To Appear in *IEEE Transactions on Parallel and Distributed Systems*

Lock Trace Reduction for Multithreaded Programs

Yan Cai and W.K. Chan

Abstract—Many happened-before based detectors for debugging multithreaded programs implement vector clocks to incrementally track the casual relations among synchronization events produced by concurrent threads and generate trace logs. They update the vector clocks via vector-based comparison and content assignment in every case. We observe that many such tracking comparison and assignment operations are removable in part or in whole, which if identified and used properly, have the potential to reduce the log traces thus produced. This paper presents our analysis to identify such removable tracking operations and shows how they could be used to reduce log traces. We implement our analysis result as a technique entitled LOFT. We evaluate LOFT on the well-studied PARSEC benchmarking suite and five large-scale real-world applications. The main experimental result shows that on average, LOFT identifies 63.9% of all synchronization operations incurred by the existing approach as removable and does not compromise the efficiency of the latter.

Index Terms—redundant operation optimization, threads, synchronization

1. Introduction

With the advent of multicore processor technology, developing multithreaded programs is increasingly popular [25]. A program *execution* of a multithreaded program typically consists of multiple *threads* [16], each of which executes a sequence of instructions. A program may use locks or user-defined synchronization idioms [21] to coordinate (or interleave) its threads to share resources (e.g., memory locations [16]).

Multithreaded program is difficult to reason due to the huge amount of interleaving sequences (even for the same input). Any improper interleaving may lead to the occurrences of concurrency bugs such as data races [10],[16],[32], atomicity or order violations [23], [25], and deadlocks [4],[13],[22]. Debugging such a program against a concurrency bug can be intricate because a concurrency bug often manifests itself into a failure only in some but not all thread interleaving schedules, even for the same input [25].

Typically, in debugging, developers and bug detectors require analyzing the available execution trace logs. Using a short (or shorter) trace that indicates the program locations of the detected failures (due to such concurrency bugs) and how the thread interleaving in the monitored execution leads to these failures can ease developers to diagnose the reported problems.

Many dynamic concurrency bug detectors (e.g., [16],[27],[30],[21],[32],[35]) use *the same* vector clock approach [29] to track the *happened-before* relations [27] among synchronization events acting on lock objects and events for thread management, which we refer to it as the FF algorithm [16]. FF *conservatively* logs all such events to ensure the causal relations between a pair of such events that can be inferred.

Can this conservative requirement be relaxed? Our analysis results show that such a relaxation not only is feasible in theory (Section 4) but also results in a reduced lock trace, which is significantly shorter than the original lock trace (Section 5). To the best of our knowledge, this paper is the first work on the relaxation of lock traces for multithreaded programs (which refer it to as a lock trace reduction technique).

The relaxation of such trace log requirements plays an important role in software testing and bug diagnosis. For example, in regression testing, as presented in a recent comprehensive survey [36], numerous test suite reduction, minimization, and prioritization techniques have been proposed. To the best of our knowledge, a vast majority of them use the coverage data on the executed program entities in individual trace logs of the corresponding test cases as their inputs. They (1) select a subset of such logs such that the subset still maximally includes all logged program entities at least once, (2) make such a subset to be minimal in terms of the number of logs retained, or (3) prioritize the corresponding trace logs. A common attribute of them is that they identify duplicated subsequences among trace logs, and only retain one subsequence for every set of duplicated subsequences.

Another example is bug reproduction. To reproduce an observed failure, many state-based deterministic replay techniques (e.g., [8],[19]) capture every input, schedule decision, and read or write value of each variable instance. They are valuable but slow, which should be improved [7]. ODR [7] efficiently finds an alternate execution trace log that produces the same failure as the original via an execution trace that may not be the same as the original trace to gain efficiency.

Our work is developed on top of an interesting ob-

- *Yan Cai is with Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: yancai2@student.cityu.edu.hk*
- *W.K. Chan is with Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: wkchan@cityu.edu.hk.* (contact author)

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography

servation: In the course of an execution of a multi-threaded program, a thread may *consecutively* acquire the same lock; and during which, no other thread uses this particular lock. If this is the case, as long as at the end of every such period, the vector clocks associated with this particular lock and the thread in question can be maintained *properly*, all the intermediate updates on these vector clocks may be skipped. The key technical challenge is to characterize a condition that a log trace reduction technique can safely replace the involved tracking operations on these vector clocks by some maintenance of thread-local data structure. Moreover, it is desirable for this condition to be effective and applicable to a wide range of applications. To ease our presentation, this paper refers to such a vector-based operation (e.g., comparison and copy between two vectors) that can be skipped as a *removable* operation.

In [12], we identified synchronization events in the online data race detection that one may remove from trace logging. This paper generalizes the preliminary work [12] by analyzing how nested locks and synchronization barriers can be handled and presenting a generalized framework. The evaluation of this paper has also significantly extended by including five additional large-scale real-world benchmarks and validating the technique comprehensively.

Specifically, we analyze all the scenarios involving a pair of consecutive synchronization events of the same thread, and show that our formulated condition can infer the presences of removable operations precisely. With this result, we define a new scheme to generate happened-before lock traces, and refer to the proposed technique as LOFT. We have evaluated LOFT on a well-studied PARSEC benchmark suite [9],[1],[15],[21] and five large-scale open-source C/C++ real-world applications (Apache `Httpd` [1], `Chromium` [2], `Firefox` [3], `MySQL` [4], and `Thunderbird` [5]). The experimental result shows that, compared to FF, LOFT identifies 63.9% of the tracked operations being removable; and on average, and the size of the generated lock traces is reduced by 48.9% and 27.4% before and after compression, respectively. Moreover, the mean tracking time of LOFT is 12−24% shorter than FF, which shows that the reduction can be achieved without compromising its performance. Additional experimentations on selected Java benchmarks can be found in the online supplementary document.

The main contribution of this paper is threefold. (1) This paper proposes the first online approach to precisely skipping removable operations from logging. (2) We present a theoretical model that shows the correctness of our algorithm. (3) We report an experiment that confirms that LOFT is both effective and efficient.

In the rest of the paper, Section 2 presents a motivating example. Section 3 revisits the preliminaries. Section 4 presents our analysis and algorithm (LOFT). Section 5 reports an experiment to validate LOFT. Section 6 reviews related work. Section 7 concludes the paper.

2. MOTIVATING EXAMPLE

Figure 1(a) shows a motivating example adapted from the classical *Producer and Consumer Problem*. It shows a shared location `pool` protected by a lock `m` and two threads `Producer` and `Consumer`. The `Producer` repetitively produces a datum, and puts it into `pool`, from which the `Consumer` repetitively fetches a datum. The lock `m` aims to protect `pool` from concurrent accesses by the two threads at lines s_1, s_2, s_3 and s_4.

Figure 1(b), from top to bottom, shows a program execution of the program. The two threads are indicated by the rightmost (`Producer`) and the leftmost (`Consumer`) columns of Figure 1(b). The `Producer` thread firstly acquires the lock `m` and then releases it twice (at lines e_1−e_4). Then, the `Consumer` thread acquires and then releases this lock twice (at lines e_5−e_8). Finally, the `Producer` thread acquires and then releases the same lock once more (at lines e_9−e_{10}). In total, the execution has 5 lock *acquisition* and 5 lock *release* events.

FF [16],[35]: The algorithm FF (see Section 3.3 for more discussion) firstly sets up three vector clocks for `Consumer`, `m`, and `Producer`, denoted by C_c, L_m, and C_p, respectively, shown as the three columns in the middle part of Figure 1(b). To track each lock *acquisition* or *release* operation, FF needs to perform two vector-based operations (one for comparing two vector clocks; and another one for updating a vector clock for L_m, C_p, or C_c). As such, the algorithm needs in total 10 such vector clock operations. Figure 1(b) also shows the values of the three vector clocks along the execution.

Our observation: The lock *acquisition* (*release*, respectively) operations marked with a star "*" ("−", respectively) symbol in Figure 1(b) either changes no content of any vector clock (see those fully shaded vector clocks) or merely assigns a value to one entry (see those partially shaded vector clocks) of exactly one vector clock. The reason is that the same lock is acquired and then released by the same thread consecutively without any interruption from other threads. As such, there is no need to update the vector clock for the

```
Shared variables:  int: pool[1000];   Lock: m;   bool: Empty, Full;
Consumer                                  Producer
  while(true)                               while(true)
  {                                         {
    while(Empty)sleep(100);                   while(Full) sleep(100);
s1  Acquire(m);                           s3  Acquire(m);
      //fetch a datum from pool                 //add a datum to pool
s2  Release(m);                           s4  Release(m);
  }                                         }
```

(a) The code

A sample interleaving scenario.
(for brevity, we only show *acquire* and *release* operations)

Consumer	C_c	L_m	C_p	Producer
	⟨1, 1⟩	⟨0, 0⟩	⟨1, 1⟩	
		⟨0, 0⟩	⟨1, 1⟩	e_1 Acquire(m); *
		⟨1, 1⟩	⟨1, 2⟩	e_2 Release(m); +
		⟨1, 1⟩	⟨1, 2⟩	e_3 Acquire(m); *
		⟨1, 2⟩	⟨1, 3⟩	e_4 Release(m); -
e_5 Acquire(m); +	⟨1, 2⟩	⟨1, 2⟩		
e_6 Release(m); -	⟨2, 2⟩	⟨1, 2⟩		
e_7 Acquire(m); *	⟨2, 2⟩	⟨1, 2⟩		
e_8 Release(m); -	⟨3, 2⟩	⟨2, 2⟩		
		⟨2, 2⟩	⟨2, 3⟩	e_9 Acquire(m); +
		⟨2, 3⟩	⟨2, 4⟩	e_{10} Release(m); -

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

Figure 1. An Example: Consumer and Producer

thread to collect the timestamp of some other threads.

For this reason, the involved vector-based operations for these "*" and "–" rows can be either removed or replaced by simpler and native assignment operations on primitive data type. The "*" rows are particularly interesting as they shorten the lock trace as well.

In summary, by a proper strategy, only the operation marked with a plus symbol "+" needs to take a vector-based operation. Seven other vector-based operations are *removable*, which can be substituted by scalar assignments (such as updating the value in the initialized vector clock of the lock `m` from "`1`" to "`2`" on the second release of the `Producer` thread, which is the current timestamp of the `Producer` thread) or such assignments can be skipped completely. By doing so, a technique can reduce the length of a trace log on these critical synchronization events. For example, after reduction, the trace log for the execution in Figure 1(b) only includes six events: $\langle e_1, e_4, e_5, e_8, e_9, e_{10}\rangle$.

3. PRELIMINARIES

3.1 Synchronization Events for Monitoring

The algorithm FF [16],[23],[30] for a happened-before based detector typically monitors a set of critical operations such as the events for *lock acquisition* (*acq* for short) and *release* (*rel* for short) for each lock m, the events for thread management (*fork* or *join*) of each thread t as well as other events like *wait*, *notify* and *notifyAll*, and *barrier*. In this paper, we present how our model handles this set of above eight critical operations; as the extension to cover other events is similar and quite straightforward [16]. This paper also uses the terms "event" and "operation" interchangeably.

We assume that a lock can only be acquired by at most one thread at a time, which is enforced by the underlying programming model such as JVM or the C/C++ Pthreads library; a thread can only release a lock that it is holding; and the execution being monitored is sequentially consistent.

A lock trace σ is the projection of an execution of a program p on the above set of critical operations. In particular, if t is a thread in an execution of a program p, we use σ_t to denote the projection of σ on t. For example, for the execution of the `Consumer` thread in Figure 1(b), $\sigma_{Consumer} = \langle e_5, e_6, e_7, e_8\rangle$.

One popular lock usages in an execution is the presence of nested locks. For example, suppose that $\sigma_t = \langle \ldots acq(m), acq(l), rel(l), rel(m) \ldots\rangle$ is a trace projection on the thread t, where m and l are two locks, and the pair of *acq* and *rel* of the lock l is nested within the pair of *acq* and *rel* of the lock m. If an inner lock (e.g., l in trace σ_t) is the same as its outer lock (e.g., m in trace σ_t), this situation is called *lock reentrance*, which is normally handled by the underlying framework [17], and is transparent to FF.

3.2 The Happened-before Relation

A *happened-before relation*, denoted by $\overset{hb}{\rightarrow}$, is a partial order of pairs of events in a parallel system [27]. It is defined by the following three conditions.

Algorithm: **FF**

On initialization:
1. For each thread t, $C_t[i]$ =1, where i is from 1 to n.
2. For each lock m, $L_m[i]$ = 0, where i is from 1 to n.

On *acquiring* a lock m for thread t:
3. $C_t[i]$ = max $\{C_t[i], L_m[i]\}$, where i is from 1 to n.

On *releasing* a lock m for thread t:
4. $L_m[i] = C_t[i]$, where i is from 1 to n.
5. $C_t[t] = C_t[t]$ +1.

Figure 2. The FF algorithm that tracks happened-before relations for synchronization events.

(a) *Program order*: If α and β are any two critical events performed by the same thread and α precedes β, then we write $\alpha \overset{hb}{\rightarrow} \beta$.

(b) *Release* and *acquire*: If α is a release operation of a lock m, and β is an acquire operation of the same lock m performed by a thread different from the one performing α, and α precedes β, then we write $\alpha \overset{hb}{\rightarrow} \beta$.

(c) *Transitivity*: if $\alpha \overset{hb}{\rightarrow} \beta$ and $\beta \overset{hb}{\rightarrow} \gamma$, then $\alpha \overset{hb}{\rightarrow} \gamma$.

3.3 The Algorithm FF

In this section, we revisit the algorithm FF [16],[23],[30] as shown in Figure 2.

A *timestamp* is a number. A vector clock is a finite sequence of timestamps, the size of which is at least the same as the number of threads in the trace. The algorithm FF assigns one vector clock C_t to each thread t. This vector clock C_t logs the current timestamp of the thread t as well as the timestamps of other threads visible to t. It also assigns one vector clock L_m to each lock m.

Each thread t has its own timestamp variable that is incremented by 1 on each *release* operation on any lock performed by t. At the same time, supposed that m is the lock released by the thread t, then the vector clock L_m records a snapshot of C_t when t releases m. On *acquiring* the lock m by t, the vector clock C_t records the current timestamp of the thread t and those of other threads gotten from the vector clock L_m of the lock m.

We firstly present some standard auxiliary notations to ease our subsequent presentation. We then describe each case with the aid of Figure 2:

- Suppose that V_1 and V_2 are two vector clocks, and the number of elements in either one is n. If $V_1[i] \leq V_2[i]$ for $1 \leq i \leq n$, we denote this condition by $\boldsymbol{V_1 \sqsubseteq V_2}$. Similarly, if $V_1[i] = V_2[i]$ for $1 \leq i \leq n$, we denote this condition by $\boldsymbol{V_1 = V_2}$. (Note that both relations "$\sqsubseteq$" and "=" are transitive.)
- We also define $\boldsymbol{V_1 \sqcup V_2}$ to be a vector clock 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 vector clock is also n.
- The checking of every such operation (i.e., $V_1 \sqsubseteq V_2$, $V_1 = V_2$, and $V_1 \sqcup V_2$) is $O(n)$ in time complexity.

To maintain the data structure, FF uses the following strategies. For every *acquisition* on the lock m by the thread t, FF updates C_t so that each of its entries is the maximum of the corresponding entries in L_m and C_t (i.e., $C_t = C_t \sqcup L_m$) as shown at line 3 in Figure 2. On every *release* of a lock m, FF copies the contents of C_t to L_m (i.e., $L_m = C_t$), and then increments the timestamp kept at $C_t[t]$ by 1 as shown at lines 4 and 5 in Figure 2.

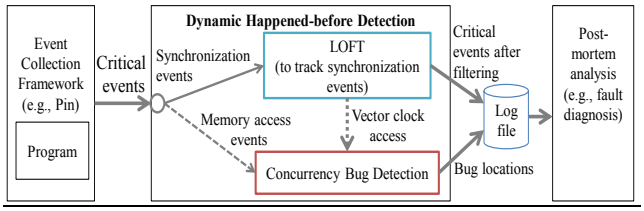

Figure 3. The overview of LOFT

4. Our Analysis and The LOFT Algorithm

Figure 3 depicts an overview on the role of LOFT. LOFT replaces the FF algorithm in the central box for the purpose of generating shorter lock traces.

In Figure 3, a program has been instrumented by a dynamic instrumentation framework [17][26] to generate critical events (i.e., `synchronization` and `memory access` events). The event collection framework forwards every such generated synchronization event to LOFT, or forwards it to the other units of the detectors (e.g., bug detector) if it is a non-synchronization event.

As stated in Section 3.1, pair of lock acquisition and release events may be nested in another pair(s) of lock acquire and release events. We firstly analyze the non-nested scenarios in Section 4.1, and then extend our analysis to handle nested locks in Section 4.2. In Section 4.3, we present the LOFT algorithm.

4.1 Non-nested Lock Acquisition and Release

We characterize the situation of lock *acquisition* and *release* operations by enumerating all possible scenarios in between a pair of critical operations consecutively performed by the same thread. We organize them as six cases as depicted in Figure 4.

We use e_j (for j=1, 2 …) to denote a critical operation. We also define two auxiliary functions: **`lastLock`**(t) denotes the most recent lock that the thread t has released, and **`lastThread`**(m) denotes the most recent thread that has released the lock m. For instance, in the motivating example, when the `Consumer` thread reaches the line e_5 to acquire the lock m, it has not yet acquired any lock. So, with respect to *acquire*(m) at e_5, the function `lastLock`(`Consumer`) returns *null*. At this moment, the lock m has been most recently released by the `Producer` thread at e_4. So, the function `lastThread`(m) at e_5 returns `Producer`.

On *acquiring* or *releasing* a lock m by a thread t, there are four cases for each operation according to whether the thread returned by `lastThread`(m) is t and whether the lock returned by `lastLock` (t) is m:

- `lastThread`(m) = t and `lastLock` (t) = m.
- `lastThread`(m) ≠ t and `lastLock` (t) = m.
- `lastThread`(m) = t and `lastLock` (t) ≠ m.
- `lastThread`(m) ≠ t and `lastLock` (t) ≠ m.

On lock acquisition, as indicated at line 3 in Figure 2, the FF algorithm wants to ensure that $L_m \sqsubseteq C_t$ holds. However, if $L_m \sqsubseteq C_t$ is already holding right before this *acquire(m)* event occurs, the corresponding vector clock comparisons and updates will not change the content of C_t. Therefore, if `lastLock`(t) = m holds (irrespective to whether the condition `lastThread`(m) = t holds or not) on acquiring m by t, it suffices to infer that $L_m \sqsubseteq C_t$ must hold (see the Case 1 and the Case 2 in following analysis). However, on handling lock releases, $C_m = C_t$ is required to be held, which may not hold if the timestamps of the thread t has been updated. So, we still need to analyze whether $C_m = C_t$ holds in each of the four cases. In short, there are two cases for *lock acquisitions* and four cases for *lock releases* to consider.

We are going to analyze these six cases in details. In each case, the condition refers to the one when the event e_* in the case occurs, which is also the highlighted (bold ***acq*** or ***rel***) event for the corresponding case example depicted in Figure 4. In this figure, both t and s are threads; both m and l are locks; and *rel*(m) and *acq*(m) represent the *release*(m) and *acquire*(m) events, respectively. We use the notation "→" to stand for the direction of vector clock assignment between a pair of thread and lock: (1) (green) arrows ended with a dot correspond to the removable *acq*(m) or *rel*(m) operations. (2) Dotted arrows show these *acq*(m) or *rel*(m) operations, which are not removable. (3) gray arrows are just for references and are not important to our analysis. Besides, in the following description, the sub-case where the event e_1 does not exist is straightforward for analysis, and for brevity, we omit its discussion. For each of the 6 cases, we outline the proof of correctness. The detailed proof is shown in Appendix A.

We firstly analyze scenarios in which e_* is an **acquire**(t, m). There are two cases: Case 1 and Case 2.

Case 1. [if `lastThread`(m) = t].

Let e_1 be an event in a trace where t releases m such that `lastThread`(m) = t, and e_* be an event in the same trace that t acquires m after the occurrence of e_1.

Consider the trace ⟨… e_1 … e_* …⟩, as illustrated by `Case 1` of Figure 4. When e_1 occurs, according to FF (at lines 3 and 4 in Figure 2), we have $L_m \sqsubseteq C_t$.

When the event e_* occurs, because the condition `lastThread`(m) = t holds, L_m must have not been changed by any other thread. However, the values in C_t may be incremented because t may have acquired some other lock(s) in between e_1 and e_* (as illustrated by "***" in `Case 1` of Figure 4); otherwise, C_t must remain unchanged. No matter C_t is incremented or not, we have $L_m \sqsubseteq C_t$. Therefore, there is no need to perform any comparison between L_m and C_t, and the comparison can be removed (shown as an arrow ended with a dot in `Case 1` of Figure 4) when e_* occurs.

Case 2. [if `lastThread`(m) ≠ t].

Let e_* be an event that t acquires m.

Consider the trace ⟨… e_* …⟩. When e_* occurs, because we have `lastThread`(m) ≠ t, there are two sub-cases to consider: m must either have been released by another thread t' (i.e., $t' \neq t$) 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 $C_{t'} \sqcup L_m$ (as illustrated by the first arrow in `Case 2` of Figure 4). In the latter case, the value of L_m must be different from that of C_t because all locks are initialized as all 0s, whereas all threads are initialized as all 1s (see lines 1 and 2 of Figure 2). Therefore, without further checking, we cannot

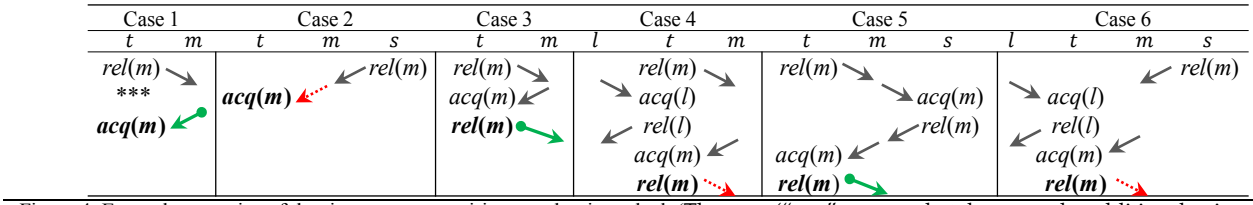

Figure 4. Example scenarios of the six cases on acquiring or releasing a lock (The stars ("***") mean that there may be additional pairs of $acq(x)$ and $rel(x)$ where x can be any lock (except m).)

determine a definite happened-before relationship between L_m and C_t. As such, when e_* occurs, the corresponding comparison and its associated potential assignment from $C_t \sqcup L_m$ to C_t are necessary. In other words, no operation is removable (shown as the second (dotted) arrow in `Case 2` of Figure 4).

We now analyze scenarios in which e_* is a **release**(t, m). There are four cases: Cases 3–6.

Case 3. [if `lastThread`(m) = t and `lastLock`(t) = m].

Let e_1 be an event where t releases m such that `lastThread`(m) = t and `lastLock`(t) = m. Also let e_* be the event that t releases m, and e_2 be the corresponding lock acquisition event performed by t with respect to e_*.

Consider the trace $\langle \dots e_1 \dots e_2 \dots e_* \dots \rangle$, which is also depicted as `Case 3` in Figure 4. When e_2 occurs, because `lastThread`(m) = t holds, the situation is the same as that in `Case 1`. As such, both L_m and C_t share the same contents except for the entries $L_m[t]$ and $C_t[t]$.

When e_* occurs, only $C_t[t]$ but not $L_m[t]$ is changed. So, to make $L_m = C_t$ true, we only need to update $L_m[t]$ to be $C_t[t]$; and the removable vector clock assignment from C_t to L_m can be eliminated (which is depicted by the third arrow ended with a dot in `Case 3` of Figure 4).

Case 4. [if `lastThread`(m) = t and `lastLock`(t) ≠ m].

Let e_1 be an event that t releases m such that `lastThread`(m) = t, e_* be an event that t releases m, and e_2 be the corresponding lock *acquire* operation of e_*. These three events are depicted as the first ($rel(m)$) and the last two events ($acq(m)$ and $rel(m)$) in `Case 4` of Figure 4.

Consider the trace $\langle \dots e_1 \dots e_2 \dots e_* \dots \rangle$. When e_1 occurs, because `lastThread`(m) = t holds, the situation is the same as that in `Case 1`. As such, both L_m and C_t share the same contents except for the entries $L_m[t]$ and $C_t[t]$.

When e_2 occurs, because of the condition `lastLock`(t) ≠ m, C_t may have been adjusted due to acquiring and releasing some other lock in between e_1 and e_2 (as depicted by the $acq(l)$ and $rel(l)$ operations in Figure 4). So, we only have $L_m \sqsubseteq C_t$ (instead of $L_m = C_t$).

When e_* occurs, the condition `lastThread`(m) = t implies that m has not been acquired and released by any other thread in between e_1 and e_*. Hence, L_m remains unchanged since $e_1$occurs. Besides, C_t is also not updated because no release operation has been done in between e_1 and e_*. To ensure the condition $L_m = C_t$ true, we cannot remove the assignment from C_t to L_m .

Case 5. [if `lastThread`(m) ≠ t and `lastLock`(t) = m].

Let e_1 be an event that t releases m such that `lastLock`(t) = m, e_* be an event that t releases m, and e_2 be the corresponding lock *acquire* operation of e_*. These three events are depicted as the first ($rel(m)$) and the last two events ($acq(m)$ and $rel(m)$) in `Case 5` of Figure 4.

Consider the trace $\langle \dots e_1 \dots e_2 \dots e_* \dots \rangle$. When e_1 occurs, according to FF at line 4 in Figure 2, both L_m and C_t share the same contents except for the entries $L_m[t]$ and $C_t[t]$. When e_2 occurs, because `lastThread`(m) ≠ t holds, like `Case 2`, the vector clock C_t will be updated to $C_t \sqcup L_m$. The net result is that the two vector clocks L_m and C_t only differ at the positions $L_m[t]$ and $C_t[t]$.

When e_* occurs, we only need to update $L_m[t]$ to be $C_t[t]$. As such, the vector clock assignment from C_t to L_m can be eliminated.

Case 6. [if `lastThread`(m) ≠ t and `lastLock`(t) ≠ m].

In this case, we know nothing about the relation between L_m and C_t. Therefore, no vector-based comparison or assignment can be removed. An example scenario is depicted as `Case 6` in Figure 4: when the thread t releases the lock m, t has acquired another lock (i.e., the lock l) and m has been acquired and released by another thread (i.e., the thread s).

Analysis Summary: For `Case 1`, we may skip the corresponding tracking on the event e_*. For `Case 3` and `Case 5`, we can merely update $L_m[t]$ to be $C_t[t]$ without applying the tracking as used in the FF algorithm, and can still achieve the same happened-before tracking result. For the other cases, we need to apply FF.

4.2 Nested Lock Acquire and Release

In order to handle nested locks, we are going to further analyze each of the six cases presented in Section 4.1, and enhance each case if necessary.

In `Cases 2`, `4`, and `6`, no vector comparison or assignment is removed. Hence, even in the nested locks scenarios, the analyses require no revision. Therefore, we only need to analyze the nested versions for `Cases 1`, `3`, and `5`. We call their nested lock versions as `Cases 1'`, `3'`, and `5'`, respectively. Figure 5 illustrates a scenario for each case.

In `Case 1'`, if a thread t is holding other locks (l in `Case 1'`) when it reacquires a lock m, C_t may have been incremented. However, this operation does not affect the condition $L_m \sqsubseteq C_t$. Hence, the comparisons between L_m and C_t can still be safely eliminated.

In `Cases 3'` and `5'`, if a thread t is holding other locks (e.g., l in `Case 3'` and `Case 5'` in Figure 5) when

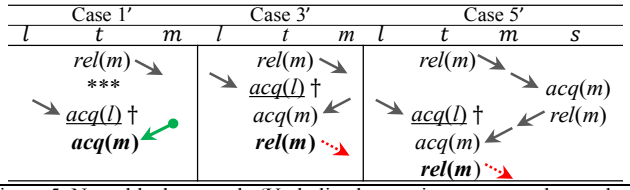

Figure 5. Nested lock example (Underlined operations are nested outer lock acquires. † indicates that where there may exist more $acq(x)$ operations that acquire a lock x. Other symbols are the same as that in Figure 4)

Algorithm: **LOFT** State: *C: Tid* → (*VC, Lock, n*) *L: Lock* → (*VC, Tid*)

On **Acquire** (*t, m*)

[**lastThread** (*m*) = *t*]

$$\frac{t.n' := t.n + 1}{(C, L) \Rightarrow^{acquire\ (t,m)} (C, L)}$$

[Otherwise]

$$\frac{C' = C[t := C_t \sqcup L_m] \quad t.n' := t.n + 1}{(C, L) \Rightarrow^{acquire\ (t,m)} (C', L)}$$

On **Release** (*t, m*)

[**lastLock**(*t*) = *m* and $t.n = 1$]

$$\frac{L' = L[m := L_m[t := C_t(t)]] \quad C' = C[t := inc_t(C_t)] \quad t.n := 0 \quad m.Tid := t}{(C, L) \Rightarrow^{release\ (t,m)} (C', L')}$$

[Otherwise]

$$\frac{L' = L[m := C_t] \quad C' = C[t := inc_t(C_t)] \quad t.n := 0 \quad m.Tid := t \quad t.Lock := m}{(C, L) \Rightarrow^{release\ (t,m)} (C', L')}$$

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

$$\frac{C' = C[u := C_u \sqcup C_t, t := inc_t(C_t)]}{(C, L) \Rightarrow^{fork\ (t,u)} (C', L)}$$

On **Join**(*t, u*)

$$\frac{C' = C[u := C_u \sqcup C_t, u := inc_u(C_u)]}{(C, L) \Rightarrow^{join\ (t,u)} (C', L)}$$

On **Wait**(*t, m*)

[pre-wait(*t, m*)]

call Release(*t, m*)

[post-wait(*t, m*)]

call Acquire(*t, m*)

On **Barrier**(*t, b*)

[pre-barrier(*t, b*)]

$$\frac{L' = L[b := C_t \sqcup L_b]}{(C, L) \Rightarrow^{acquire\ (t,m)} (C, L')}$$

[post-barrier(*t, b*)]

$$\frac{C' = C[t := C_t \sqcup L_b]}{(C, L) \Rightarrow^{acquire\ (t,m)} (C', L)}$$

Figure 6. The LOFT Algorithm

it reacquires the lock m, C_t may have been incremented due to acquiring some other locks. In between the *acquisition* and *release* events of m, the thread t might have already acquired other locks, some of which have not been released yet. Therefore, on the *release* of m, the condition $C_t[i] = L_m[i]$ $(1 \le i \le n, i \ne t)$ may not hold. So, the assignment from the whole vector clock C_t to L_m is necessary. To distinguish `Case 3` from `Case 3'` and `Case 5` from `Case 5'`, we define the third auxiliary function **`acqCounter`**(t). It returns the number of locks that the thread t has acquired but not yet released since its most recently released lock returned by `lastLock`(t).[1]

Analysis Summary: considering both non-nested lock usage and nested lock usage, the vector clock comparison can be removed if either of the following two conditions is satisfied.

- On ***acquire***(m) by t: [`lastThread`(m) = t], or
- On ***release***(m) by t: [`lastLock`(t) = m and `acqCounter`(t)= 1].

4.3 The LOFT Algorithm

Figure 6 shows the LOFT algorithm. Apart from the introduction of the conditions stated at the end of Section 4.2, LOFT extends FF by adding two variables (*Lock* and n) to each thread and one variable (*Tid*) to each lock as shown in the `State` section in Figure 6.

To ease our presentation, we use the same notations as those used in [16]. LOFT maintains an analysis state (C, L) composing of two parts: (1) C maps each thread t (identified by a unique identity *Tid*) to a vector clock C_t, a lock m (denoted as $t.Lock$) to keep `lastLock`(t), and a counter n to keep `acqCounter`(t). (2) L maps each lock m to a vector clock L_m and a thread t (denoted as $m.Tid$) to keep `lastThread`(m).

Initially, each thread is mapped to a triple: a newly initialized vector clock with all contents as 0s, an empty lock, and the variable n to 0. Moreover, each lock is mapped to an empty thread and a newly initialized vector clock with 1 in every entity of it.

Operations on Lock Acquisition: As shown in Figure 6, on acquiring a lock m by a thread t, LOFT firstly checks whether **`lastThread`**(m) = t holds (i.e., $t = m.Tid$). If this condition is satisfied, LOFT does nothing on C'. Otherwise, it performs $C' = C[t := C_t \sqcup L_m]$, where the notation $C' = C[t := x]$ means that C' is constructed from C by substituting the entry $C[t]$ by x. Finally, LOFT increments $t.n$ by 1 (that is $t.n' := t.n + 1$).

Operations on Lock Release: On releasing a lock m by a thread t, LOFT firstly checks whether **`lastLock`**(t) = m (i.e., $m = t.Lock$) and $t.n = 1$ hold. If these two conditions are satisfied, $L' = L[m := L_m[t := C_t(t)]]$ is performed; otherwise, an $O(n)$ operation $L' = L[m := C_t]$ is performed. Lastly, LOFT increases the timestamp of the thread t ($C' = C[t := inc_t(C_t)]$, where $inc_t(X)$ means $X = [t := X[t] + 1]$), and resets $t.n$ to 0. It also updates the mapping (**`lastThread`** and **`lastLock`**) between m and t by performing both $m.t := Tid$ and $t.Lock := m$.

Other operations: There are some other operations that must be considered to track the happened-before realtions among the critcial events. These operations include *fork*(), *join*(), *wait*(), *notify*(), *notifyAll*(), and *barrier*(), as shown in Figure 6. LOFT takes *fork*() and *join*() in the same way as presented in [16]. Other operations (*wait*(), *notify*(), *notifyAll*(), and *barrier*()) are monitored as follows:

On *wait*(), LOFT firstly performs a lock *release* operation (*pre-wait*()). After either a *notify*() or a *notifyAll*() is performed, the *wait*() operation will return to its caller. Just before the return from the *wait*(), LOFT performs a lock acquisition *acquire*() (*post-wait*()). In this way, the happened-before relation on the pair of *wait*() and *notify*() or *wait*() and *notifyAll* () can be monitored.

Every *barrier*() operation involves a barrier instance b. Every such an instance is mapped to a vector clock (L_b) in L. (Because any barrier instances can be distinguished from any lock instance by their distinct addresses, we can safely use the same L to map all locks and all barriers to their vector locks without causing any mistake.) However, to handle a *barrier*() operation, the required action is different from that to handle a lock acquisition or release. For each barrier instance b, L_b is initialized to contain 0 in each entity. On *pre-barrier*(t, b), LOFT performs $L_b = L_b \sqcup C_t$. After all threads reach at the barrier b, L_b collects the newest vector clocks of each thread that has called *barrier*(b). When a thread t returns from a *barrier*() call, a *post-barrier*(b) operation is performed so as to update C_t to reflect the latest timestamps of all other threads involved in the *barrier*(b) by performing $C_t = L_b \sqcup C_t$, as shown in Figure 6.

[1] Note that `acqCounter`(t) is not the total number of locks that t has acquired but not released. `acqCounter`(t) is set to be 0 initially. On *acquire*(x) (where x is a lock) by thread t, `acqCounter`(t) is incremented by 1; and on *release*(y) (where y is also a lock), `acqCounter`(t) is **reset** to be 0.

TABLE I. Comparisons on effectiveness and its efficiency of LOFT and FF on C/C++ benchmarks.

Benchmarks	Application Domain	Size (lines of code)	# of worker threads	# of Vector operations			Time (μs)		
				FF (A)	LOFT (B)	(B) ÷ (A)	FF (C)	LOFT (D)	(D) ÷ (C)
facesim	Animation	29,428	8	49,021.1	25,318.4	0.52	18,146.3	16,057.8	0.88
raytrace	Rendering	13,323	8	291.1	112.8	0.39	113.6	97.0	0.85
bodytrack	Computer Vision	11,891	8	6,520.4	3,205.0	0.49	2,819.4	2,283.4	0.81
swaptions	Financial Analysis	1,629	8	46.0	2.0	0.04	18.8	15.8	0.84
blackscholes	Financial Analysis	1,665	8	3.0	1.0	0.33	1.7	1.3	0.76
canneal	Engineering	4,526	8	61.0	11.0	0.18	25.3	21.3	0.84
streamcluster	Data Mining	2,429	8	314,333.8	131,021.4	0.42	109,798.1	95,347.9	0.87
MySQL	Database	1,015,047	12	396,569.7	135,000.5	0.34	130,615.4	112,275.8	0.86
vips	Media Processing	131,103	4	11,724.3	8,221.7	0.70	4,004.9	3,454.4	0.86
dedup	Enterprise Storage	3,704	8	17,545.9	14,276.1	0.81	9,661.3	8,337.9	0.86
x264	Media Processing	37,526	8	1,601.6	1,251.8	0.78	671.2	517.4	0.77
Httpd	Web server	47,145	30	1,896.0	56.0	0.03	699.5	590.4	0.84
Chromium	Web browser	3,907,957	22	631,223.4	117,382.7	0.19	184,816.2	150,369.5	0.80
Firefox	Web browser	3,807,299	17	3,819,783.2	293,813.3	0.08	1,335,533.0	1,096,242.5	0.81
Thunderbird	E-mail client	4,252,805	21	1,979,290.8	215,625.3	0.11	692,602,4	567,992.2	0.82
Total	-	13,267,477	-	7,229,911.30	945,299.00	0.131	2,489,527.05	2,053,631.66	0.825
Mean	-	-	-	-	-	0.361	-	-	0.831

5. EXPERIMENT

5.1 Implementation and Benchmark

To support our evaluation, we implemented both FF and LOFT for C++ programs with Pthreads on top of *Pin* 2.9 [26], which is a program dynamic instrumentation tool. For LOFT, on top of the implementation of FF, we further added a 32-bit integer to every lock to record the last thread (`lastThread`) that releases the lock concerned and a 32-bit integer to every thread, with its first 16 bits to record the most recent lock (`lastLock`) released by the thread concerned and with its last 16 bits to record the number of the locks (`acqCounter`) that this thread has acquired but did not released after the most recently released lock which is the same as its `lastLock`. 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 FF. The introduction of the additional integers in our technique does not affect this worst case space complexity order.

We also implemented LOFT and FF for data race detection as $\mathbf{LOFT}_{race}$ and $\mathbf{FF}_{race}$, respectively. See Appendix B for the implementation details. $\mathbf{FF}_{race}$ is essentially the state-of-the-art FastTrack algorithm [16].

5.2 Benchmark

We selected the PARSEC benchmark suite 2.1 [9] and five widely-used large-scale real-world open-source applications to evaluate LOFT. These large-scale applications (Apache `Httpd` [1], `Chromium` [2], `Firefox` [3], `MySQL` [4], and `Thunderbird` [5]) have been used to evaluate techniques in [13]. The PARSEC benchmark suite includes a set of multithreaded programs for emerging and system applications [9]. The application domains include financial analysis, engineering, computer vision, enterprise storage, animation, rendering, data mining, and media processing (see TABLE I for details). These benchmarks have been well studied to validate the concurrency related experiments including concurrent bug detection techniques ([11],[15],[21],[33]). The PARSEC benchmark suite includes 13 benchmarks: `black-scholes`, `bodytrack`, `canneal`, `dedup`, `facesim`, `ferret`, `fluidanimate`, `freqmine`, `raytrace`, `stream-cluster`, `swaptions`, `vips`, and `x264`. Among these benchmarks, `freqmine` does not use the Pthreads library, we discard it because our implementations were built on top of the Pthreads library; `ferret` and `fluidanimate` crashed even we ran them under the Pin environment (without the use of our tool) due to segmentation faults. All other 10 benchmarks can be successfully invoked and completed under the Pin environment. For each PARSEC benchmark, we use the `simsmall` input test. For each remaining benchmark, our test harness starts and then stops it as what we did in [13].

5.3 Experimental Setup

We performed the experiment on a machine running the Ubuntu 10.04 Linux with 3.16GHz Duo2 processor and 3.25G physical memory. Each benchmark was run 100 times. TABLE I shows the average number of vector operations performed and time needed to complete all such tracking on each benchmark (`Vector operations` and `Time`, respectively). For the PARSEC benchmark suite, we set each benchmark to have 8 worker threads except *`vips`* which were preset to have 4 fixed worker threads in the downloaded suite. For other five applications, we used their owned configurations, which cannot be changed by us.

We also experimented to see how the actual size (in terms of bytes) of the log files can be affected by our scheme. The log files were generated using the following scheme: on *`acquire`*(t, m) or *`release`*(t, m): FF_{race} adds a corresponding event line "`Thread t [acquires] a lock m.`" or "`Thread t [releases] a lock m.`" to the log file for each benchmark. For LOFT_{race}, if both conditions [`lastThread`(m) = t] and [`lastLock`(t) = m and `acqCounter`(t) = 1] are satisfied on *`acquire`*(t, m) or *`release`*(t, m), it does nothing; otherwise, it also adds a corresponding event line. When a data race is detected, either tool adds a line to the log file: `Data race at: source file:line number.` We compare the file sizes with and without compression by using the Linux Gzip compressor.

We also compared FF with LOFT by using different numbers of threads on the benchmarks. We systematically varied the number of threads: 2, 4, 8, 16, 32, and

TABLE II. Size of Log Files on C/C++ benchmarks.

Benchmarks	Application Domain	Application Classification	Log size (bytes)			Compressed by gzip (bytes)		
			FF_{race} (E)	$LOFT_{race}$ (F)	(F) ÷ (E)	FF_{race} (G)	$LOFT_{race}$ (H)	(H) ÷ (G)
facesim	Animation	Computer Graphics	2,030,782.8	1,228,941.5	0.61	35,976.6	20,395.9	0.57
raytrace	Rendering		12,525.9	9,731.6	0.78	523.4	512.6	0.98
bodytrack	Computer Vision		273,703.7	239,662.1	0.88	4,526.4	4,236.6	0.94
swaptions	Financial Analysis	Compute-bound	2,040.0	170.0	0.08	126.0	111.0	0.88
blackscholes	Financial Analysis		170.0	85.0	0.50	101.0	97.0	0.96
canneal	Engineering	Data-centric	2,712.0	672.0	0.25	169.4	155.0	0.92
streamcluster	Data Mining		31,203,402.4	11,080,455.5	0.36	138,847.4	113,956.7	0.82
MySQL	Database		16,686,778.0	11,302,350.1	0.68	84,853.8	61,977.7	0.73
vips	Media Processing		487,771.9	386,395.0	0.79	5,594.4	4,638.6	0.83
dedup	Enterprise Storage		782,809.8	756,066.8	0.97	63,791.2	63,613.0	1.00
x264	Media Processing		75,403.9	73,789.6	0.98	1,407.1	1,325.1	0.94
Httpd	Web server	Web Application	244,246.3	49,159.1	0.20	3,554.8	1,828.0	0.51
Chromium	Web browser		24,244,261.5	8,512,124.2	0.35	287,403.4	125,909.1	0.44
Firefox	Web browser		339,161,513.7	16,423,195.9	0.05	2,461,324.2	152,920.7	0.06
Thunderbird	E-mail client		83,558,698.4	14,745,031.1	0.18	645,895.1	197,800.7	0.31
Total	-		498,766,820.30	64,807,829.50	0.130	3,734,094.20	749,477.70	0.201
Mean	-		-	-	0.511	-	-	0.726

64, and re-ran the test harnesses of the benchmarks in each case by 100 times each. The subjects `vips`, `dedup`, `Httpd`, `Chromium`, `Firefox`, and `MySQL`, and `Thunderbird` either used a pre-configured number of worker threads or we were unable to modify this setting. Hence, we did not report the result of these seven subjects in the scalability experiment. On `dedup`, when the number of thread is 32 or more, it crashed under Pin even without applying our tools. Hence, we did not report the result on this subject as well.

As we have stated in the implementation paragraph, compared to FF, LOFT needs to maintain one more variable for each thread and one more variable for each lock. However, the number of threads and locks in a typical program execution are often limited. Take `Httpd` for example. The number of locks encountered in the subjects is only 21. The number of threads in a program execution (as we will study in the next section) is also a small integer. Because the addition of each of the two variables only means an extra space of one integer, we anticipate that the extra space needed for LOFT is insignificant. As such, we skipped the experiment that measured the memory footprints consumed by LOFT and FF.

5.4 Data Analysis

TABLE I summarizes the results of the experiment. The second and the third columns (`Application Domain` and `Size`) report the application domain ([1],[9],[4]) and the lines of code for each benchmark, respectively. The column "`# of worker threads`" shows the number of threads used in the experiment. The column "`Vector operations`" shows the number of vector clock operations performed by FF and LOFT, as well as the ratio of LOFT to FF in the column "`(B) ÷ (A)`". The column "`Time`" shows the corresponding time needed to complete all such tracking in microsecond (µs) for FF and LOFT, as well as the ratio of LOFT to FF in the column "`(D) ÷ (C)`". We note that the reported time for LOFT has included the time overhead to maintain the **`lastThread`** and **`lastLock`** conditions.

Dimension 1 (*The Amount of Removable Operations*): From TABLE I, we observe that LOFT, on average, identifies 0.639 of all the vector clock operations that are needed by FF for lock acquisition or release as removable. Specifically, compared to FF, on `swaptions`, `Httpd`, and `Firefox`, LOFT identifies 0.96, 0.97, and 0.92 of all vector clock operations, respectively, as removable. If we consider the total amount of operations that can be removed from the entire suite of benchmarks, LOFT identifies 0.869 of all such operations as removable. This experimental result is consistent with our conjecture that *a lock is often acquired and released consecutively by the same thread in an execution*. In other words, in theory, one can find that the conjecture may not hold; but in practice, our findings show that this phenomenon popularly occurs in executions in programs that can be represented by this experiment.

TABLE II shows the sizes of the lock trace (log) files generated by FF_{race} and $LOFT_{race}$, without ("`Log size`") and with ("`Compressed by gzip`") applying compression on the generated lock trace files.

From TABLE II, we observe that, on average, $LOFT_{race}$ reduces up to 0.489 of the log files generated by FF_{race}. On `swaptions` and `Firefox`, we find that most of the lock acquisitions and releases were only performed by the main thread; therefore, the size of the log files generated by $LOFT_{race}$ was much smaller than that generated by FF_{race}. After compression, the mean size of the log files genreated by $LOFT_{race}$ is 0.274 smaller than that generated by FF_{race}.

We also classified all benchmarks into four set according to their appliction types as shown in the third column ("`Application clssfication`") in TABLE II. The four sets are: `Computer Graphics`, `Compute-bound programs`, `Data-centric programs`, and `Web Applications`. We find from column "`(F)÷(E)`" that the lock trace reduction ratio for the benchmarks in the `Web Application` category is 0.65−0.95. For the compute-bound benchmarks, the corresponding ratio is 0.50−0.92. Such saving is significant. For the other two categories, the reduction ratios vary from 0.02 to 0.75.

Noticeably, for the four programs with millions line of code (`MySQL`, `Chromium`, `Firefox`, and `Thunderbird`), the reduction ratios are 0.32−0.95 without compression and 0.27−0.94 with compression.

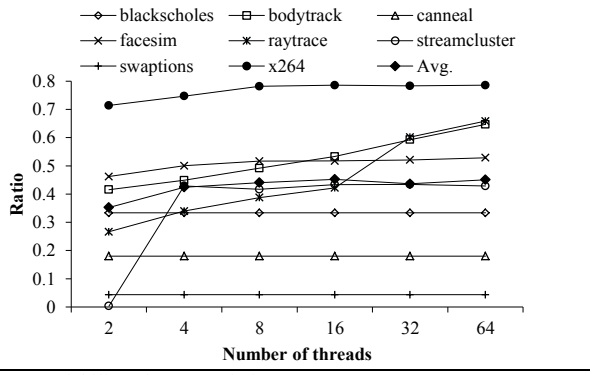

Figure 7. Scalability of LOFT relative to FF.

Appendix D shows supplementary results.

Dimension 2 (*Scalability of* LOFT): Figure 7 shows the ratio (*x*-axis) on the number of vector operations retained in a log trace generated by LOFT to that of FF at different numbers of threads (*y*-axis) on each benchmark that we are able to configure the number of threads used for the benchmark. From Figure 7, we observe that the ratio is fairly stable across different numbers of threads on the same benchmark, except on `bodytrack` and `raytrace`. On average, in the experiment, LOFT can remove 0.55 of all such operations in the lock traces. From Figure 7, on `bodytrack` and `raytrace`, the curves move up with an increasing number of threads. On the two *Computer-Graphics* benchmarks, we found that the interleaving among the threads becomes more complex and lock contentions among worker threads are increasingly noticeable.

Dimension 3 *(Online Tracking Time)*: From the `Time` column in TABLE I, we find that, on average, LOFT ran 16.9% faster than FF on tracking all the critical events. The results were consistent across all the benchmarks. On the three large-scale real-life programs, LOFT runs 18−20% faster than FF. The result indicated that the overheads of LOFT can be well-compensated by the amount of reduced removable operations. Appendix E shows the corresponding hypothesis testing result that confirm the difference is statistically meaningful.

Dimension 4 *(Precision on Concurrency Bug Detections)*: Although we have analyzed in Section 4 that LOFT does not compromise the tracking of critical events (and hence, it does not affect the precision of the associated concurrency bug detections), we still report the results here as a second-line validation of LOFT. We find that FF_{race} and $LOFT_{race}$ reported the same number of data races on all subjects we used. The numbers of races detected by either technique for the subjects listed in TABLE I, from top to bottom, are 0, 13, 5, 0, 5, 0, 29, 12, 0, 0, 26, 12, 134, 2, and 3, respectively.

Appendix C shows a further evaluation on a suite of Java benchmarks. The results on Java benchmarks are consistent with the results reported above.

6. RELATED WORK

Tracking of Happened-before Relations: Online dynamic concurrency bug detectors (e.g., [10],[16],[21],[27],[30]) often use vector clocks to track the happened-before relations among the monitored events. The empirical results of *FastTrack* [16] show that most of such events are data access events, and *FastTrack* optimizes the bug detection algorithm for such data access events. If the amount of data access events can be reduced such as by sampling some but not all data access events, the incurred overhead can be reduced, but then the amount of synchronization events emerges to become the bottleneck [10].

Our approach can be regarded as a strategy toward addressing this bottleneck by identifying two kinds of synchronization events (as quantified as Case 1, Case 3, and Case 5 in Section 4) not to be tracked in full. We have compared to *FastTrack* extensively in this experiment. *LiteRace* [28] is another sampling strategy, that it consists of an online monitoring phase and an offline tracking phase to detect concurrency bugs. *RaceZ* [33] is also a sampling strategy on data race detection; however, it is based on lockset and collects memory accesses through hardware rather than through software instrumentation. *Carisma* [38] can effectively detect race conditions when the sampling rate is very low. They all fully track the synchronizations among lock and thread events. Our technique is applicable during their synchronization collection phases, and complements their techniques as well as in other fields (i.e., atomicity violation detection [23] and execution replay [7]) that also employ the happened-before relation to track synchronization events.

FastTrack [16] introduces the concept of epoch for the tracking of memory accesses. Their idea cannot be applied to track the happened-before relations among synchronization events because the successful application of epoch relies on the insight that the write operations of the same memory location among threads in an execution forms a total order; and yet for lock acquisition and release events among threads, no such total order can be assumed.

Event Filtering: 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 detectors (e.g., [30],[31],[37]). Such a strategy reduces the slowdown overhead without compromising the precision of such detectors. Another type of filter is to remove irrelevant lock dependency generated from a log trace, which in [13], we showed that such a strategy can significantly improve the performance of a detector in an empirical study. LOFT can also be regarded as an event filter. It complements existing work by filtering removable synchronization events.

Log Reduction: There is also a large body of log reduction techniques. *Checkpointing* [14] records part of the execution logs such that these logs are adequate to replay the given program with aim to reproduce bugs. Lee et al. [24] use the rich runtime information to reduce the log size. It firstly instruments the given program to record the selective events (e.g., *read* and *write*) in forms of units (e.g., for-loops) during runtime. Then, it analyzes the recorded logs offline and eliminates the

event that has no dependency relation to the next event. However, these reduction techniques are only applicable for sequential programs [24]. LOFT is applicable to reduce traces of multithreaded programs. Tallam et al. [34] proposed a demand-driven approach to log reduction. For a given bug signature, their technique identifies the threads and execution regions that are irrelevant to the bug and removes them. LOFT is generic and does not rely on any bug signature. *SimTrace* [20] is an offline and static technique to simplify an execution trace by merging two events if they are performed by the same thread and no any other event occurs in between them through trace equivalence. LOFT is a dynamic and on-the-fly technique.

7. Conclusion

In this paper, we have proposed an approach entitled LOFT to substitute removable operations in the online tracking of the happened-before relations of synchronization events by thread-local timestamp updating. We have analyzed the scenarios of consecutive thread-centric lock operations and have identified a sound condition that a technique can rely on it to safely remove the involved vector clock comparisons and content copy of vector clock in the tracking of such relationships among such events. We have conducted an experiment to validate LOFT. In the experiment, using FF as the baseline, LOFT has identified, on average 63.9% of all the vector comparisons and updates as removable, reduced 87.0% in terms of the lock trace size, and run 16.9% faster in completing such tracking.

Our work only answers an aspect of log reduction. As discussed in Section I, the potential of duplicated subsequences in log traces have been explored by a large body of regression testing techniques. It is interesting to integrate the log traces generated by our approach with such techniques.

Acknowledgement

This work is supported in part by the General Research Fund of the Research Grant Council of Hong Kong (project nos. 111410 and 716612).

References

[1] Apache Httpd 2.3.8, *http://httpd.apache.org*.
[2] Chromium 18.0.1025.151, *http:// www.chromium.org*.
[3] Firefox 14.0.1, *http://www.mozilla.org/firefox*.
[4] MySQL 5.0.92, *http://www.mysql.com.*
[5] Thunderbird 14.0, *http://www.mozilla.org/thunderbird*.
[6] R. Agarwal, L. Wang, and S.D. Stoller, Detecting potential deadlocks with static analysis and run-time monitoring. In Hardware and Software, Verification and Testing, LNCS 3875/2006, 191–207, 2006.
[7] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In Proc. *SOSP*, 193–206, 2009.
[8] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drini ć, D. Miho ˇcka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In Proc. *VEE*, 154–163, 2006.
[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In Proc. *PACT*, 72–81, 2008.
[10] M. D. Bond, K. E. Coons and K. S. Mckinley. PACER: Proportional detection of data races. In Proc. *PLDI*, 255–268, 2010.
[11] N. Barrow-Williams, C. Fensch and S. Moore. A Communication characterization of SPLASH-2 and PARSEC. In Proc. *IISWC*, 86–97, 2009.
[12] Y. Cai and W.K. Chan. LOFT: Redundant synchronization event removal for data race Detection. in Proc. *ISSRE*, 160–169, 2011.
[13] Y. Cai and W.K. Chan. MagicFuzzer: Scalable deadlock detection for large-scale applications. In Proc. *ICSE*, 606–616, 2012.
[14] K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, 1985.
[15] G. Contreras and M. Martonosi. Characterizing and improving the performance of Intel threading building blocks. In Proc. *PACT*, 57–66, 2008.
[16] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In Proc. *PLDI*, 121–133, 2009.
[17] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In Proc. *PASTE*, 1–8, 2010.
[18] C. Flanagan, S.N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In Proc. *PLDI*, 293–303, 2008.
[19] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX Annual Technical Conference, General Track*, 289–300, 2006.
[20] J. Huang and C. Zhang. An efficient static trace simplification technique for debugging concurrent programs. In Proc. *SAS*, 163–179, 2011.
[21] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy. Helgrind+: an efficient dynamic race detector. In Proc. *IPDPS*, 1–13, 2009.
[22] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, Deadlock immunity: enabling systems to defend against deadlocks. In Proc. *OSDI*, 295–308, 2008.
[23] Z.F. Lai, S.C. Cheung, and W.K. Chan, Detecting atomic-set serializability violations for concurrent programs through active randomized testing. In Proc. *ICSE*, 235–244, 2010.
[24] K.H. Lee, Y.H. Zheng, N. Sumner, and X.Y Zhang, Toward generating reducible replay logs. In Proc. *PLDI*, 246–257, 2011.
[25] S. Lu, S. Park, E. Seo, and Y.Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Proc. *ASPLOS*, 329–339, 2008.
[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Proc. *PLDI*, 191–200, 2005.
[27] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7):558–565, 1978.
[28] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. in Proc. *PLDI*, 134–143, 2009.
[29] F. Mattern. Vitual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithm*, 215-226. 1988.
[30] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In Proc. *PPoPP*, 179–190, 2003.
[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM TOCS*, 1997, 15(4): 391–411.
[32] K. Serebryany and T. Iskhodzhanov. Threadsanitizer – data race detection in practice. In Proc. *WBIA*, 62–71, 2009.
[33] T.W. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W.G. Chen, and W.M. Zheng. RACEZ: A lightweight and non-invasive race detection tool for production applications. In Proc. *ICSE*, 401–410, 2011.
[34] S. Tallam, C. Tian, R. Gupta, and X.Y. Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In Proc. *ISSTA*, 207–218, 2007.
[35] X.W. Xie and J.L. Xue. ACCULOCK: Accurate and efficient detection of data races. In Proc. *CGO*, 201–212, 2011.
[36] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, to appear. DOI: 10.1002/stvr.430.
[37] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In Proc. *SOSP*, 221–234, 2005.
[38] K. Zhai, B.N. Xu, W.K. Chan, and T.H. Tse. CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In Proc. *ISSTA*, 221–231, 2012.

Yan Cai is a PhD student at Department of Computer Science, City University of Hong Kong. He received his BEng degree in Computer Science and Technology from Shandong University, China, in 2009. His research interest is concurrency bug detection and reproduction in multithreaded programs and distributed systems.

W.K. Chan is an assistant professor at Department of Computer Science, City University of Hong Kong. He is on the editorial board of *Journal of Systems and Software*. He served as guest co-editors of a number of international software engineering journals, program chairs of AST 2010 and QSIC 2010, and innovative showcase chairs of ICWS and SCC for both 2009 and 2010. His research results have been extensively reported in top-notch venues such as TOSEM, TSE, CACM, COMPUTER, ICSE, FSE, ISSTA, ASE, WWW, and ICDCS. His current research interest includes program analysis and testing for large-scale software systems.

Supplementary File for the TPDS Manuscript:

Lock Trace Reduction for Multithreaded Programs

Yan Cai and W.K. Chan

APPENDICES

A. Correctness Proof

In this section, we extend Section 4.1 with the correctness proof for the cases where the vector clock updates can be eliminated. To distinguish the different vector clock content at different events, we use the symbol V^e to denote the vector clock content when the event e occurs. For example, on event e_a = *acquire*(t, m), we use $C_t^{e_a}$ to denote the vector clock of the thread t and $L_m^{e_a}$ to denote the vector clock of the lock m.

We firstly analyze the scenarios in which e_* is an **acquire**(t, m). There are two cases: Case 1 and Case 2.

Case 1. [if `lastThread`(m) = t].

Let e_1 be an event in a trace denoting that t releases m such that `lastThread`(m) = t, and e_* be an event in the same trace that t acquires m after the occurrence of e_1.

Consider the trace $\langle ... e_1 ... e_* ... \rangle$, when e_1 occurs, according to *FF* (at lines 3 and 4 in Figure 2), we must have:

$$L_m^{e_1}[i] = C_t^{e_1}[i] \ (1 \le i \le n, i \ne t), \text{ and}$$
$$L_m^{e_1}[i] < C_t^{e_1}[i] \ (i = t) \quad (1)$$

When e_* occurs, because `lastThread`(m) = t holds, L_m is not changed by any other thread. So, we have:

$$L_m^{e_*} = L_m^{e_1} \quad (2)$$

However, the values in C_t may be incremented because t may acquire some other lock(s) in between e_1 and e_* (as illustrated by the "***" symbols in `Case 1` of Figure 4); otherwise, C_t must remain unchanged. No matter C_t is incremented or not, we have

$$C_t^{e_1} \sqsubseteq C_t^{e_*} \quad (3)$$

From Eq. (1), (2), and (3), when e_* occurs, we have $L_m^{e_*} \sqsubseteq C_t^{e_*}$. Therefore, there is no need to perform any comparison between L_m and C_t, and the corresponding comparison can be removed when e_* occurs. Moreover, we need not to update the timestamp at $C_t^{e_1}[t]$ on this *acquire* event according to *FF* (at line 5 in Figure 2).

Case 2. [if `lastThread`(m) ≠ t].

Let e_* be an event that t acquires m.

Consider the trace $\langle ... e_* ... \rangle$, when e_* occurs, because we have `lastThread`(m) ≠ t, there are two sub-cases to consider: m must either have been released by a thread t' (where $t' \ne t$) 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, ⊔ L_m. In the latter case, the value of L_m must be different from that of C_t because all locks are initialized as all 0s, whereas all threads are initialized as all 1s (see lines 1 and 2 of Figure 2). Therefore, without further checking, we cannot determine a definite happened-before relation between L_m and C_t. In this situation, when e_* occurs, such a comparison and its associated potential assignment from $C_t \sqcup L_m$ to C_t are necessary. Hence such operations are not removable (which is depicted by the second (dotted) arrow in `Case 2` of Figure 4).

We now analyze scenarios in which e_* is a **release**(t, m). There are four cases: Cases 3−6.

Case 3. [if `lastThread`(m) = t and `lastLock`(t) = m].

Let e_1 be an event in a trace that t releases m such that `lastThread`(m) = t and `lastLock`(t) = m, e_* be an event that t releases m, and e_2 be the corresponding lock acquisition operation performed by t with respect to e_*.

Consider the trace $\langle ... e_1 ... e_2 ... e_* ... \rangle$, when e_2 occurs, because `lastThread`(m) = t held, the situation is the same as that in `Case 1`. According to `Case 1`, we have:

$$L_m^{e_1}[i] = C_t^{e_1}[i] \ (1 \le i \le n, i \ne t), \text{ and}$$
$$L_m^{e_1}[i] < C_t^{e_1}[i] \ (i = t) \quad (4)$$

and

$$L_m^{e_2} = L_m^{e_1} \quad (5)$$

In between the occurrences of e_1 and e_2, because of the condition `lastLock`(t) = m, we have:

$$C_t^{e_1} = C_t^{e_2} \quad (6)$$

(Note that Eq. (6) is different from Eq. (3) for `Case 1` because, here, we know that t neither acquires nor release any other lock in between e_1 and e_2 due to the condition `lastLock`(t) = m.) Therefore, we have:

$$C_t^{e_*} = C_t^{e_2} \quad (7)$$

Besides, in between the occurrences of e_2 and e_*, m is held by t and has not been released by the latter. So, L_m must be kept unchanged. Therefore, we have:

$$L_m^{e_*} = L_m^{e_2} \quad (8)$$

From Eq. (4), (5), (6), (7), and (8), we obtain the following:

$$L_m^{e_*}[i] = C_t^{e_*}[i] \ (1 \le i \le n, i \ne t) \text{ and}$$
$$L_m^{e_*}[i] < C_t^{e_*}[i] \ (i = t).$$

Therefore, to make $L_m = C_t$ true when e_* occurs, we only need to update $L_m[t]$ to be $C_t[t]$; and the removable vector clock comparison between C_t and L_m can be eliminated.

Case 4. [if `lastThread`(m) = t and `lastLock`(t) ≠ m].

Let e_1 be an event that t releases m such that `lastThread`(m) = t, e_* be an event that t releases m, and e_2 be the corresponding *acquire* operation of e_*.

Consider the trace $\langle ... e_1 ... e_2 ... e_* ... \rangle$, when e_1 occurs, because `lastThread`(m) = t holds, the situation is the same as that in `Case 1`, we have:

$$L_m^{e_1}[i] = C_t^{e_1}[i] \ (1 \le i \le n, i \ne t) \quad (9)$$

$L_m^{e_1}[i] < C_t^{e_1}[i]$ $(i = t)$

When e_2 occurs, because of the condition `lastLock`$(t) \neq m$, C_t may have been adjusted due to acquiring and releasing some other lock. So, we only get:

$$C_t^{e_1} \sqsubseteq C_t^{e_2} \quad (10)$$

When e_* occurs, the condition `lastThread`$(m) = t$ implies that m has not been acquired and released by any other thread in between e_1 and e_*. Hence, we have:

$$L_m^{e_*} = L_m^{e_2} = L_m^{e_1} \quad (11)$$

In between the occurrences of e_2 and e_*, t has not acquired and released any other lock, we have:

$$C_t^{e_*} = C_t^{e_2} \quad (12)$$

From Eq. (9), (10), (11), and (12), we can only infer that $L_m^{e_*} \sqsubseteq C_t^{e_2} = C_t^{e_*}$ holds. Therefore, to ensure the condition $L_m = C_t$ to be true, the vector clock assignment from C_t to L_m is not removable.

Case 5. [if `lastThread`$(m) \neq t$ and `lastLock`$(t) = m$].

Let e_1 be an event that t releases m such that `lastLock`$(t) = m$, e_* be an event that t releases m, and e_2 be the corresponding lock *acquire* operation of e_*.

Consider the trace $\langle \dots e_1 \dots e_2 \dots e_* \dots \rangle$, on e_1, according to *FF* at line 4 in Figure 2, we have:

$$L_m^{e_1}[i] = C_t^{e_1}[i] \ (1 \leq i \leq n, i \neq t)$$
$$L_m^{e_1}[i] < C_t^{e_1}[i] \ (i = t) \quad (13)$$

From `lastThread`$(m) \neq t$, we know that, between e_1 and e_2, there must exist at least one pair of lock acquisition and release events on m performed by some other thread s such that $s \neq t$. To ease our proof, we denote the last pair of such lock acquisition and release events by s as e_{*a} and e_{*r}, respectively.

From e_1 to e_2, because held `lastLock`$(t) = m$, C_t must keep unchanged and L_m must have been incremented. Therefore, we have:

$$C_t^{e_{*r}} = C_t^{e_{*a}} = C_t^{e_1}, \text{ and}$$
$$L_m^{e_1} \sqsubseteq L_m^{e_{*a}} \sqsubseteq L_m^{e_{*r}} \quad (14)$$

When e_2 occurs, because `lastThread`$(m) \neq t$ holds, which is the same as `Case 2`. We have

$$C_t^{e_2}[i] = \max\{C_t^{e_{*r}}[i], L_m^{e_{*r}}[i]\}, (1 \leq i \leq n) \quad (15)$$

In between the occurrences of e_2 and e_*, L_m cannot be changed because no *release* event of m occurs. Hence, we have:

$$L_m^{e_2} = L_m^{e_{*r}} \quad (16)$$

Then, from Eq. (13), (14), (15), and (16), we infer that

$$C_t^{e_2}[i] = L_m^{e_2}[i] \ (1 \leq i \leq n, i \neq t) \quad (17)$$

In between the occurrences of e_2 and e_*, both vector clocks of m and t cannot be changed, which implies:

$$C_t^{e_*} = C_t^{e_2} \text{ and}$$
$$L_m^{e_*} = L_m^{e_2} \quad (18)$$

So, when e_* occurs, from Eq. (17) and (18), we have:

$C_t^{e_*}[i] = L_m^{e_*}[i]$ $(1 \leq i \leq n, I \neq t)$ and

$C_t^{e_*}[i] < L_m^{e_*}[i]$ $(i = t)$.

Hence, to make sure $L_m = C_t$ true when e_* occurs, we only need to update $L_m[t]$ to be $C_t[t]$ and the removable vector clock comparison between C_t and L_m can be eliminated.

Case 6. [if `lastThread`$(m) \neq t$ and `lastLock`$(t) \neq m$].

In this case, we know nothing about the relation between L_m and C_t. Therefore, no vector-based comparison or assignment is removed.

B. Implementation

We have shown by a theoretical analysis in Section 4 that our algorithm can identify removable operations, and update the set of vector clocks for threads and locks to achieve an equivalent result of *FF* in tracking critical events. We have further implemented the data race tracking algorithm (*FastTrack*) presented in [16] to LOFT. The detail is as follows. We note that both *FF* and LOFT have no need to track memory accesses.

Memory Shadow: To track memory accesses, we adopted a two level shadow implementation `M0` as described in [41]. We also used a *Copy-on-Write* strategy to start to shadow a memory location on the first access to it which is either a *read* or a *write* so that the allocated but non-accessed memory was not shadowed, which could save the memory consumption and the time needed to shadow those non-accessed memory. For each thread, because Pin supplies a thread-local storage (TLS) per thread [26], we used this TLS to store a data set (e.g., thread vector clock) for each thread.

TABLE III. Comparisons on effectiveness and its efficiency of LOFT and FF on Java benchmarks.

Benchmarks	Application Domain	Size (lines of code)	# of worker threads	# of Vector operations			Time (μs)		
				FF (K)	LOFT (L)	(L) ÷ (K)	FF (M)	LOFT (N)	(N) ÷ (M)
crypt	IDEA Encryption	1,191	7	12.0	2.0	0.17	80.1	35.2	0.44
lufact	LU Factorisation	1,580	4	12.0	4.0	0.33	89.3	37.6	0.42
moldyn	Molecular Dynamics Simulation	1,351	4	22.0	2.0	0.09	113.8	58.5	0.51
montecarlo	Monte Carlo Simulation	3,630	4	22.0	2.0	0.09	108.6	51.3	0.47
series	Fourier Coefficient Analysis	919	4	12.0	2.0	0.17	79.5	34.1	0.43
sor	Successive Over-relaxation	828	4	12.0	2.0	0.17	82.6	35.0	0.42
sparse	Sparse Matrix Multiplication	820	4	12.0	2.0	0.17	108.0	38.1	0.35
tsp	Traveling Saleman Problem	718	5	45.0	36.5	0.81	213.3	166.3	0.78
raja	Ray Tracer	10,516	9	17.1	5.0	0.29	107.1	48.1	0.45
raytracer	3D Ray Tracer	1,938	4	48.5	11.0	0.23	227.9	139.7	0.61
elevator	Discrete Events Simulator	550	4	5,174.4	1,962.3	0.38	14,151.2	11,188.2	0.79
philo	Dinning Philosophers Simulator	116	6	24.0	4.1	0.17	153.2	59.4	0.39
Total	-	24,157	-	5,413.0	2,034.9	0.376	15,514.6	11,891.4	0.766
Mean	-	-	-	-	-	0.255	-	-	0.506

TABLE IV. Size of Log Files on Java benchmarks.

Benchmarks	Application Domain	Application Classification	Log size (bytes)			Compressed by gzip (bytes)		
			FF_{race} (O)	$LOFT_{race}$ (P)	(P) ÷ (O)	FF_{race} (Q)	$LOFT_{race}$ (R)	(R) ÷ (Q)
crypt	IDEA Encryption	Compute-bound	324.0	54.0	0.17	79.0	76.0	0.96
lufact	LU Factorisation		324.0	108.0	0.33	85.0	84.0	0.99
moldyn	Molecular Dynamics Simulation		594.0	54.0	0.09	82.0	77.0	0.94
montecarlo	Monte Carlo Simulation		594.0	54.0	0.09	86.0	81.0	0.94
series	Fourier Coefficient Analysis		324.0	54.0	0.17	80.0	77.0	0.96
sor	Successive Over-relaxation		324.0	54.0	0.17	77.0	74.0	0.96
sparse	Sparse Matrix Multiplication		324.0	54.0	0.17	87.0	84.0	0.97
tsp	Traveling Saleman Problem		1,215.0	991.7	0.82	146.1	142.0	0.97
raja	Ray Tracer	Computer Graphics	459.5	135.3	0.29	95.4	88.4	0.93
raytracer	3D Ray Tracer		1,404.0	297.0	0.21	128.8	120.7	0.94
elevator	Discrete Events Simulator	Simulation	137,577.7	52,413.2	0.38	1,210.1	505.7	0.42
philo	Dinning Philosophers Simulator		648.0	110.2	0.17	87.7	82.1	0.94
Total	-		144,112.2	54,379.4	0.377	2,244.2	1,492.1	0.665
Mean	-		-	-	0.255	-	-	0.909

For each lock and each barrier, we used an unordered map supplied by the GCC compiler to map the lock or the barrier to a set of data (e.g., a vector clock).

C. Further Evaluation on Java Benchmarks

In Section 5, we have evaluated LOFT on a suite of C/C++ multithreaded programs. In this section, we further evaluated LOFT on a set of multithreaded Java benchmarks.

We firstly downloaded the RoadRunner framework [17] that included the latest implementation of FastTrack [16] written by the authors of FastTrack. We then implemented LOFT[2] on top of this FastTrack implementation on the downloaded RoadRunner. We ran the benchmarks for both LOFT and FF as well as FF_{race} and $LOFT_{race}$ in the same machine as reported in Section 5.

We selected 12 benchmarks including `crypt`, `lufact`, `moldyn`, `montecarlo`, `series`, `sor`, `sparse`, `tsp`, `raja`, `raytracer`, `elevator`, and `philo`. All these benchmarks are from [16] that have been used to evaluate FF_{race}. We did not use the other benchmarks such as `jbb` reported in [16] due to our financial budget issue as they were paid benchmarks. We did not include `eclipse` because we found that the downloaded RoadRunner framework fails to run correctly when monitoring `eclipse` even without any changes we made to implement LOFT on it.

TABLE III shows the average number of vector operations performed and time taken to complete such tracking on each Java benchmark. The columns of TABLE III carry the same meanings as these of TABLE I. From TABLE III, we find the following: (1) on average, LOFT identifies 74.5% of vector clock operations needed by *FF* are removable; (2) for the online tracking time, on average, LOFT ran 49.4% faster than *FF*. These results are similar to our findings on the C/C++ benchmarks reported in Section 5 of this paper.

TABLE IV shows the size of trace log on the Java benchmarks both before and after compression. The columns of TABLE IV carry the same meanings as these of TABLE II. From TABLE IV, we observe that $LOFT_{race}$ reduces up to 74.5% of log files generated by FF_{race}. After compression, on average, a reduced log file is 9.1% smaller than that generated by FF_{race}.

The numbers of reported data races on these Java benchmarks by FF_{race} and $LOFT_{race}$ are the same, which are 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, and 0 for the benchmarks in TABLE III from top to down, respectively. Note that two of these numbers (on `montecarlo` and `raja`) are different from the original data reported in [16]. This is because we used the latest version of FF_{race} and the original implementation used in [16] has an implementation bug as reported in [40] by the same authors of the FastTrack.

D. Study on Generated Lock Trace

To understand the generated log trace better, we further study the generated lock trace files. TABLE VI shows the corresponding parts of the log files on the benchmark `raytrace` in the first column and the second column generated by FF_{race} and $LOFT_{race}$, respectively. On `raytrace`, `thread 0` consecutively acquired and released a lock *m* for 22 times; and then it acquired and released a lock *n* once, followed by acquiring and releasing the lock *m* again for another 5 times. At this moment, `thread 1` acquired and released a lock *k*, and a data race was detected. Therefore, FF_{race} generated one log event line for each *acquire* or *release* event, as shown in the first column. In total, there were 60 event lines before the data race is reported. $LOFT_{race}$ was able to identify the consecutive *acquire* and *release* of the same lock through **`lastThread`**, **`lastLock`**, as well as **`acqCounter`**. It avoided generating the corresponding log event lines incurred by FF_{race}. As shown in the second column of TABLE VI, LOFT merely generated 8

[2] Our LOFT is available at: www.cs.cityu.edu.hk/~51948163/loft/ .

TABLE VI. Part of Log Files on `raytrace` from C/C++ benchmarks.

Log file generated by FF_{race}		Log files generated by $LOFT_{race}$	
1	Thread 0 [`acquires`] a lock (*m*)	1	Thread 0 [`acquires`] a lock (*m*)
2	Thread 0 [`releases`] a lock (*m*)	2	Thread 0 [`releases`] a lock (*m*)
…	… (here above pair (lines 1 and 2) repeated for 21 additional times)		
45	Thread 0 [`acquires`] a lock (*n*)	3	Thread 0 [`acquires`] a lock (*n*)
46	Thread 0 [`releases`] a lock (*n*)	4	Thread 0 [`releases`] a lock (*n*)
47	Thread 0 [`acquires`] a lock (*m*)	5	Thread 0 [`acquires`] a lock (*m*)
48	Thread 0 [`releases`] a lock (*m*)	6	Thread 0 [`releases`] a lock (*m*)
…	… (here above pair (lines 47 and 48) repeated for 5 additional times)		
59	Thread 1 [`acquires`] a lock (*k*)	7	Thread 1 [`acquires`] a lock (*k*)
60	Thread 1 [`releases`] a lock (*k*)	8	Thread 1 [`releases`] a lock (*k*)
61	Data race at: *RTTL/common/RTThread.hxx:19*.	9	Data race at: *RTTL/common/RTThread.hxx:19*.

event lines for the event trace of `raytrace` before reporting the same data race as FF_{race} did. It not only results in a log file that is only 15% of the original log file size, but also can be more comprehensible for programmers to study the reported race problem.

Indeed, from the generated log files, we found that during the startup period of a program in the benchmark suites, the main thread popularly and consecutively acquired and released the same lock. For example, as shown in TABLE VI, on `raytrace`, the main thread consecutively acquired and released a same lock for 22 times before acquired and released a second lock, and then acquired and released the first lock consecutively for another 5 times. It is interesting to study on why programs were developed as such.

E. Mann-Whitney U Test Result on Tracking Time

To further compare the online tracking time, we also computed the `Mann-Whitney U Test` result using the Matlab `ranksum` tool [39] on the dataset that composed the timing statistics presented in TABLE I and TABLE IV. The result is presented in TABLE V, which shows that LOFT and *FF* are different significantly at the 0.001 significance level across all the benchmarks.

TABLE V. Mann-Whitney U Test Result on Tracking Time

Benchmarks	Mann-Whiney U Test Result	Benchmarks	Mann-Whiney U Test Result
C/C++		Java	
facesim	< 0.00001	crypt	< 0.00001
raytrace	< 0.00001	lufact	< 0.00001
bodytrack	< 0.00001	moldyn	< 0.00001
swaptions	< 0.00001	montecarlo	< 0.00001
blackscholes	0.00062	series	< 0.00001
canneal	< 0.00001	sor	< 0.00001
streamcluster	< 0.00001	sparse	< 0.00001
MySQL	< 0.00001	tsp	< 0.00001
vips	< 0.00001	raja	< 0.00001
dedup	< 0.00001	raytracer	< 0.00001
x264	< 0.00001	elevator	< 0.00001
Httpd	< 0.00001	philo	< 0.00001
Chromium	< 0.00001		
Firefox	< 0.00001		
Thunderbird	< 0.00001		

F. Threats to Validity

The C/C++ version of our LOFT and *FF* were implemented by us and the Java versions are based on the original *FF* implementation [16]. We have assured our implementations on several small multithreaded C/C++ and Java programs. Different implementations may have different effects on the performance and the collected data. All benchmarks we used (including both C/C++ and Java benchmarks) are widely studied in previous experiments. Using other set of benchmarks may produce different results. Besides, profiling an execution may affect the original program execution. However, since both LOFT and *FF* only analyze the synchronizations; and the profiling is much less intrusive than profiling memory accesses. We compared LOFT and *FF* by using the following three metrics: the number of vector operations on synchronizations, the size of generated log files, and the time taken to track all these synchronizations. Using different metrics may have different results.

REFERENCE

[39] Matlab 7.8.0 (R2009a), at *http://www.mathworks.com*.

[40] C. Flanagan and S.N. Freund. Adversarial memory for detecting destructive races. In Proc. *PLDI*, 244–254, 2010.

[41] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In Proc. *VEE*, 65–74, 2007.