When \( e_n \) occurs, the condition \( \text{lastLock}(t) = m \) implies that \( t \) has not acquired or released any lock other than \( m \) since \( e_n \) occurred. Therefore, in between \( e_1 \) and \( e_2 \), \( C_t \) remains unchanged. The condition \( \text{lastThread}(m) \neq t \) further implies that, in between \( e_1 \) and \( e_2 \), \( m \) must have been acquired by other threads (as illustrated by the \( \text{acq}(m) \) operation by the thread \( s \) in the example). Hence, \( L_m \) might have been updated (e.g., by the \( \text{rel}(m) \) operation of \( s \) in the example). So, we must have \( C_t \subseteq L_m \) in this period.

The condition \( \text{lastThread}(m) \neq t \) also implies that, when \( e_2 \) occurs, \( C_t \) must be updated to be \( C_t \cup L_m \). Hence, we have \( L_m = C_t \) when \( e_2 \) occurs.

In between \( e_2 \) and \( e_n \), \( m \) cannot be acquired or released by any other thread because \( m \) is being held by \( t \). Moreover, during this period, \( t \) cannot release a second lock (otherwise, the condition \( \text{lastLock}(t) = m \) cannot hold). Therefore, when \( e_n \) occurs, the condition \( L_m = C_t \) still holds. Similar to Case 3, both the vector-based comparison and the assignment can be removed.

**Case 6.** When \( \text{lastThread}(m) \neq t \) and \( \text{lastLock}(t) \neq m \).

In this case, we cannot infer anything between \( t \) and \( m \). Therefore, no vector-based comparison or assignment can be removed. An example scenario is depicted as Case 6 in Figure 4.

In the rest of the paper, we refer to the condition in Case 1 (i.e., \( \text{lastThread}(m) = t \)) on \( \text{acquire}(t, m) \) as \( \text{LRDB}(t, m) \), which standing for “Last ReleaseD By”. Similarly, the conditions in Case 3 and Case 5 can be combined into one condition: \( \text{lastLock}(t) = m \) on \( \text{Release}(t, m) \), which we refer to it as \( \text{LRG}(t, m) \), standing for “Last Releasing”.

---

**LOFT State:**

- **C**: Tid → (VC, Lock)
- **L**: Lock → (VC, Tid)
- **W**: Var → Epoch
- **R**: Var → (Epoch ∪ VC)

### On Acquire (t, m)

- **LRDB(t, m)**
  
  \[ t = m \cdot \text{Tid} \]
  
  \( (C, L, R, W) \Rightarrow \text{acquire}(t, m) (C', L, R, W) \)

- **Otherwise**
  
  \( C' = C [t := C_t \cup L_m] \)
  
  \( (C, L, R, W) \Rightarrow \text{acquire}(t, m) (C', L, R, W) \)

### On Release (t, m)

- **LRG(t, m)**
  
  \[ m = t \cdot \text{Lock} \]
  
  \[ m \cdot \text{Tid} := t \]
  
  \[ L' = L[m := L_m[t := C_t(t)]] \]
  
  \( C' = C[t := \text{inc}_L(C_t)] \)
  
  \( (C, L, R, W) \Rightarrow \text{release}(t, m) (C', L', R, W) \)

- **Otherwise**
  
  \[ m \cdot \text{Tid} := t \]
  
  \[ t \cdot \text{Lock} := m \]
  
  \( L'[m := C_t] \)
  
  \( C' = C[t := \text{inc}_L(C_t)] \)
  
  \( (C, L, R, W) \Rightarrow \text{release}(t, m) (C', L', R, W) \)

### Other rules of LOFT

#### On Reads(t, x):

- **read same epoch**
  
  \[ R_x = E(t) \]
  
  \( (C, L, R, W) \Rightarrow \text{rd}(t, x) (C, L, R, W) \)

- **read shared**
  
  \[ R_x \in VC \]
  
  \[ W_x \subseteq C_t \]
  
  \[ R' = R[x := R_x[t := C_t(t)]] \]
  
  \( (C, L, R, W) \Rightarrow \text{rd}(t, x) (C, L, R', W) \)

- **read exclusive**
  
  \[ R_x \in \text{Epoch} \]
  
  \[ W_x \subseteq C_t \]
  
  \[ R' = R[x := E(t)] \]
  
  \( (C, L, R, W) \Rightarrow \text{rd}(t, x) (C, L, R', W) \)

#### On Writes(t, x):

- **write same epoch**
  
  \[ W_x = E(t) \]
  
  \( (C, L, R, W) \Rightarrow \text{wr}(t, x) (C, L, R, W) \)

- **write exclusive**
  
  \[ R_x \in \text{Epoch} \]
  
  \[ W_x \subseteq C_t \]
  
  \[ W' = W[x := E(t)] \]
  
  \( (C, L, R, W) \Rightarrow \text{wr}(t, x) (C, L, R, W') \)

- **write shared**
  
  \[ R_x \in VC \]
  
  \[ W_x \subseteq C_t \]
  
  \[ W' = W[x := L_x] \]
  
  \( (C, L, R, W) \Rightarrow \text{wr}(t, x) (C, L, R', W') \)

#### On Fork(t, u):

\[ C' = C[u := C_u \cup C_t; t := \text{inc}_L(C_t)] \]

\( (C, L, R, W') \Rightarrow \text{fork}(t, u) (C', L, R, W') \)

#### On Join(t, u):

\[ C' = C[u := C_u \cup C_t; u := \text{inc}_L(C_u)] \]

\( (C, L, R, W') \Rightarrow \text{join}(t, u) (C', L, R, W') \)

---

Figure 5. LOFT and its comparison to FastTrack (shading lines show the differences between FastTrack and LOFT)
As a result, we formulate the following strategy: if $\text{LRDB}(t, m)$ holds on $\text{acquire}(t, m)$, the corresponding comparison between $L_m$ and $C_t$ and its associated vector clock assignment from $C_t \cup L_m$ to $C_t$ can be removed. Moreover, if $\text{LRG}(t, m)$ holds on $\text{release}(t, m)$, such a comparison and the associated assignment from $C_t \cup L_m$ to $L_m$ can also be removed. In our empirical experiment to be presented in the paper, this strategy can successfully remove 58.0% such operations.

Figure 5 shows our Lock-Optimized FastTrack (LOFT) algorithm and its comparison with the FastTrack algorithm (i.e., the rules without the shaded parts). Apart from introducing the conditions, LOFT also extends FastTrack by adding one variable to each thread and one variable to each lock as shown in the State section of LOFT in Figure 5.

To ease our presentation, we use the same notations as these used in [9]. Specifically, LOFT maintains an analysis state $(C, L, R, W)$ composing of four parts: (1) $C$ maps each thread $t$ (identified by a unique identity $Tid$) to a vector clock $(VC)$ and a lock $m$ (identified by a unique identity $Lock$), where $m$ is the most recent lock that the thread $t$ has released. (2) $L$ maps each lock $m$ to a vector clock and a thread $t$ where $t$ is the last thread that releases $m$. (3) $R$ maps a memory location to an epoch [9] or a vector clock of this location. (4) $W$ maps a memory location to an epoch. We use $C_t$ to denote the vector clock of the thread $t$, and $L_m$ to denote the vector clock of the lock $m$. We also use $t.Lock$ and $m.Tid$ to denote the lock $m$ mapped from the thread $t$ in $C$ and the thread $t$ mapped from the lock $m$ in $L$, respectively.

Initially, each thread is mapped to an empty lock and a newly initialized vector clock instance with a value of “1” in every entry. Moreover, each lock is mapped to an empty thread and a newly initialized vector clock instance with a value of “0” in every entry. The rest of the initial state is the same as that of FastTrack.

**Operations on Lock Acquisition:** As shown in Figure 5, on acquiring a lock $m$ by a thread $t$, LOFT firstly checks whether $\text{LRDB}(t, m)$ holds (by $t = m.Tid$). If this condition is satisfied, LOFT does nothing. Otherwise, $C' = C[t := C_t \cup L_m]$ is performed as FastTrack does, where the notation $C' = C[t := x]$ means that $C'$ is constructed from $C$ by substituting the entry $C[t]$ by $x$.

**Operations on Lock Release:** On releasing a lock $m$ by a thread $t$, LOFT firstly checks whether $\text{LRG}(t, m)$ holds (by $m = t.Lock$). If this condition is satisfied, $L' = L[m := L_m[t := C_t(t)]]$ is performed; otherwise, $L' = L[m := C_t]$ is performed as FastTrack does. Lastly, LOFT increases the timestamp of the thread $t$ ($C' = C[t := inc_t(C_t)]$, where $inc_t(X)$ means $X = [t := X[t] + 1]$). It also updates the mapping between the lock $m$ and the thread $t$ by performing both $m.t := Tid$ and $t.Lock := m$.

**V. EXPERIMENT**

**A. Implementation and Benchmark**

**Implementation.** We implemented LOFT by adding a 32-bit integer to every lock and every thread to record the last thread that releases the lock concerned and the most recent lock released by the thread concerned, respectively. For a program with $n$ threads and $k$ locks, the worst case space complexity to keep the state for these threads and locks is $O(n^2 + kn)$, which is the same as that of the FastTrack. The introduction of the additional integers in our technique does not affect this worst case space complexity order.

We implemented both LOFT and FastTrack using Pin 2.9 [15], which is a program dynamic instrument analysis tool. To implement such a data race detection tool, we needed to shadow every memory location to a set of data (i.e., write epoch, read epoch, and shared read vector clock). We adopted a two level shadow implementation [10] described in [18]. For each thread, because Pin supplies a thread-local storage (TLS) per thread [15], we used this TLS to store a data set (i.e., a vector clock) for each thread. For each lock, we used an unordered map supplied by the GCC compiler to map the lock to a set of data (i.e., a vector clock). Regarding events monitoring, except the thread-starting event supplied by Pin, we dynamically inserted event calls before or after the interesting operations. Following [9], and to allow a fair comparison, our implemented FastTrack and LOFT also reported at most one race condition for each memory location.

**Benchmarks.** We selected the PARSEC benchmark suite 2.1 [2] to evaluate LOFT, which is a set of multithreaded programs used in previous experiments (e.g., [4][6][11]). The suite includes 13 benchmarks: blackscholes, bodytrack, canneal, dedup, facesim, ferret, fluidanimate, freqmine, raytrace, streamcluster, swaptions, vips, and x264. Among these benchmarks, freqmine does not use the standard Pthreads library, we discarded it because our implementations are built on top of the standard Pthreads library; ferret and fluidanimate crashed when we ran them under the Pin environment. We used all the remaining 10 benchmarks in our experiment and executed them with the simsmall input test.

Our experiment was performed on the Ubuntu 10.04 Linux configured with a 3.16GHz Duo2 processor and 3.25GB physical memory. Each benchmark was run 100 times. TABLE I shows the average number of vector operations performed on synchronization events and time needed to complete all such tracking on each benchmark (see the columns Vector operations and Time, respectively). We set each benchmark to have eight worker threads except vips that were preset to have four (fixed) worker threads in the downloaded suite.

**B. Threats to Validity**

In the experiment, we used the PARSEC benchmark suite to validate LOFT. These benchmarks belong to either desktop applications (blackscholes, bodytrack, facesim,
raytrace, swaptions, vips, and x264) or OS kernels (cannal, dedup, and streamcluster). Further experiment on widely used applications such as Firefox, and Apache Web Server may strengthen the experiment.

Our tool used in this paper was implemented in C++. The time measurement may be affected if other programming languages were used for implementation.

We have carefully studied several C/C++ tools that use the Pin framework, especially those related to thread operations. We have compared our detected data races to those detected by other tools (e.g., [13]) to help assure our tool.

C. Data Analysis

Summary of Results. TABLE I summarizes the result of the experiment. The second and the third columns counting from the left report the application domain [2] and the lines of code for each benchmark, respectively. The fourth column shows the number of threads used in the experiment. The column “Vector operations” shows the number of vector clock operations performed for FastTrack (FT) and LOFT, as well as the ratio of LOFT to FastTrack in the column “(B) + (A)”. The column “Time” shows the corresponding time needed to complete all such tracking in microsecond (µs) for FastTrack and LOFT, as well as the ratio of LOFT to FastTrack in the column “(D) + (C)”. We note that the reported time for LOFT has included the time overhead to maintain the LRDB and LRG conditions. The last column is for reference, which shows the number of detected data races on each benchmark because our main focus is on the removal of redundant operations related to threads and locks.

Precision. We find that FastTrack and LOFT reported the same number of data races in each run on each benchmark, except on x264. On x264, 77 data races were reported during most of runs, and we took an average on 100 runs. The mean results are shown in the rightmost column of TABLE I. From the number of detected data races, we find that LOFT does not compromise the precision of FastTrack.

Vector Operations Analysis. From TABLE I, we observe that LOFT, on average, can remove 58.0% of all the vector clock operations that are needed in FastTrack for lock acquisition or release. If we consider the total amount of operations that can be removed from the entire suite, LOFT can remove 60.2% on top of FastTrack.

Such reduction can help a technique to reduce the size of the operation log for subsequent analysis such as execution reply, where such synchronization events play a key role in determining the interleaving sequence among threads in a replayed execution.

Time Analysis. From the column Time in TABLE I, we observe that, on average, LOFT runs 16.2% faster than FastTrack on completing these vector operations. On examining the time needed for each benchmark, we find that the variance in time is large among the set of runs for the same technique. For example, on dedup, the mean time for FastTrack is 8,337.9µs. However, we have experienced that some runs on this subject take 2 to 3 folds of time than this average value (e.g., 26,908µs, 11,219µs, and 19,108µs). Therefore, in order to compare FastTrack and LOFT on the time dimension more accurately, we present a graph in Figure 6 that compares FastTrack and LOFT using boxplot, where the dataset is the same as that used to produce TABLE I.

In Figure 6, each sub-figure shows a boxplot graph for its corresponding benchmark as marked in the title position, where the x-axis represents FastTrack and LOFT, and the y-axis represents the time needed in each of the 100 runs in microsecond. The lines in each box show the lower quartile, median and upper quartile time, respectively. Figure 6 shows that the time variance for bodytrack, dedup, facesim, and x264 can be large. However, we can still obviously see that the lower quartile, the median, and the upper quartile of LOFT are all lower than that of FastTrack, respectively, in each sub-figure except the median value in the plot entitled “blackscholes”.

We also compute the Mann-Whitney U Test result on the raw data presented in Figure 6. The result is shown in TABLE II. From TABLE II, we find that LOFT and FastTrack are

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Application Domain</th>
<th>Size (loc)</th>
<th># of worker threads</th>
<th>Vector operations</th>
<th>Time (µs)</th>
<th># of data races</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FT (A)</td>
<td>LOFT(B)</td>
<td>(B) + (A)</td>
</tr>
<tr>
<td>blackscholes</td>
<td>Financial Analysis</td>
<td>1,665</td>
<td>8</td>
<td>3.0</td>
<td>1.0</td>
<td>0.33</td>
</tr>
<tr>
<td>bodytrack</td>
<td>Computer Vision</td>
<td>11,891</td>
<td>8</td>
<td>6.520.4</td>
<td>3,205.0</td>
<td>0.49</td>
</tr>
<tr>
<td>cannel</td>
<td>Engineering</td>
<td>4,526</td>
<td>8</td>
<td>61.0</td>
<td>11.0</td>
<td>0.18</td>
</tr>
<tr>
<td>dedup</td>
<td>Enterprise Storage</td>
<td>3,704</td>
<td>8</td>
<td>17,545.9</td>
<td>14,276.1</td>
<td>0.81</td>
</tr>
<tr>
<td>facesim</td>
<td>Animation</td>
<td>29,428</td>
<td>8</td>
<td>49,021.1</td>
<td>25,318.4</td>
<td>0.52</td>
</tr>
<tr>
<td>raytrace</td>
<td>Rendering</td>
<td>13,323</td>
<td>8</td>
<td>291.1</td>
<td>112.8</td>
<td>0.39</td>
</tr>
<tr>
<td>streamcluster</td>
<td>Data Mining</td>
<td>2,429</td>
<td>8</td>
<td>314,333.8</td>
<td>131,021.4</td>
<td>0.42</td>
</tr>
<tr>
<td>swaptions</td>
<td>Financial Analysis</td>
<td>1,629</td>
<td>8</td>
<td>46.0</td>
<td>2.9</td>
<td>0.04</td>
</tr>
<tr>
<td>vips</td>
<td>Media Processing</td>
<td>131,103</td>
<td>4</td>
<td>11,724.3</td>
<td>8,221.7</td>
<td>0.70</td>
</tr>
<tr>
<td>x264</td>
<td>Media Processing</td>
<td>37,526</td>
<td>8</td>
<td>1,601.6</td>
<td>1,251.8</td>
<td>0.78</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>235,559</td>
<td></td>
<td>-</td>
<td>-</td>
<td>0.39</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td>235,559</td>
<td></td>
<td>-</td>
<td>-</td>
<td>0.420</td>
</tr>
</tbody>
</table>

TABLE I. COMPARISONS ON ALL VECTOR CLOCK OPERATIONS (FT REFERRING TO FASTTRACK)
In general, a LOFT-converted program maintains the additional data structure for the checking of our sufficient condition in LOFT can be fully compensated.

As we have stated in the implementation paragraph, compared to FastTrack, LOFT maintains one more variable for every thread or lock. We conjecture that the number of threads and locks in a real-life program is limited. The addition of each variable only means an extra space of one integer. The extra space needed for LOFT may be marginal.

We have not measured the size of an event log for execution replay after applying our operation removal technique. In the future, we will perform such an experiment.

VI. RELATED WORK

Existing data race detectors can be broadly classified into three categories: static, dynamic, and hybrid. In general, a static approach focuses on program analysis without executing the program; whereas the dynamic ones analyze the observed executions of the program to find data races or infer them, but their scopes are limited to those observed executions. A hybrid approach usually uses a static approach to find a candidate set of data races, and then uses a dynamic approach to verify these candidates. However, a dynamic or hybrid algorithm has other limitations such as potential omissions of racing pairs on program paths that have not been monitored. They are inapplicable to a piece of code (e.g., a library) that is not in a closed form or traces being unavailable. The three approaches complement one another.

Lockset-based algorithms have the advantages of interleaving insensitive when detecting races. For instance, Eraser [22] is an early attempt to apply dynamic race detection on multithreaded programs. It proposed to detect races when the intersection of the locksets held by two threads at an execution point is empty. A pure lockset-based algorithm does not use vector clocks in their algorithms, whereas LOFT removes redundant events on top of the tracking of such relations. The analysis result used in LOFT has not been explored by them.

Pozniansky and Schuster [19] developed MultiRace and DJIT+. MultiRace is a hybrid of lockset-based technique and happens-before based technique. It uses an Eraser-like algorithm to detect spurious races on a variable, and then invokes DJIT+ to check subsequent races. DJIT+ has been extensively reviewed in Section III. MultiRace postpones the time to use a precise happens-before based race detection algorithm, and yet some races before the invocation of DJIT+ may be missed to be reported. Our analysis result can be applied to optimize the DJIT+ phase of MultiRace. Rather than using a lockset based and happens-before based approaches separately, Yu et al. in RaceTrack [24] used them at the same time, and reported a data race whenever the lockset of a memory location becomes empty and multiple threads are still active in accessing this location. GoldiLocks [7] refines the traditional lockset based algorithm by also tracking the happens-before relations among events. We are unsure

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Mann-Whitney U Test Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>0.000620</td>
</tr>
<tr>
<td>bodytrack</td>
<td>&lt; 0.000001</td>
</tr>
<tr>
<td>canneal</td>
<td>&lt; 0.000001</td>
</tr>
<tr>
<td>dedup</td>
<td>&lt; 0.000001</td>
</tr>
<tr>
<td>facesim</td>
<td>&lt; 0.000001</td>
</tr>
<tr>
<td>raytrace</td>
<td>&lt; 0.000001</td>
</tr>
<tr>
<td>streamcluster</td>
<td>&lt; 0.000001</td>
</tr>
<tr>
<td>swaptions</td>
<td>&lt; 0.000001</td>
</tr>
<tr>
<td>vips</td>
<td>&lt; 0.000001</td>
</tr>
<tr>
<td>x264</td>
<td>&lt; 0.000001</td>
</tr>
</tbody>
</table>
whether a LOFT-similar strategy can be integrated with GoldiLocks.

Although LOFT is built on top of FastTrack [9], as mentioned in Section IV, their focuses are different. Pacer [3] used a sampling strategy that samples program execution at the memory accesses (read or write) level to reduce time overhead. LiteRace [16] maintained two copies of each function in the source code, and dynamically turned on and off the sampling of the read and write operations in a function. Both techniques fully track the happens-before relations for synchronization events in a program being monitored. LOFT works on the manipulation of vector clock operations related to threads and locks. Both LiteRace and Pacer did not explore this dimension.

Helgrind+ [11] was also a combination of lockset based and the happens-before based algorithms. Helgrind+ considered that the conditional variables as a synchronization idiom should also be monitored to track the happens-before relations among events so as to reduce the amount of false positives due to the lost signal problem [11]. It improved the precision of Helgrind [13]. Like FastTrack, LOFT does not detect condition variables.

AccuLock [23] was another detector that combines a lockset based algorithm and a happens-before based algorithm. Xie et al. observed that although the happens-before based detectors were fast and can avoid reporting false positives, they are sensitive to interleaving order among threads. Hence such detectors can only detect data races existed in certain execution. AccuLock used an improved lockset based algorithm (Lock-Subset [23]) and a relaxed happens-before relation (which discards the causal relations due to lock acquisition and release) to infer data races. It suffered imprecision to a certain extent. Because AccuLock did not track any vector clock for any lock, our strategy cannot be applied to it directly.

To iron out the thread-local memory locations from the pool of all memory locations, using a state machine event filter is popular in many detection detectors (e.g., Eraser [22], MultiRace [19], RaceTrack [24], and MulticoreSDK [21]), which not only improves the precision of the detectors, but also reduces the slowdown. Our approach can also be considered as an event filter. However, our approach retains the resultant happens-before graph being sound and precise, even after non-consecutive series of event removals on a trace.

Our model is based on the sequential consistency memory model. If the memory accesses cannot be guaranteed to be first-in-first-out, one may develop a similar strategy for adversarial memory models [8].

VII. CONCLUSION

In this paper, we have studied the problem of vector clock update reduction for the on-the-fly tracking of happens-before relations on an execution trace. We have quantified a sufficient condition that can soundly remove the involved vector clock comparisons and assignment of vector clock instances without affecting the precision of such tracking. We have also applied our result to data race detection to formulate LOFT. We have further conducted an experiment to validate our approach. The result has shown that, on average, on top of FastTrack, LOFT reduces 58.0% of all such vector comparison and updates, and runs 16.2% faster in completing the required tracking. We are generalizing the approach. It is interesting to integrate it with a guided execution strategy.

REFERENCES