HistLock+: Precise Memory Access Maintenance Without Lockset Comparison for Complete Hybrid Data Race Detection

Jialin Yang, Bo Jiang, *Member, IEEE*, and W. K. Chan, *Member, IEEE*

***Abstract*—Dynamic hybrid data race detectors alleviate the detection imprecision problem incurred by pure lockset-based race detectors and the thread interleaving sensitive problem incurred by pure happens-before race detectors. Nonetheless, to ensure at least one data race on every memory location to be detected, keeping all historical memory access events in the analysis state of such a detector is impractical. Existing complete hybrid race detectors perform extensive comparisons among the locksets of the memory accesses on each memory location to identify which of them to be retained in its analysis state, which incurs significant runtime overhead. In this paper, we investigate to what extent a complete hybrid data race detector able to perform such identifications without lockset comparison. We present HistLock+, which is built atop thread epoch and lock release events to infer whether two memory accesses on the same memory location from the same thread in between consecutive lock release operations have any lock subset relation without performing expensive lockset comparison. HistLock+ guarantees exactly one racy memory access event to be reported on each thread segment separated by lock releases and hard-order thread synchronizations, and it never reports false positive on lockset violation. We have validated HistLock+ using the PARSEC benchmark suite and four real-world applications. The experimental results showed that HistLock+ was 122% faster and 28% more memory-efficient than the previous state-of-the-art complete hybrid race detector. Moreover, HistLock+ achieved the highest effectiveness in race detection among all evaluated race detectors in our experiment.**

***Index Terms*—Data race detection, dynamic analysis, hidden races, multithreaded programs, testing.**

Acronyms and Abbreviations

LD	Locking discipline.
HB	Happens-before.
LRC	Lock release counter.

Manuscript received August 7, 2017; revised January 7, 2018; accepted April 18, 2018. Date of publication August 6, 2018; date of current version August 30, 2018. This work was supported in part by the General Research Fund of Research Grants Council of Hong Kong SAR under Grants 111313, Grant 11201114, Grant 11200015, and Grant 11214116, in part by National Natural Science Foundation of China under Grant 61772056 and Grant 61690202, and in part by the research fund of the State Key Laboratory of Virtual Reality Technology and Systems. Associate Editor: Y. Le Traon. *(Corresponding author: W. K. Chan.)*

J. Yang and W. K. Chan are with the Department of Computer Science, City University of Hong Kong, Kowloon Tong, Hong Kong (e-mail: jialiyang4@gapps.cityu.edu.hk; wkchan@cityu.edu.hk).

B. Jiang is with the School of Computer Science and Engineering, Beihang University, Beijing 100083, China (e-mail: jiangbo@buaa.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2018.2832226

Nomenclature

CL_x	Common lockset of memory location x.
C_t	Current vector clock of thread t.
$C(e)$	Vector clock at event e.
E_t	Current epoch of thread t.
$E(e)$	Epoch at event e.
L_t	Current lockset of thread t.
$L(e)$	Lockset of thread t when it generates event e.
LRC_t	Current lock release counter of thread t.
$LRC(e)$	Lock release counter at event e.
W_x	WRITE context(s) of memory location x.
R_x	READ context(s) of memory location x.
Σ	Analysis state of HistLock+.

I. Introduction

In a multithreaded program, multiple threads may execute concurrently. A data race (*race* for short) occurs when two threads access the same memory location without proper synchronization and at least one of these accesses is a WRITE [18]. Harmful races may lead to memory corruptions, incorrect outputs, or crashes. The presence of races in a program may also correlate with the presences of other concurrency bugs, such as linearizability errors [4] and atomicity violations [18], [37].

There are two broad categories of race detectors: static (e.g., [1], [9], [11], [30]) and dynamic (e.g., [10], [14], [22], [24], [27], [33]). The former analyzes the program code and infers potential races. It is intricate to isolate real races from such potential ones due to the absence of program inputs to confirm the reported cases. The latter analyzes the memory access events (*memory accesses* for short) and synchronization patterns in a program execution trace (*trace* for short) to infer races. These dynamic race detectors report fewer false positives than their static counterparts.

In dynamic race detection, there are three board analysis approaches: happens-before analysis and its variants (e.g., [10], [13], [15], [23], [28]), lockset analysis [24], and hybrid of the former two (e.g., [14], [22], [27], [32]–[34]).

The happens-before analysis approach tracks the happens-before relations [17] among memory accesses and synchronization events in a trace. However, it is sensitive to thread schedules and can only expose races appearing in given traces.

This class of techniques has been intensively studied [10], [13], [15], [23], [28].

The lockset analysis approach checks whether two threads hold any lock in common when they consecutively access the same memory location, coined as the locking discipline (LD for short) [24]. If no common lock is held, the involving pair of memory accesses forms an LD violation (which nonetheless may or may not be a real race) [24].

The hybrid analysis approach [14], [22], [27], [32]–[34] performs lockset-based analysis to detect LD violations and improves its detection precision by performing (relaxed) happens-before analysis. We refer to this class of race detectors as *hybrid race detectors*. They alleviate the detection imprecision problem incurred by pure lockset-based race detectors and the thread interleaving sensitive problem incurred by pure happens-before race detectors. Interestingly, previous experiments also showed that hybrid race detectors can be quite precise [33].

An important class of hybrid race detectors is the *complete* hybrid race detectors. It ensures to report at least one race, if any, on each memory location. However, keeping all memory accesses and checking every one against those preceding the current access on the same memory location incurs impractical runtime overhead and memory consumption. A practical complete hybrid race detector should maintain a small subset of all memory accesses in its analysis state (*state* for short).

The state-of-the-art complete hybrid race detector is MultiLock-HB [33]. To identify memory accesses that can be dropped from its state, MultiLock-HB compares the lockset of the current memory access a_{n+1} with all its prior memory accesses $A = \{a_1, a_2, \ldots, a_n\}$ that accessed the same memory location (and within the same epoch[1] [10] of a_{n+1}) kept in its state. If any memory access $a' \in A$ has a larger lockset than a_{n+1}, MultiLock-HB removes a' from A. After finishing all lockset comparisons with each memory access in A, it adds a_{n+1} to A. If the lockset of a' is a subset of the lockset of a_{n+1}, MultiLock-HB drops a_{n+1} and skips to perform further lockset comparisons on a_{n+1} with the remaining memory access in A. In other words, on each memory location, MultiLock-HB only keeps memory accesses having the smallest lockset within each epoch in its state by employing exhaustive lockset comparisons. As a result, MultiLock-HB can alleviate memory consumption at the expense of higher runtime slowdown.

This paper extends its preliminary work [34], which presented an incomplete hybrid race detector called *HistLock* that explored the possibility to remove historical memory accesses without involving any lockset comparison. It uses the tuple $\langle tid, fn \rangle$ (referred to as *access site*) to index each memory access event a where the event a is generated by thread *tid* within the body of the function fn. For each memory access a_{n+1} on each memory location, HistLock iteratively compares the access site of a_{n+1} with that of each historical memory access $a' \in Q = [a_1, a_2, \ldots, a_n]$ starting from a_n to a_1. In each iteration, if a_i shares the same *tid* with a_{n+1} is found, HistLock terminates the entire comparison process and then checks whether the function calls generating these two memory accesses (i.e., a_{n+1} and a') are different. If this is the case, HistLock removes all the memory accesses in Q that share the same access site with a', and then appends a_{n+1} to Q. The experimental results showed that HistLock ran significantly faster and consumed less memory than MultiLock-HB. Owing to the above function-based memory access removal strategy, HistLock is incomplete (and detects fewer races than MultiLock-HB).

In this paper, we present *HistLock+,* a novel complete hybrid race detector. HistLock+ is designed to investigate how far a hybrid race detector can go in terms of efficiency without performing lockset comparison as much as possible.

HistLock+ is built on top of two insights. First, on the thread segment in between two consecutive lock releases generated by the same thread, all memory accesses can at most acquire additional locks. Thus, the locksets associating with the memory accesses generated by the same thread cannot lose any lock that has been acquired while the thread executes the instructions in that thread segment. This implies that on such a segment, the lockset of the first encountering memory access on each memory location is always the smallest and all other memory accesses on the same memory location is always its supersets (possibly equal). HistLock+ counts the number of lock releases that each thread has ever performed, and if the counter values of two memory access are the same (of course their epochs are the same too), then these two memory accesses are located in the same thread segment. In this way, there is no need to compare the locksets of these two events, and HistLock+ may drop the later encountering events which have locksets not smaller than the former one.

Since READs and WRITEs on a memory location need to be differentiated for distinguishing READ–WRITE, WRITE–READ, and WRITE–WRITE races, we further ask a question: *To make the detector complete, for the same memory location, is it necessary to keep both the first* WRITE *and the first* READ *on each such thread segment and drop the latter ones?*

Our second insight is that on such a segment, if the first memory access on some memory location is a WRITE, we only need to keep this first WRITE, and can drop the subsequent first READ (if any) on the same memory location. This is because any memory access in race with the first READ that appears after the first WRITE should be a WRITE, which should also be in race with the first WRITE and more importantly has already been kept. We have formally proven the validity of this insight as Theorem 2 to be presented in Section IV.

In short, HistLock+ keeps the first READ for detecting READ–WRITE race(s) until the first WRITE on the same segment is observed and in which case, HistLock+ swaps to keep this first WRITE and drops the first READ for detecting subsequent WRITE–READ and WRITE–WRITE race(s).

We have evaluated HistLock+ on the PARSEC C/C++ suite and four real-world applications including *Aget, Pbzip2, Httpd,* and *MySQL*. We compared it with MultiLock-HB and HistLock by running them on each benchmark 100 times and measured the runtime overhead, memory consumption, and the number of reported races. The results showed that on average, HistLock+ and MultiLock-HB were found to incur 1.69× and 3.75× slowdown, respectively, on top of the underlying Maple framework

[1]Epoch in MultiLock-HB represents the thread segment that separated by "hard order" synchronization primitives such as fork–join, semaphore, and barrier.

[38], which indicated that HistLock+ ran much faster than MultiLock-HB. HistLock+ also consumed 28% less memory than MultiLock-HB, and it was the most effective race detector among the race detectors in our experiment.

The main contribution of this paper is threefold.

1) To the best of our knowledge, this paper presents the *first work* in eliminating skippable memory accesses without any lockset comparison while ensuring the completeness for hybrid race detection.
2) It showed the feasibility of the elimination approach by implementing it as HistLock+.
3) It presents an experiment that has validated HistLock+. The experimental results showed that HistLock+ can be more efficient and effective than the previous state-of-the-art detector.

The rest of this paper is as follows. Section II presents the preliminaries. Section III introduces a motivating example of our work. Section IV presents HistLock+. Section V reports the experimental evaluation on HistLock+. Section VI further discusses the findings from the experiment as well as some issues of HistLock+. Section VII reviews the related work. Section VIII concludes this paper.

II. Preliminaries

This section presents the backgrounds and terminologies commonly used in the dynamic race detection literature.

A. Events and Execution Trace

A race detector monitors events in a program execution, where each *event* e is a triple $\langle t, c, op\rangle$, modeling that event e is generated by thread t at timestamp c to execute operation op.

Each *operation* op is one of the following types: *WRITE*(x) and *READ*(x) for writing and reading on memory location x; *release*(m) and *acquire*(m) for lock release and acquisition on lock m; *fork*(u) and *begin*(u) for forking thread u and u starting its execution; *end*(u) and *join*(u) for thread u terminating its execution, and u joining to its parent thread; *wait*(cv) and *signal*(cv)/*broadcast*(cv) for waiting on condition variable cv and signaling/broadcasting the wait(s) on cv; *prebarrier*(b) and *postbarrier*(b) for entering and exiting barrier b. Note that except *WRITE*(x) and *READ*(x), other operation types are paired by the way they were introduced.

We refer to an event of the first two operation types (i.e., *WRITE* and *READ*) as *memory access* and an event of the remaining operation types as *synchronization primitive*. Two synchronization primitives are *paired* if their operation types are paired. An *execution trace* τ (*trace* τ for short) is a sequence of events $[e_1, e_2, \ldots, e_n]$.

Following previous works [10], [13], [24], [33], we assume that a lock can only be acquired by at most one thread at a time. If a thread t acquires a lock m, then t holds m and *cannot* reacquire m until it releases m.

B. Happens-Before Relation and HB-Race

The *happens-before* (HB) relation [17], denoted by $\rightarrow_{hb}$, is a partial order over the events in a trace. The following three rules define this relation.

1) If a thread generates both α and β where α precedes β, then $\alpha \rightarrow_{hb} \beta$.
2) If two different threads generate α and β respectively, both α and β are paired synchronization primitives, and α precedes β, then $\alpha \rightarrow_{hb} \beta$.
3) If $\alpha \rightarrow_{hb} \beta$ and $\beta \rightarrow_{hb} \gamma$, then $\alpha \rightarrow_{hb} \gamma$.

The *must-happen-before* (*mhb*) relation [13], denoted by $\Rightarrow_{mhb}$, is a relaxed HB relation defined as follows.

1) If α and β are events generated by the same thread, and α precedes β, then $\alpha \Rightarrow_{mhb} \beta$.
2) If two different threads generate α and β respectively, both α and β are paired synchronization primitives *other than* lock release and acquisition, and α precedes β, then $\alpha \Rightarrow_{mhb} \beta$.
3) If $\alpha \Rightarrow_{mhb} \beta$ and $\beta \Rightarrow_{mhb} \gamma$, then $\alpha \Rightarrow_{mhb} \gamma$.

In other words, the mhb relation encodes the "hard order" of a multithreaded program execution synchronized by fork–join, semaphore, and barrier primitives. Note that lock imposes *no* "hard" execution order for two critical sections (protected by the same lock) from two different threads.

Events α and β are *concurrent* if neither $\alpha \rightarrow_{hb} \beta$ nor $\beta \rightarrow_{hb} \alpha$. If α or β (at least one of them) is a WRITE, then α and β form a *HB-race* [10]. Events α and β are *mhb-concurrent* if neither $\alpha \Rightarrow_{mhb} \beta$ nor $\beta \Rightarrow_{mhb} \alpha$.

C. Vector Clock and Epoch

A *vector clock* [20] is a tuple of timestamps (e.g., numbers), each recording the clock of an entity (e.g., thread) at a specific execution point. We refer to the vector clock being currently held by a thread t as C_t. We also denote the vector clock of a thread when it generates an event e by $C(e)$.

Each thread t keeps one vector clock C_t. The timestamp of a thread u in C_t is denoted by $C_t[u]$. Specifically, $C_t[t]$ is the timestamp of t itself in its vector clock. We denote the vector clock in which every timestamp is 0 by $\bot_{VC}$. We use $C_{t1} \sqcup C_{t2}$ to denote the operation that the timestamp is $\max(C_{t1}[u], C_{t2}[u])$ for each thread u.

An *epoch* [10] is a pair of a timestamp c and a thread t, denoted as $c@t$, where $c = C_t[t]$. We refer to the current epoch of thread t as E_t and denote the epoch of a thread when it generates an event e by $E(e)$. Following [10], we denote $c \leq C_t[t]$ by $c@t \preccurlyeq C_t$ (or $\not\preccurlyeq$ otherwise). The epoch of a thread t increases whenever t generates a target synchronization primitive (e.g., lock release, fork, and join) so that the relation $\rightarrow_{hb}$ or the relation $\Rightarrow_{mhb}$ in a trace can be tracked.

D. LD and ∅-Race

We refer to the set of locks (i.e., *lockset*) being currently held by a thread t as L_t. We also denote the lockset of a thread when it generates an event e by $L(e)$.

The LD [24] is an assertion: For every memory location x, there is a *nonempty* common lockset CL_x hold by all threads when they consecutively access the same memory location x in a trace.

In the original lockset algorithm [24], before any access to a shared memory location x in a trace τ, the location x is marked to associate with the set of all possible locks CL_x. Whenever thread t (which holds the lockset L_t at that moment) accesses

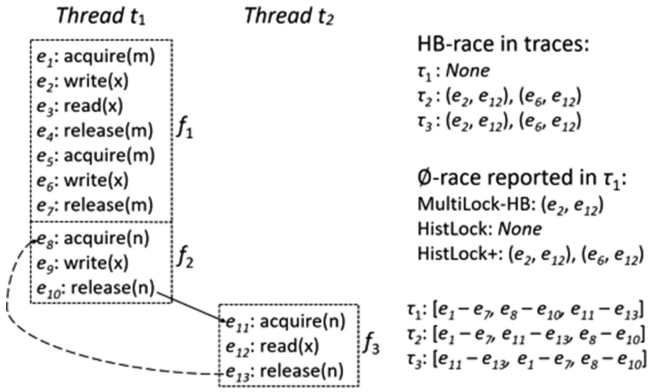

Fig. 1. Motivating example. Events are labeled by $e_1 - e_{13}$; x is a memory location; m, n are locks; $f_1 - f_3$ are function calls. Solid arrow indicates cross-thread HB relation in trace τ_1; dash arrow shows the cross-thread HB relation in traces τ_2 and τ_3.

x, it updates CL_x to $CL_x \cap L_t$ (denoted as CL'_x) and reports an LD violation if CL'_x is empty.

More generally, for two arbitrary memory accesses α and β from two different threads accessing the same memory location x, CL_x is defined as the intersection of their locksets, that is, $CL_x = L(\alpha) \cap L(\beta)$. If $L(\alpha) \cap L(\beta) = \emptyset$, then α and β form an LD violation. Moreover, if α or β (at least one of them) is a WRITE; they are *mhb-concurrent*; they form an LD violation, then α and β form a *Ø-race* [32].

III. Motivating Example

Fig. 1 shows an example to motivate our work. It illustrates a trace τ_1, which consists of 2 threads (t_1 and t_2), 3 function calls (f_1, f_2, and f_3), and 13 events labeled as $e_1, e_2, \ldots, e_{13}$. Functions $f_1 - f_3$ generate event sequences $e_1 - e_7, e_8 - e_{10}$, and $e_{11} - e_{13}$, respectively. Five of these events (e_2, e_3, e_6, e_9, and e_{12}) access memory location x. The trace τ_1 executes these 13 events as follows: t_1 executes f_1 and f_2, then t_2 executes f_3. We simply refer to the execution as $f_1 \rightarrow f_2 \rightarrow f_3$.

In trace τ_1, there is one cross-thread HB relation, which is $e_{10} \rightarrow_{hb} e_{11}$. With this HB relation and two others from the program orders $e_2 \rightarrow_{hb} e_{10}$ and $e_{11} \rightarrow_{hb} e_{12}$, we have $e_2 \rightarrow_{hb} e_{12}$, which protects x from concurrent accesses of e_2 and e_{12}. So, happens-before race detectors (e.g., [10]) cannot detect any race in τ_1.

Next, let us consider a trace τ_2 generated by an alternative thread schedule: $f_1 \rightarrow f_3 \rightarrow f_2$, where t_1 executes f_1 first, then t_2 executes f_3, and finally t_1 executes f_2. In this case, the cross-thread HB relation is $e_{13} \rightarrow_{hb} e_8$, and this HB relation does not protect the concurrent memory accesses from e_2 and e_{12}, and from e_6 and e_{12}. So, in trace τ_2, there are two HB-races, denoted as (e_2, e_{12}) and (e_6, e_{12}), respectively. Similarly, trace τ_3 under execution $f_3 \rightarrow f_1 \rightarrow f_2$ also has the HB-races (e_2, e_{12}) and (e_6, e_{12}).

Note that all the events in τ_1 are *mhb-concurrent* due to no mhb relation among them. For events e_2 and e_{12} in trace τ_1, the lockset $L(e_2)$ is $\{m\}$, and the lockset $L(e_{12})$ is $\{n\}$. Because e_2 and e_{12} are *mhb-concurrent*, and they cause an LD violation (i.e., $L(e_2) \cap L(e_{12}) = \{m\} \cap \{n\} = \emptyset$), the pair (e_2, e_{12}) is a (WRITE–READ) Ø-race in τ_1. Similarly, the pair (e_6, e_{12}) is also a (WRITE–READ) Ø-race in τ_1

MultiLock-HB [33]: For each memory access, MultiLock-HB keeps a pair of epoch and lockset as the *context* of the event. In analyzing trace τ_1, for memory location x, it maintains two sets of WRITE events denoted as $W_x[t_1]$ and $W_x[t_2]$, and two sets of READ events denoted as $R_x[t_1]$ and $R_x[t_2]$. Columns "Context (ML)" and "Analysis State (ML)" in Fig. 2 show the event context and the sets of events kept by MultiLock-HB while processing τ_1, respectively.

MultiLock-HB analyzes trace τ_1 as follows: Suppose the four sets $W_x[t_1], W_x[t_2], R_x[t_1]$, and $R_x[t_2]$ are empty initially. MultiLock-HB increases thread's epochs on events for mhb relations. Thus, all the memory accesses in f_1 and f_2 share the same epoch. On e_2, MultiLock-HB adds e_2 into $W_x[t_1]$ since e_2 is the first WRITE in τ_1. On e_3, because it shares the same epoch with e_2, MultiLock-HB performs a lockset comparison between the contexts of e_2 and e_3 to determine whether there is any lock subset relation between them. In this case, MultiLock-HB deems e_3 as *redundant*, and drops e_3 because e_3 and e_2 share the same lockset. Similarly, on e_6, MultiLock-HB performs a lockset comparison between the contexts of e_2 and e_6, and drops e_6 because e_6 and e_2 share the same lockset. On e_9, MultiLock-HB performs a lockset comparison between the contexts of e_2 and e_9. Since e_9 associates with a different context from e_2 and e_9 has no lock subset relation with e_2, MultiLock-HB adds e_9 into $W_x[t_1]$.

On e_{12} generated by t_2, MultiLock-HB adds e_{12} into $R_x[t_2]$ since both sets $W_x[t_2]$ and $R_x[t_2]$ are empty. For detecting races, because $W_x[t_1]$ keeps e_2 and e_9, and $R_x[t_2]$ keeps e_{12}, we have $L(e_2) \cap L(e_{12}) = \{m\} \cap \{n\} = \emptyset$, and $L(e_9) \cap L(e_{12}) = \{n\} \cap \{n\} = \{n\}$, respectively. So, MultiLock-HB reports (e_2, e_{12}) as a Ø-race. However, it misses reporting Ø-race (e_6, e_{12}) because e_6 is not kept in its state. Note that MultiLock-HB performs lockset comparisons three times to determine lock subset relation in this example between the contexts of e_2 and e_3, e_2 and e_6, and e_2 and e_9, respectively.

MultiLock-HB guarantees to report *at least one* Ø-race (i.e., (e_2, e_{12}) in this example) on each memory location within each epoch if any, and reports no false positive [33]. In the previous experiment [33], MultiLock-HB was significantly less efficient in both runtime slowdown and memory consumption than its incomplete counterpart AccuLock [32]. The experiment indicates that achieving completeness in race detection and retaining high efficiency could be *challenging*.

HistLock [34]: Consider τ_1 again. HistLock keeps the function calls in the corresponding event contexts, and maintains two lists R_x and W_x for keeping READ and WRITE events on memory location x respectively, where the order of events in each list is the same as the order of events in trace τ_1. Unlike MultiLock-HB, HistLock encodes the epoch of each thread by both mhb relation and lock release event [32]. Columns "Context (HL)" and "Analysis State (HL)" in Fig. 2 show the event context and the two lists of events kept by HistLock on processing trace τ_1, respectively.

Event	Context (ML)	Analysis State (ML)	Context (HL)	Analysis State (HL)	Context (HL+)	Analysis State (HL+)
e_2	$\langle 1@1, \{m\}\rangle$	$W_x[t_1] = \{e_2\}$	$\langle 1@1, \{m\}, f_1\rangle$	$W_x = [e_2]$	$\langle 1@1, \{m\}, 0\rangle$	$W_x = [e_2]$
e_3	$\langle 1@1, \{m\}\rangle$	$W_x[t_1] = \{e_2\}$	$\langle 1@1, \{m\}, f_1\rangle$	$W_x = [e_2]$	$\langle 1@1, \{m\}, 0\rangle$	$W_x = [e_2]$
e_6	$\langle 1@1, \{m\}\rangle$	$W_x[t_1] = \{e_2\}$	$\langle 2@1, \{m\}, f_1\rangle$	$W_x = [e_2, e_6]$	$\langle 1@1, \{m\}, 1\rangle$	$W_x = [e_2, e_6]$
e_9	$\langle 1@1, \{n\}\rangle$	$W_x[t_1] = \{e_2, e_9\}$	$\langle 3@1, \{n\}, f_2\rangle$	$W_x = [e_9]$	$\langle 1@1, \{n\}, 2\rangle$	$W_x = [e_2, e_6, e_9]$
e_{12}	$\langle 1@2, \{n\}\rangle$	$W_x[t_1] = \{e_2, e_6\}, R_x[t_2] = \{e_{12}\}$	$\langle 1@2, \{n\}, f_3\rangle$	$W_x = [e_9], R_x = [e_{12}]$	$\langle 1@2, \{n\}, 0\rangle$	$W_x = [e_2, e_6, e_9], R_x = [e_{12}]$

Fig. 2. Summary of analysis variables of trace τ_1. Column "Event" indicates the event to be processed. Columns "Context (ML)," "Context (HL)," and "Context (HL+)" show the context of each event in MultiLock-HB, HistLock, and HistLock+, respectively. Columns "Analysis State (ML)," "Analysis State (HL)," and "Analysis State (HL+)" show the events kept by MultiLock-HB, HistLock, and HistLock+ in their event placeholders after processing each event, respectively. Empty event placeholders are omitted.

Suppose the two lists W_x and R_x are empty initially. On e_2, HistLock appends e_2 to W_x since it is the first WRITE in τ_1. On e_3, HistLock drops it because e_3 and e_2 share the same epoch. On e_6, HistLock appends e_6 to W_x because e_6 and e_2 are in different epochs. On e_9, HistLock finds that the function call f_2 of e_9 (generated by t_1) is different from the function call f_1 of e_2 (the last historical WRITE generated by t_1 in W_x), so it removes all historical WRITEs that are generated by t_1 with f_1 in W_x (i.e., e_2) and appends e_9 to W_x. On e_{12}, HistLock appends it to R_x, because e_{12} is associated with a different epoch (thus under a different context). For detecting races, because W_x keeps e_9 and R_x keeps e_{12}, we have $L(e_9) \cap L(e_{12}) = \{n\} \cap \{n\} = \{n\}$. Thus, HistLock misses reporting the $\emptyset$-races (e_2, e_{12}) and (e_6, e_{12}).

In the previous experiment [34], HistLock incurred significantly lower runtime overhead than MultiLock-HB. However, the use of function call as the basis for historical memory access removal could not be generalized to make a hypothetical version of HistLock to be a complete detector.

We thus ask the following three questions:

1) Is it possible to design a complete hybrid race detector without lockset comparison in memory access maintenance?
2) Can such a detector be efficient in both runtime overhead and memory consumption?
3) Can such a detector achieve higher effectiveness in race detection than the state-of-the-art?

HistLock+: HistLock+ introduces the notion of lock release counter (LRC) for each thread, which increments by 1 after each lock release. It increases a thread's epoch on each hard-order synchronization events (e.g., fork–join, semaphore, and barrier primitives). HistLock divides the event sequence generated by the same thread into segments according to each combination of epoch and LRC, which is referred to as an *EL-span*.

In trace τ_1, events e_4, e_7, and e_{10} are lock releases generated by thread t_1, and e_{13} is lock release generated by thread t_2. Suppose that the LRC for each thread is initially 0. Thus, the LRCs for the events $e_1 - e_4, e_5 - e_7, e_8 - e_{10}$, and $e_{11} - e_{13}$ are 0, 1, 2, and 0, respectively, and those four event segments are grouped under four different EL-spans.

Similar to HistLock, HistLock+ maintains two lists R_x and W_x to keep the READ and WRITE events on memory location x following the appearing event order in trace τ_1, respectively.

For each EL-span, HistLock+ only keeps the first WRITE if no READ precedes this first WRITE in the same EL-span, or both the first READ and the first WRITE otherwise.

Columns "Context (HL+)" and "Analysis State (HL+)" in Fig. 2 show the event context and the two lists of events kept by HistLock+ on processing τ_1, respectively.

Suppose W_x and R_x are empty initially. On e_2, HistLock+ appends e_2 to W_x because e_2 is the first WRITE in the corresponding EL-span. On e_3, HistLock+ skips it because the first WRITE e_2 of the same EL-span has been kept in W_x. On e_6 and e_9, HistLock+ appends e_6 and e_9 into W_x because they are the first WRITEs of two new EL-spans. On e_{12}, similarly, HistLock+ appends it to R_x because e_{12} is the first READ of a new EL-span. For detecting races, because W_x keeps e_2, e_6, and e_9, and R_x keeps e_{12}, we have $L(e_2) \cap L(e_{12}) = \{m\} \cap \{n\} = \emptyset, L(e_6) \cap L(e_{12}) = \{m\} \cap \{n\} = \emptyset$, and $L(e_9) \cap L(e_{12}) = \{n\} \cap \{n\} = \{n\}$, respectively. As a result, HistLock+ reports $\emptyset$-races (e_2, e_{12}) and (e_6, e_{12}).

Discussion: MultiLock-HB must check the lock subset relation on *every* encountered memory access sharing the same epoch, and it is unable to skip subsequent memory accesses because lock release events are *not* modeled in its epoch updating scheme. As a result, MultiLock-HB cannot determine whether there is any lock release in between two consecutive memory accesses without lockset comparison. HistLock+ eliminates such lockset comparisons by introducing LRC and EL-span to determine the lock subset relation among consecutive memory accesses with lower overhead and scalar comparisons.

Although HistLock increases its epoch on each lock release event [31] to skip consecutive memory accesses without comparing their locksets, its strategy is heuristic to approximate the lock subset relation. In the design of HistLock+, we decouple LRC from epoch. By doing so, in the next section, we show that correct lock subset relations can be maintained. Consequently, the memory access maintenance strategy in HistLock+ has been significantly redesigned and is very different from that of HistLock.

IV. HistLock+

In this section, we present the design of HistLock+. HistLock+ maintains *at most* one historical WRITE and optionally one READ for each EL-span on each memory location, which is proven to be sufficient to ensure at least one Ø-race, if any, is reported on each memory location. In fact, HistLock+ has a stronger guarantee: It ensures to report exactly one (if any) racy memory access for either READs or WRITEs between two EL-spans from two different threads irrespective to the total number of Ø-races. While attaining this level of completeness, HistLock+ checks whether a memory access can be dropped in $O(1)$ time complexity, and efficiently maintains its state. We summarize the theory behind HistLock+ as Theorems $1-3$ and the algorithmic design of HistLock+ in Sections IV-A–IV-B, and discuss the design rationale on its efficiency in Section IV-C.

A. Relevant Ø-Race

In Section III, we have mentioned the notion of LRC in HistLock+. In this section, we first define LRC, and show how to use it to develop Theorem 1 to quantify the condition of identifying "inferable" Ø-races.

We define LRC of a thread t as the total number of lock releases ever generated by t. We refer to the LRC currently being held by a thread t as LRC_t. We also refer to the LRC of a thread when it generates an event e as $LRC(e)$.

For ease of our presentation, we refer to each combination of thread's epoch and LRC as an EL-span. As such, for any two events α and β, $E(\alpha) = E(\beta) \wedge \text{LRC}(\alpha) = LRC(\beta)$ if and only if α and β are in the same EL-span.

Property 1 (Lockset Subsumption): For any events α and β such that α precedes β in a trace, if α and β are in the same EL-span, then $L(\alpha) \subseteq L(\beta)$.

Proof: By definition, an epoch $c@t$ is a shorthand of the pair of c and t for the relation $c = C_t[t]$. Because $E(\alpha) = E(\beta)$, events α and β are generated by the same thread t. Since t generates events sequentially, and α precedes β in the trace, t has generated α before generating β. Recall that any lock once acquired by t continues to be held until t releases that lock. Because $LRC(\alpha) = LRC(\beta)$, there is no lock release generated by t between α and β. Thus, every lock being held by t when t generates α will be carried forward to the execution point that t generates β. Hence, we have the lockset $L(\alpha)$ being a subset of the lockset $L(\beta)$. ■

We make a few notes: If $E(\alpha) = E(\beta)$ and $LRC(\alpha) \neq LRC(\beta)$, there must be a lock release, say on lock m, occurred in between α and β. Nonetheless, there is no guarantee that the same lock m will be reacquired by the same thread before β occurs. So, the subset relation on the locksets of α and β cannot be inferred if only $E(\alpha) = E(\beta)$ is known. On the other hand, if $LRC(\alpha) = LRC(\beta)$ and $E(\alpha) \neq E(\beta)$, then α and β may be generated by different threads, and there is no subset relation between the locksets of two events generated by different threads even if their LRC values are the same.

Theorem 1: Suppose that α and β are two memory accesses on the same memory location such that α and β are in the same EL-span, and α precedes β. Then, for any memory access γ such that neither α nor γ are READs, if β and γ form a Ø-race, then α and γ also form a Ø-race.

Proof: Case A: β is a READ. Suppose that a memory access γ is in Ø-race with β. In this case, we have both neither $\beta \Rightarrow_{mhb} \gamma$ nor $\gamma \Rightarrow_{mhb} \beta$ and $L(\beta) \cap L(\gamma) = \emptyset$. Recall that α and β are in the same EL-span, we have $E(\alpha) = E(\beta)$, so the two events α and β are generated by the same thread with the same epoch. By the definition of mhb relation, we have neither $\alpha \Rightarrow_{mhb} \gamma$ nor $\gamma \Rightarrow_{mhb} \alpha$. Also, when α precedes β, by Property 1 we have $L(\alpha) \subseteq L(\beta)$. Because $L(\beta) \cap L(\gamma) = \emptyset$, we should have $L(\alpha) \cap L(\gamma) = \emptyset$ as the lockset $L(\alpha)$ cannot contain any lock other than those locks in $L(\beta)$.

We also know that β is a READ. So, γ must be a WRITE so that γ can be in Ø-race with β (as no race could be formed between two READs). If α is a WRITE, then α and γ form a WRITE–WRITE Ø-race. If α is a READ, then α and γ form a READ–WRITE Ø-race.

Case B: β is a WRITE. Suppose that the next memory access γ is in Ø-race with β. We have neither $\alpha \Rightarrow_{mhb} \gamma$ nor $\gamma \Rightarrow_{mhb} \alpha$ and $L(\alpha) \cap L(\gamma) = \emptyset$. There are four combinations of memory access types between α and γ to consider.

1) If both α and γ are READs, then α and γ do not form any race.
2) If both α and γ are WRITEs, then α and γ form a WRITE–WRITE Ø-race.
3) If α is a READ and γ is a WRITE, then α and γ form a READ–WRITE Ø-race.
4) If α is a WRITE and γ is a READ, then α and γ form a WRITE–READ Ø-race.

Therefore, we conclude that if neither α nor γ are READs and γ is in Ø-race with β, then γ is in Ø-race with α. ■

Corollary 1 (Relevant Ø-race): Suppose three memory accesses α, β, and γ satisfy Theorem 1. If β and γ form a Ø-race, then α and γ always form a Ø-race.

B. Algorithm

Corollary 1 implies a Ø-race detection strategy, which needs not to keep memory access β in provision of memory access γ and still reports a relevant warning on a Ø-race involving β and γ *indirectly* via the pair of memory accesses α and γ. Also, based on Theorem 1, this strategy does not involve any lockset comparison. For ease of reference, we refer to such a memory access β as *skippable memory access*.

The HistLock+ algorithm below is designed by extending the above strategy. It keeps the first memory access (if the first event is a WRITE) or both the first READ and the first WRITE (if the first event is a READ) on each memory location generated by each thread in each EL-span. It guarantees no miss on reporting Ø-races for nonskippable memory accesses at EL-span level. Theorem 2 backs this extended guarantee.

Theorem 2 (Complete): Let x be a memory location. Suppose that *Event*(x) is the sequence of all memory accesses accessing x that appear in a trace. Using the above extended strategy, for every pair of accesses involved in a Ø-race, there is at least one relevant Ø-race reported in each EL-span if the span contains any Ø-race.

Proof: We first partition *Event*(x) into equivalence classes by EL-span. That is, a set of the equivalence class $[e] = \{e' \in Event(x) \mid E(e) = E(e') \wedge LRC(e) = LRC(e')\}$ is constructed. Note that events in each class $[e]$ are ordered by how they occur in the given trace. Recall the extended strategy keeps the very first event, say α, of each class $[e]$. If α is a READ, the strategy further keeps the very first WRITE, say β, to prevent missing a WRITE–READ Ø-race, if any, between β and any event γ when α and γ are both READs. By Theorem 1, if γ is in Ø-race with any event in $[e]$, there is always a Ø-race between α and γ (if neither α nor γ are READs). In case α and γ are both READs, a Ø-race is still reported via β and γ, because the strategy further keeps β (i.e., the very first WRITE in $[e]$) if α is a READ. ■

Based on Theorem 2, for each such equivalence class $[e]$, we should collect the epoch and the LRC of the first event involving the Ø-race to be reported. Moreover, to detect a Ø-race, we should collect the lockset of the two events involving Ø-race.

As such, we define the *context* of a memory access a as the triple $\langle E(a), L(a), LRC(a)\rangle$, representing the epoch, lockset, and LRC when a thread generates a. Specifically, at the execution point when a thread t generates a, we have $E_t = E(a), L_t = L(a)$ and $LRC_t = LRC(a)$. For ease of our subsequent references, the three elements in a context ctx are referred to as *ctx.epoch, ctx.lockset,* and *ctx.lrc*, respectively.

For ease of presentation of the HistLock+ algorithm, we first define a few auxiliaries: We use the notation $F' := F[x \mapsto V]$ to indicate that F' is constructed from F by keeping everything identical except mapping x to V. The notation $inc_t(VC)$ is the shorthand of $VC[t \mapsto VC[t] + 1]$, which increases the timestamp of thread t by 1 in the vector clock VC. If R_x (W_x, respectively) is a list, we denote the last element of thread t in R_x by $R_{x|lst\_t}$ ($W_{x|lst\_t}$, respectively). Note that $R_{x|lst\_t}$ may or may not be the last element in R_x because the last element may be from a different thread u (same to $W_{x|lst\_t}$). $wt(cv)$ is a function to return one thread [used in *signal*(*cv*)] or the set of threads [used in *broadcast*(*cv*)] waiting on the conditional variable cv. $br(b)$ is a function to return all threads having entered but not exited the barrier b.

The HistLock+ algorithm is summarized in Fig. 3. The algorithm maintains an analysis state Σ, and transits it by processing each event e in an execution trace, denoted as $\Sigma \xrightarrow{e} \Sigma'$. An analysis state $\Sigma = (C, L, R, W, S)$ is a 5-tuple: C and L map a thread t to the vector clock C_t and the lockset L_t that t is holding, respectively. R and W map a memory location x to a list R_x of READ contexts and a list W_x of WRITE contexts on x, respectively. S maps a thread t to a LRC belonging to t, which is denoted by S_t. The initial analysis state is: $\Sigma_0 = (\lambda t.inc_t(\bot_{VC}), \lambda t.\emptyset, \lambda x.\ \bot, \lambda x.\ \bot, \lambda t.0)$.

In the sequel, we present how the HistLock+ algorithm handles each type of event one by one. Readers who are knowledgeable in structural operational semantics can directly refer to Fig. 3.

Fork and Join: On a thread t forking a new thread u, the vector clock C_u is joined with C_t, and finally $C_t[t]$ is increased by 1. On a thread u joining a thread t, the vector clock C_t is joined with C_u, and $C_t[t]$ is increased by 1.

In other words, a thread increases its own epoch by 1 whenever a fork operation or a join operation is performed.

Acquire and Release: On acquiring a (nonreentrant) lock m by a thread t, m is added to the lockset L_t. On releasing a (nonreentrant) lock m by a thread t, m is removed from the lockset L_t, and LRC_t is increased by 1. Acquisition and release on reentrant lock are not modeled as explained in Section II-A.

Signal and Broadcast: On *signal*(*cv*), meaning that thread t sends a signal to notify a thread u that is waiting on a conditional variable cv (i.e., $u \in wt(cv)$), C_u is joined with C_t, and $C_t[t]$ is increased by 1. Similarly, on *broadcast*(*cv*) where thread t broadcasts a signal to notify all those threads that are waiting on cv, for each thread u (where $u \in wt(cv)$) that is waiting on cv, the vector clock C_u is joined with C_t, and $C_t[t]$ is increased by 1.

Barrier: On thread t exiting a barrier b, C_u of each thread u in $br(b)$ is updated to $\sqcup_{u \in br(b)} C_u$, and $C_t[t]$ is increased by 1.

Similar to the handling of fork and join operations, a thread will increase its own epoch by 1 when it performs *signal*(*cv*) or *broadcast*(*cv*), or exits a barrier.

READ: On an event rd reading a memory location x generated by the thread t, the algorithm performs the following. The algorithm first checks whether both $R_{x|lst\_t}.epoch = E_t$ and $R_{x|lst\_t}.lrc = S_t$, then checks whether both $W_{x|lst\_t}.epoch = E_t$ and $W_{x|lst\_t}.lrc = S_t$. If either condition is met, the processing on rd ends (see [HL+ READ Short-Circuited]), and rd is dropped. Otherwise, the algorithm further processes rd via the transition rule [HL+ READ]. It appends the context of rd to R_x, then checks whether each WRITE context wc in W_x satisfies both $wc.epoch \preccurlyeq C_t$ and $wc.lockset \cap L_t \neq \emptyset$ (as highlighted in transition rule [HL+ READ]). If this is not the case, HistLock+ reports a WRITE–READ Ø-race produced by wc and rd.

WRITE: On an event wt writing a memory location x generated by a thread t, the algorithm performs the following. It first checks whether $W_{x|lst\_t}.epoch = E_t$ and $W_{x|lst\_t}.lrc = S_t$ are met. If this is the case, the processing on wt ends (see [HL+ WRITE Short-Circuited]), and wt is dropped. Otherwise, the algorithm further processes wt via the transition rule [HL+ WRITE]. It appends wt to W_x, then, the algorithm checks: 1) whether each READ context rc in R_x satisfies both $rc.epoch \preccurlyeq C_t$ and $rc.lockset \cap L_t \neq \emptyset$, and 2) whether each WRITE context wc in W_x satisfies both $wc.epoch \preccurlyeq C_t$ and $wc.lockset \cap L_t \neq \emptyset$ (as highlighted in transition rule [HL+ WRITE]). If either is not the case, HistLock+ reports a READ–WRITE Ø-race produced by rc and wt, or a WRITE–WRITE Ø-race produced by wc and wt.

As presented in Fig. 3, HistLock+ updates C and S of analysis state Σ only by tracking synchronization primitives without involving any memory access. Because epoch can be extracted from C, both epoch and LRC are already obtained (thus, EL-span can be determined) from the synchronization tracking phase. Based on Property 1, the lockset subset relation between two memory accesses then can be determined in the same phase. In contrast, MultiLock-HB needs two phases (*synchronization tracking* and *lockset comparison*) to determine lockset subset relation and remove redundant accesses (see Section III).

Theorem 3 (Sound): HistLock+ only reports Ø-races.

HL+ Analysis State Σ (C, L, W, R, S)

$$C : Tid \to VC \qquad L : Tid \to \overline{Lockset} \qquad S : Tid \to LRC$$
$$W : Loc \to \overline{Context} \qquad R : Loc \to \overline{Context} \qquad Context : \langle Epoch, Lockset, LRC \rangle$$

[HL+ Fork]
$$\frac{C' \coloneqq C[u \mapsto C_u \sqcup C_t, t \mapsto inc_t(C_t)]}{(C, L, W, R, S) \xrightarrow{t:fork(u)} (C', L, W, R, S)}$$

[HL+ Join]
$$\frac{C' \coloneqq C[t \mapsto C_t \sqcup C_u,\ u \mapsto inc_t(C_t)]}{(C, L, W, R, S) \xrightarrow{t:join(u)} (C', L, W, R, S)}$$

[HL+ Signal/Broadcast]
$$\frac{\begin{array}{c} C' \coloneqq \lambda u.\, \text{if } u \in wt(cv) \text{ then } C_u \sqcup C_t \\ \text{else if } u = t \text{ then } inc_u(C_u) \text{ else } C_u \end{array}}{(C, L, W, R, S) \xrightarrow{t:signal(cv)/broadcast(cv)} (C', L, W, R, S)}$$

[HL+ Barrier]
$$\frac{C' \coloneqq [t \mapsto inc_t(\sqcup_{u \in br(b)} C_u)]}{(C, L, W, R, S) \xrightarrow{t:postbarrier(b)} (C', L, W, R, S)}$$

[HL+ Acquire]
$$\frac{L' \coloneqq L[t \mapsto L_t \cup \{m\}]}{(C, L, W, R, S) \xrightarrow{t:acquire(m)} (C, L', W, R, S)}$$

[HL+ Release]
$$\frac{\begin{array}{c} L' \coloneqq L[t \mapsto L_t \backslash \{m\}] \\ S' \coloneqq S[t \mapsto S_t + 1] \end{array}}{(C, L, W, R, S) \xrightarrow{t:release(m)} (C, L', W, R, S')}$$

[HL+ Write Short-Circuited]
$$\frac{W_{x|lst\_t}.epoch = E_t \wedge W_{x|lst\_t}.lrc = S_t}{(C, L, W, R, S) \xrightarrow{t:write(x)} (C, L, W, R, S)}$$

[HL+ Read Short-Circuited]
$$\frac{\begin{array}{c} R_{x|lst\_t}.epoch = E_t \wedge R_{x|lst\_t}.lrc = S_t \\ W_{x|lst\_t}.epoch = E_t \wedge W_{x|lst\_t}.lrc = S_t \end{array}}{(C, L, W, R, S) \xrightarrow{t:read(x)} (C, L, W, R, S)}$$

[HL+ Write]
$$\frac{\begin{array}{c} W' \coloneqq [x \mapsto W_x.append(E_t, L_t, S_t)] \\ \forall rc \in R_x : rc.epoch \preccurlyeq C_t \wedge rc.lockset \cap L_t \neq \emptyset \\ \forall wc \in W_x : wc.epoch \preccurlyeq C_t \wedge wc.lockset \cap L_t \neq \emptyset \end{array}}{(C, L, W, R, S) \xrightarrow{t:write(x)} (C, L, W', R, S)}$$

[HL+ Read]
$$\frac{\begin{array}{c} R' \coloneqq [x \mapsto R_x.append(E_t, L_t, S_t)] \\ \forall wc \in W_x : wc.epoch \preccurlyeq C_t \wedge wc.lockset \cap L_t \neq \emptyset \end{array}}{(C, L, W, R, S) \xrightarrow{t:read(x)} (C, L, W, R', S)}$$

Fig. 3. HistLock+ algorithm.

Proof: In the algorithm, the timestamps of each thread are maintained according to the definition of mhb relation, and we use vector clocks (rather than epochs) to capture the partial order among synchronization primitives excluding events for lock acquisitions and releases. Each warning is reported right after the check on whether the two events involving in that $\emptyset$-race is mhb-*concurrent* and their locksets are disjoint. Thus, HistLock+ only reports $\emptyset$-races. ■

C. Design for Efficiency

HistLock+ is designed for efficient processing of individual events and the whole event stream in a trace.

On processing each memory access e, HistLock+ takes a "short-circuited" path right after it has identified e as a memory access (via [HL+ READ Short-Circuited] and [HL+ WRITE Short-Circuited])—It always performs two *scalar comparisons* (each between two integers, one pair for epochs and another pair for LRCs) to check whether e is skippable.

V. EXPERIMENTAL EVALUATION

In this section, we report an evaluation of HistLock+ on runtime slowdown, memory consumption, and race detection effectiveness.

A. Experimental Setup

We have implemented FastTrack (FT) [10], HistLock (HL) [34], MultiLock-HB (ML) [33], and HistLock+ (HL+) as online race detectors in the same framework on Maple [38]. FastTrack is a pure happens-before race detector. HistLock is an *incomplete* hybrid race detector. Both MultiLock-HB and HistLock+ are *complete* hybrid race detectors. All the above detectors are *sound*, that is, they will *not* report false positive in detecting HB-races (for FastTrack) or $\emptyset$-races (for HistLock, MultiLock-HB, and HistLock+).

Our tool was based on the source code of $Djit$ [22] implemented in Maple [38]. Since Maple provides callback analysis function probes to instrument interested events (e.g., synchro-

TABLE I
BENCHMARK STATISTICS

Benchmark	KLOC	# of Thds.	# of Memory Accs. (in 10^3)		# of Synchronization Primitives		
			Read	Write	Lock	Signal	Barrier
blackscholes	1.2	4	32	2	2	0	0
bodytrack	11.0	5	64,992	13,694	1,126	76	256
canneal	4.3	4	71	12	6	0	8
dedup	3.7	12	84,940	15,661	967	401	34
ferret	10.8	18	27,874	4,625	19,049	41	38
fluidanimate	2.1	4	86,539	5,059	364,469	48	18
raytrace	14.7	4	28	0.2	20	3	15
streamcluster	2.8	8	642	56	22,538	11,265	19
swaptions	1.6	4	857	209	2	0	0
vips	134.3	3	8,230	1,775	887	95	47
x264	39.0	5	494	240	20	9	6
aget	2.7	2	43	22	8	17	0
pbzip2	2.1	3	190	37	11	2	3
httpd	231.3	4	4,741	948	13,765	13	22
mysql	368.7	12	15,341	7,671	126,377	36	31
Total	830.3	92	295,014	50,011	549,247	12,006	497

nization primitives and memory accesses) of a program on the fly, we implemented those detectors by inserting those probes and realizing their race detection algorithms straightforwardly. For each execution trace, every detector reported each racy pair of events involving in a race at the statement level. In other words, if a race was detected by any of our implemented detectors, it reported the pair of racy statements in the source code to represent the race. Thus, different (dynamic) racy events in the trace are counted as the same race if they are from the same (static) racy statements. We also note that like previous works (e.g., [10], [13], [33]), our experiment treated different type of races (i.e., WRITE–WRITE race, WRITE–READ race, and READ–WRITE race) from the same racy statements as different race instances.

We chose the PARSEC [2] benchmark suite 2.1 as well as four real-world open-source applications Aget 0.57, Pbzip2 0.9.4, Apache HTTP server 2.0.48, and MySQL 4.0.12 as our benchmarks.

The PARSEC benchmark suite is a set of C/C++ multithreaded programs, which is also used in previous experiments (e.g., [5], [8], [14]) on race detection. Because *freqmine* was implemented by OpenMP API [7] and our tool only supported POSIX Threads API (i.e., *pthreads*), and *facesim* crashed when we ran it under the Maple without our tool, we excluded these two benchmarks from our evaluation. Our experiment used all the remaining 11 benchmarks and executed them with the shipped *simdev* [2] as test inputs.

Aget (*aget*) is a multithreaded download accelerator and Pbzip2 (*pbzip2*) is a parallel compressor. Apache HTTP Server (*httpd*) and MySQL (*mysql*) are multithreaded web server and database server, respectively, widely used in practice. The former two benchmarks represent small to medium size multithreaded programs used in the real world, and the latter two benchmarks represent large-scale enterprise applications. These four real-world applications are widely studied in the literature (e.g., [31], [37], [38], [41]) on concurrency bug detection. The test inputs of these four benchmarks are taken from [38].

Table I summarizes the descriptive statistics of each benchmark. Column "KLOC" shows size (i.e., thousands of lines of code) of the program. Column "# of Thds." shows the number of *working* threads allocated by the benchmark, excluding the test harness thread that started the test run. Columns "# of Memory Accs." and "# of Synchronization Primitives" show the average numbers of memory accesses (i.e., READ and WRITE) and synchronization primitives (i.e., *lock release*, *signal/broadcast*, *postbarrier*) counted in the trace of each benchmark in 100 runs, respectively.

Our experiment was performed in VMs each installed with Ubuntu 12.04 (64 bits) with 2.9 GHz E5-2690 quadcore processor and 16 GB physical memory managed by Windows Server 2012 Hyper-V Platform. We measured the means of execution time and memory consumption, and the sum of all distinct reported races (the same racy pair was only counted once) of each detector by running each benchmark 100 times.

We compiled our tool and benchmarks using GCC 4.6.3. We compared the races detected by the tool with those detected by other detectors (e.g., [5]) to assure quality. The tool implemented each LRC as a thread-local integer and major data structures (e.g., *epoch*, *lockset, context*) by directly using the C++ STL library (i.e., <*vector*>, <*set*>, <*map*>, etc.). We chose direct implementations of each algorthm to alleviate the potentail of impelmentation biases due to coding efforts.

B. Threats to Validity

We were aware that the main advantage of HistLock+ is on its efficiency, which was affected by both the algorithmic design and the data structure reported in this paper as well as the actual implementation in code. To alleviate this issue, we used the same data structure library in implementing all four detectors (e.g., all the lists in all four detectors, we implemented by C++ STL library <*list*>) and chose the same implementation strategies and manipulation styles in updating the analysis states of these detectors. The same probes to monitor events were used in these detectors whenever possible. We implemented all race detectors by the most straightforward way according to their algorithms and whenever a particular design/impelementaiton strategy was used, we consistently applied it to every detector (if applicable).

On the other hand, we were aware of some unique design/implementation in a certain detector may affect the results. For instance, MultiLock-HB must additionally use the *set* data structure to realize its algorithm, whereas HistLock and HistLock+ only used *list*. This may cause MultiLock-HB consuming more memory than HistLock and HistLock+ under the same condition. However, using *set* in HistLock and HistLock+ to simulate the behavior of *list* would be against our straightforward implementation principle (and violates the design goals of HistLock and HistLock+). Thus, we consider the comparisions among implemented detectors were still fair.

C. Data Analysis

1) Skippable Memory Access Elimination: Both MultiLock-HB and HistLock+ can eliminate checking on some skippable memory accesses. Table II shows the results on the eliminating

TABLE II
SKIPPABLE MEMORY ACCESS ELIMINATION

Benchmark	Memory Accesses (in 10^3)	LS Comp. (in 10^3)		Time on Elimination (in second)		
		ML	*HL+*	*ML(A)*	*HL+(B)*	*B/A*
blackscholes	34	47	**0**	0.05	0.01	**0.20**
bodytrack	78,686	137,086	**0**	146	10	**0.07**
canneal	83	128	**0**	0.16	0.02	**0.13**
dedup	100,600	175,868	**0**	183	7.9	**0.04**
ferret	32,499	49,483	**0**	59	21	**0.36**
fluidanimate	163,244	254,623	**0**	286	12	**0.04**
raytrace	28	41	**0**	0.02	0.01	**0.50**
streamcluster	8,699	11,048	**0**	23	1.2	**0.05**
swaptions	1,066	1,459	**0**	1.7	0.16	**0.09**
vips	10,005	13,098	**0**	8.9	1.1	**0.11**
x264	735	920	**0**	0.63	0.09	**0.14**
aget	65	94	**0**	0.13	0.02	**0.15**
pbzip2	5,322	7,432	**0**	5.61	0.72	**0.12**
httpd	5,689	8,429	**0**	0.53	0.04	**0.07**
mysql	23,012	33,586	**0**	5.88	0.68	**0.11**
Total	429,768	693,343	**0**	720.61	54.95	-
Mean	-	-	-	-	-	**0.15**

Note: "Time on Elimination" represents the time spent on lockset comparison and removing redundant contexts for ML, and comparing LRC for HL+, respectively.

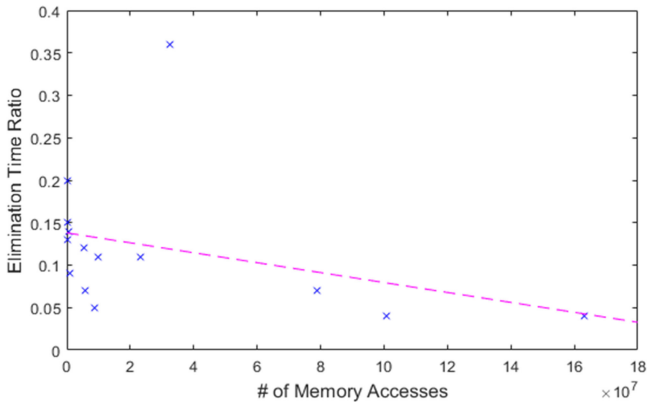

Fig. 4. Correlation between elimination time ratio and the number of memory accesses. The time ratio on eliminating skippable memory accesses will be reduced as the number of memory accesses increases.

skippable memory accesses achieved by MultiLock-HB and HistLock+. Column "Memory Accesses" shows the number of memory accesses in each benchmark. For column "LS Comp.," we compared the number of lockset comparisons performed by the elimination strategies of MultiLock-HB and HistLock+, respectively.

From those two columns of Table II, on average, MultiLock-HB needed to make 1.5 lockset comparisons to handle one access. On the other hand, by design, HistLock+ did not need any lockset comparison on a similar elimination process.

In the column "Time on Elimination," subcolumn (*A*) shows the amount of time MultiLock-HB spent to eliminate skippable memory accesses, and subcolumn (*B*) shows the time HistLock+ spent to do its elimination. From the column "*B* / *A*," on average, HistLock+ spent 15% of the time to complete the elimination operation compared to MultiLock-HB.

We have also analyzed the correlation between this time ratio and the number of memory accesses and plotted it as shown in Fig. 4. As the plot shows, there is a negative correlation between these two factors. It seems indicating that HistLock+ will show more advantage over MultiLock-HB when applying it to analyze program executions having more access events.

TABLE III
SLOWDOWN FACTOR AND MEMORY CONSUMPTION

Benchmark	Base Time (sec)	Slowdown Factor			Memory Consumption (in MB)		
		*HL**	*ML#*	*HL+#*	*HL**	*ML#*	*HL+#*
blackscholes	0.5	0.80	0.83	**0.81**	96	95	**105**
bodytrack	39	0.64	6.22	**1.38**	2,402	4,400	**2,519**
canneal	1.1	0.18	0.27	**0.64**	130	155	**135**
dedup	33	1.58	11.32	**1.88**	2,797	2,813	**2,776**
ferret	12	3.17	10.25	**6.33**	1,109	2,147	**1186**
fluidanimate	69	1.51	6.38	**2.14**	6,596	7,764	**7,053**
raytrace	91	0.24	0.22	**0.15**	187	186	**199**
streamcluster	31	4.51	7.19	**5.23**	3,062	3,185	**3,256**
swaptions	0.8	0.63	3.75	**1.63**	171	221	**186**
vips	6.2	0.42	2.48	**1.08**	770	1,671	**799**
x264	2.2	0.27	0.51	**0.91**	303	468	**326**
aget	0.7	0.31	0.43	**0.29**	102	102	**102**
pbzip2	3.2	0.53	2.94	**0.41**	529	1,266	**536**
httpd	2.9	0.38	0.38	**0.38**	176	193	**179**
mysql	8.0	1.75	3.13	**2.13**	1,522	2,603	**2,001**
Mean	-	1.13	3.75	**1.69**	-	-	-
Total	-	-	-	-	19,952	27,269	**21,358**

**: incomplete hybrid race detector; #: complete hybrid race detector.*

2) Execution Time and Runtime Slowdown: Table III summarizes the result on comparing the execution time (i.e., the elapsed real time, measured by *time -e* command in Linux) of HistLock, MultiLock-HB, and HistLock+. The "Base" time was the execution time (in second) that each benchmark ran on Maple with empty instrumentation so that the runtime overhead caused by Maple itself can be factored out. The formula below calculated the slowdown factor:

$Slowdown\ factor = (T - Base)\ /\ Base$, where T is the execution time of each detector.

From the row "Mean" in Table III, HistLock, MultiLock-HB, and HistLock+ incurred mean slowdowns of 1.13×, 3.75×, and 1.69×, respectively. HistLock and HistLock+ were significantly faster than MultiLock-HB by 232% and 122%. Moreover, since the slowdown factor of MultiLock-HB is not bad, and that of HistLock+ is half of it. Hence, if we deem the slowdown of MultiLock-HB practical, HistLock+ can do a better job.

Based on Table II, the overheads of MultiLock-HB and HistLock+ on skippable memory access elimination were 1.75× and 0.19×, respectively, and the overheads incurred by MultiLock-HB and HistLock+ on other operations (e.g., race checking) are 2.0x and 1.5x, respectively. It indicates that MultiLock-HB spent 46% of its relative overhead in memory access elimination, whereas HistLock+ only spent 11%.

The above analysis indicates that making race checking more efficient could be the next research objective to further reduce the performance overheads of hybrid race detectors.[2]

[2]For example, a line of research similar to the use of LRC in HistLock+ is to check the *number* of locks protecting a memory location rather than checking the actual lockset is interesting (albeit incurring false positive issues) [35]. Another potential line of research is to reduce the number of race checking performed by a detector. For instance, some interesting sampling techniques (e.g, [3], [19], [40]) may result in high probability to expose a race with limited race checking budget (but the notion of completeness is compromised).

TABLE IV
RACE DETECTION EFFECTIVENESS

Benchmark	# of HB-Races	# of HB-Races Per Run				# of Racy EL-Span	Racy EL-Span to Ø-Race Ratio			Racy EL-Span to Racy Statement Ratio		
		*FT^*	*HL**	*ML#*	*HL+#*		*HL**	*ML#*	*HL+#*	*HL**	*ML#*	*HL+#*
blackscholes	0	0	0	0	**0**	0	-	-	-	-	-	-
bodytrack	11	7	9	11	**11**	38	†0.47	0.50	**0.61**	0.98	1	**1**
canneal	0	0	0	0	**0**	0	-	-	-	-	-	-
dedup	0	0	0	0	**0**	27	†0.41	0.70	**0.70**	1	1	**1**
ferret	4	4	4	4	**4**	10	†0.40	0.50	**0.50**	1	1	**1**
fluidanimate	36	20	33	36	**36**	112	0.51	0.54	**0.54**	0.86	1	**1**
raytrace	1	1	1	1	**1**	2	0.50	0.50	**0.50**	1	1	**1**
streamcluster	117	91	95	98	**102**	266	0.52	0.63	**0.67**	0.83	0.89	**1**
swaptions	0	0	0	0	**0**	0	-	-	-	-	-	-
vips	0	0	0	0	**0**	44	†0.39	0.57	**0.59**	1	1	**1**
x264	0	0	0	0	**0**	24	0.50	0.50	**0.54**	1	1	**1**
aget	4	1	4	4	**4**	12	0.50	0.50	**0.50**	1	1	**1**
pbzip2	10	6	9	9	**9**	42	†0.39	†0.43	**0.55**	0.88	1	**1**
httpd	32	18	21	23	**23**	50	†0.44	†0.46	**0.50**	0.92	1	**1**
mysql	197	118	138	142	**145**	322	†0.55	0.69	**0.72**	0.96	0.98	**1**
Total	412	266	314	328	**335**	949	-	-	-	-	-	-
Mean	-	-	-	-	-	-	0.47	0.54	**0.58**	0.95	0.99	**1**

†: *at least one racy EL-Span is unable to be identified;* ^: *happens-before race detector;* *: *incomplete hybrid race detector;* #: *complete hybrid race detector.*

3) Memory Consumption: Table III also summarizes the memory consumption (i.e., *maximum* resident set size, measured by *time* $-M$ command in Linux) of HistLock, MultiLock-HB, and HistLock+. From the row "Total," we observe that HistLock+ consumed 28% less memory than MultiLock-HB. In a closer look, HistLock+ incurred significantly lower memory overhead than MultiLock-HB in 6 out of 15 benchmarks (i.e., *bodytrack*, *ferret*, *vips*, *x264*, *pbzip2*, and *mysql*), and HistLock+ and MultiLock-HB incurred similar memory overheads in the remaining benchmarks (i.e., *blackscholes*, *canneal*, *dedup*, *fluidanimate*, *raytrace*, *streamcluster*, *swaptions*, *aget*, and *httpd*). Moreover, HistLock+ only incur neglectable (7%) memory overhead than the incomplete race detector HistLock, which is interesting.

The result indicates that the aggressive memory access removal strategy employed by MultiLock-HB introduces heavier memory consumption than the more lightweight memory access skipping strategy employed by HistLock+, although HistLock+ may keep more memory accesses in its analysis state. One reason is that MultiLock-HB must use a set-based data structure with heavier memory overhead than the list-based data structure used by HistLock+ (i.e., *set* versus *list*) to keep *contexts* for realizing its algorithm. If there is no lock subset relation among the memory accesses in the same epoch (on mhb relation), which is a special case, MultiLock-HB will keep the same contexts as HistLock+ kept. As such, MultiLock-HB tends to consume more memory than HistLock+ in this aspect. Another cause is that MultiLock-HB additionally used map-based data structure to map memory accesses generated by different threads to different sets while HistLock+ needed no such mapping (see Section III), which further enlarge the difference on memory consumption.

4) HB-Race Detection Effectiveness: Table IV shows the race detection effectiveness of each detector. Since the notion of HB-race is widely recognized in both academia and industry, we evaluated HistLock+ for its HB-race detection effectiveness.

All three hybrid detectors (HistLock, MultiLock-HB, and HistLock+) only detect Ø-race and cannot distinguish HB-races from the reported Ø-races. According to [29], all the races detected by each studied technique can be exposed by executing the same benchmark 10 000 times. Therefore, we ran FastTrack on each benchmark 10 000 times to approximate the full set of HB-races that can be detected (column "# of HB-races") from the benchmark. Then, we recorded the sets of HB-races reported by each detector on each execution of the benchmark by inspecting this approximated full set. Column "# of HB-races Per Run" shows the mean number of HB-races detected in single run averaged over 100 times.

From Table IV, the three hybrid race detectors reported more HB-race than FastTrack per run in 7 out of 9 benchmarks (i.e., *bodytrack*, *fluidanimate*, *streamcluster*, *aget*, *pbzip2*, *httpd*, and *mysql*) that contain HB-race(s). In total, HistLock, MultiLock-HB, and HistLock+ reported more HB-races per run than FastTrack by 18%, 23%, and 26%, respectively. Among all three hybrid detectors, HistLock+ reported the most HB-races. HistLock+ reported 21 more HB-races than its incomplete counterpart HistLock. It confirmed that the use of complete hybrid detection is advantageous.

We computed the probabilities $P(r_{id}|\text{FT})$, $P(r_{id}|\text{HL})$, $P(r_{id}|\text{ML})$, and $P(r_{id}|\text{HL+})$ of the same HB-race, denoted as r_{id}, detected by FastTrack, HistLock, MultiLock-HB, and HistLock+ per run over 100 times, respectively, where id is a unique HB-race ID. Then, we computed the Δ values (e.g., $P(r_{id}|\text{HL+}) - P(r_{id}|FT)$) between HistLock+ and each other detector on all the 412 HB-races found in the entire experiment.

Fig. 5 shows the Δ values of HistLock+ compared with FastTrack, HistLock, and MultiLock-HB on each HB-race in descending order of Δ value. For ease of presentation, we indexed the HB-race ID according to the sorted Δ values for each pair of detectors.

From the figure, HistLock+ gained significant improvements (more than 30%) on the detection probability for some HB-races compared to FastTrack. In certain cases, the Δ values between HistLock+ and FastTrack were over 50% (and the highest value is 73%). It indicates that some HB-races which cannot be de-

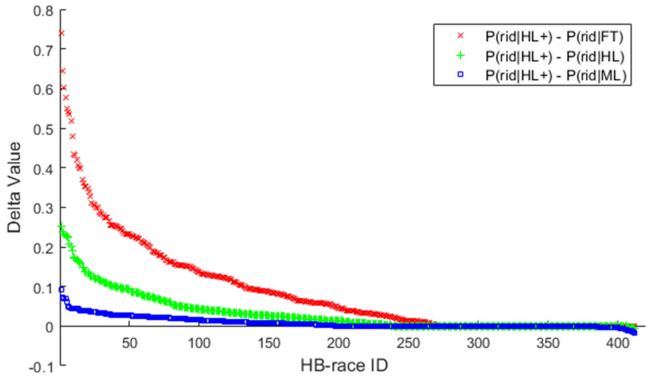

Fig. 5. Sorted Δ values of detection probability on each HB-race between HistLock+ and the other three race detectors. HB-race ID is labeled according to the sorted Δ values.

tected in high probability by pure happens-before detectors in many runs (i.e., hidden HB-race [6]) can be predicted by hybrid race detectors with high probability in a single run.

HistLock+ also outperformed HistLock by achieving higher probability on detecting some HB-races. It is intuitive because HistLock is an incomplete hybrid detector while HistLock+ is complete.

On the other hand, HistLock+ only shows small advantage on detection probability when compared to MultiLock-HB. This is understandable because both of them are designed as complete detectors, but HistLock+ has a finer notion (EL-span) in dividing the same thread segment of code than MultiLock-HB (which is based on epoch only), which a more detailed analysis is presented in the next Section V-C.5.

5) Ø-Race Detection Effectiveness: EL-span is the unit HistLock+ used to capture racy memory accesses, and there is no similar notions used in HistLock and MultiLock-HB. We have analyzed the Ø-races detected by HistLock, MultiLock-HB, and HistLock+ to evaluate their effectiveness of identifying racy EL-spans.

We mapped each memory access involved in the reported Ø-race to its corresponding EL-span. In Table IV, column "# of Racy EL-span" shows the total number of distinct racy EL-spans identified by any of the three detectors in 100 runs.

Column "Racy EL-span to Ø-race Ratio" (Ø-race ratio for short) shows the mean number of Ø-race in each racy EL-span over these 100 runs. Note that each race involves two EL-spans, a value smaller than 0.5 indicates that some Ø-races were missed by one detector but reported by other detector(s) in some racy EL-spans; whereas a value higher than 0.5 indicates that some EL-spans were involved in more than one Ø-race in some runs.

We then annotated each cell by "†" before its value if there is at least one racy EL-span unable to be identified by the corresponding detector as racy. We found that on 7 and 2 benchmarks, HistLock and MultiLock-HB missed some racy EL-spans, respectively. The results indicate that HistLock could not replace HistLock+ to identify the set of racy EL-spans and MultiLock-HB could be a good approximator.

Column "Racy EL-span to Racy Statement Ratio" (racy statement ratio for short) shows the mean number of racy statements in each racy EL-span over 100 runs. We calculated the racy statement ratio for READs and WRITEs separately, then we took the mean of both results.

On *blackscholes*, *canneal*, and *swaptions*, all three hybrid race detectors reported no Ø-race. From the column, the racy statement ratio of HistLock+ retains as 1 on every benchmark, which is consistent with our presented theory that HistLock+ always keeps the first racy memory access belonging to each racy EL-span, and thus, it always reports the same racy statement in every racy EL-span even under different interleaving scenarios (see Theorem 2). The mean racy statement ratios of HistLock and MultiLock-HB are both less than one, confirming that HistLock and MultiLock-HB have no such guarantee.

In our experiment, HistLock+ reported both the superset of Ø-races and the superset of racy statements compared to those reported by MultiLock-HB. It is consistent with our algorithm presented in Section V because on every memory location, both HistLock+ and MultiLock-HB keep the memory accesses with minimal lockset in each thread segment they divide. Since HistLock+ employs a finer granularity of thread segment (i.e., EL-span) than that of MultiLock-HB (i.e., epoch on mhb relation), in general, HistLock+ keeps a superset of memory accesses than those kept in MultiLock-HB (recall the example in Section III).

Moreover, HistLock+ also reported the supersets of Ø-races and racy statements compared to those reported by HistLock, because HistLock+ keeps extra memory accesses that removed by HistLock through its function-based memory access removal strategy (recall the example in Section III).

VI. Further Discussion

A. Bugs Found

From the races detected in the experiment, we found many concurrency bugs that can be exposed from our race report.[3] For example, *bodytrack* uses a smart pointer to keep the references of an image object, but the lock protecting the update (i.e., increase and decrease) of its reference counter is commented out. As a result, the reference counter would be no longer correct when multiple threads update it, and the image object could be freed early, or a memory leak would occur. A race in *ferret* may produce incorrect output for printing the number of enqueued elements due to no protection on the shared counter. We also found races in *fluidanimate* and *aget* that lead to wrong results, and a race in *pbzip2* that may cause program crash. We next discuss three confirmed bugs in the real-would application exposed by our tool in detail.

Fig. 6 shows a simplified bug in Apache HTTP Server (Bug#21287). This bug contains one race on *obj->refcount* between statements $S2$ and $S3$. Statements $S1$ and $S3$ atomically decreases the reference counter (i.e., $\&obj{-}{>}\,refcount$) of cache object obj by 1, and obj will be cleaned if $\&obj{-}{>}\,refcount$ reaches 0. However, there is no proper synchronization among $S1, S2$, and $S3$. If $\&obj{-}{>}\,refcount$ reaches 1 after $S1$, and then reaches 0 after $S3$ but before $S2, obj$ could

[3] Available at http://www.cs.cityu.edu.hk/~wkchan/content/tools

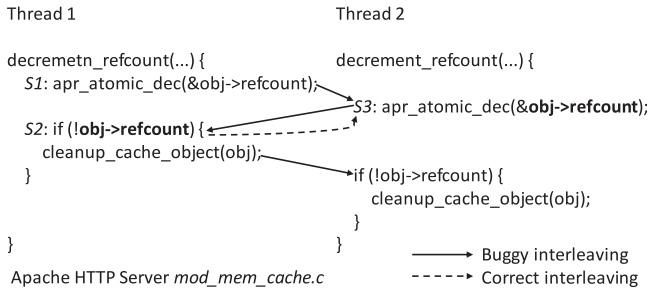

Fig. 6. Concurrency bug in Apache HTTP Server 2.0.48. There is a race on $obj{-}{>}\ refcount$ (between statement $S2$ and $S3$). The cache object obj could be freed twice.

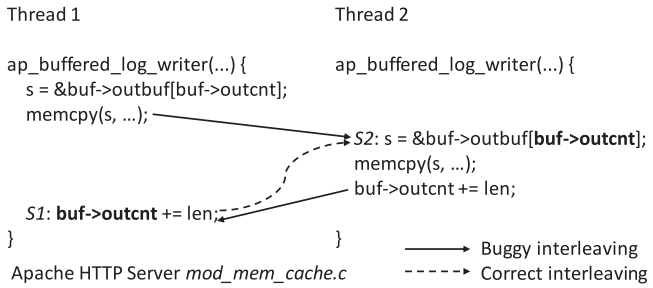

Fig. 7. Another concurrency bug in Apache HTTP Server 2.0.48. There is a race on $buf{-}{>}\ outcnt$ (between statement $S1$ and $S2$). The log buffer would be messed.

be freed twice. Note that there is no race between $S1$ and $S3$ because the decrease on $\&obj{-}{>}\ refcount$ is protected by a common lock. Interestingly, if we use a lock to only protect $S2$ and $S3$ from being a race, this bug still *cannot* be fixed even it is race free. The bug can be fixed by using a lock to ensure that $S1$ and $S2$ execute atomically.

Fig. 7 shows another simplified bug in Apache HTTP Server (Bug#25520). Similar to the previous case, this bug contains one race on $buf{-}{>}\ outcnt$ between statements $S1$ and $S2$. Function *ap_buffered_log_writer*(. . .) intends to get the address (i.e., $\&buf{-}{>}\ outbuf\ [buf{-}{>}\ outcnt]$) of the log buffer and to append the buffer, where $buf{-}{>}\ outcnt$ is the index to the end of the buffer. Because $S1$ and $S2$ are not synchronized, $S2$ could be executed before $S1$, that is, Thread 2 may append the buffer from the same address where Thread 1 append it, before Thread 1 finishes increasing $buf{-}{>}\ outcnt$. As a result, the log buffer will be messed. Note that this bug *cannot* be fixed by simply eliminating the race between $S1$ and $S2$. The bug can be fixed by making these operations execute atomically.

Fig. 8 shows a simplified bug in MySQL (Bug#791). This bug contains two races on *log_type* between statements $S1$ and $S2$, and between statements $S2$ and $S3$, respectively. Function *MYSQL_LOG::new_file*(. . .) closes the old log file and opens a new one. It temporarily sets *log_type* to *LOG_CLOSE* before opening the new file, and then set *log_type* to *save_log_type* after the new file is opened. Function *mysql_insert*(. . .) writes the log file when *log_type* is *not LOG_CLOSE*. Because $S1, S2$, and $S3$ are not synchronized properly, $S3$ could be executed in between $S1$ and $S2$, that is, Thread 2 may read the temporary

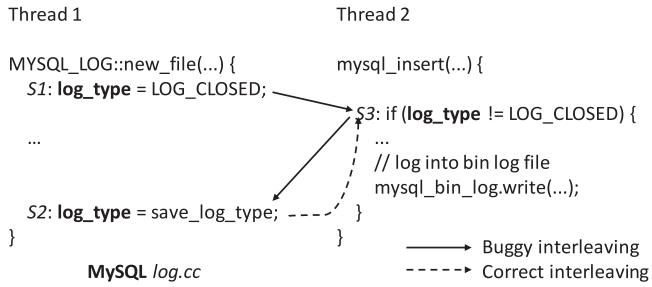

Fig. 8. Concurrency bug in MySQL 4.0.12. There are two races on *log_type* (between statement $S1$ and $S2$, and between statements $S2$ and $S3$, respectively). Some database actions could be lost in the log file.

LOG_CLOSE flag and skip writing the log file during Thread 1 opening a new log file. As a result, some database actions are lost in the log file. Eliminating the two races on *log_type* is insufficient to fix this bug. Like the above two bugs, this bug can be fixed by making statements from $S1$ to $S2$ execute atomically.

We note that all those bugs can be exposed by MultiLock-HB and HistLock+ (in 100 runs). It is understandable because both MultiLock-HB and HistLock+ are complete race detectors. However, HistLock+ is superior to MultiLock-HB in terms of runtime slowdown and memory consumption. On the other hand, FastTrack and HistLock were unable to expose all those bugs in 100 runs.

B. Benign Races

We also found that some races (either HB-races or $\emptyset$-races) are not necessarily harmful, while their occurrences do not compromise the correctness of the program. We classify those benign races into the following four patterns.

Write constant value: Some programs write a constant value to a shared memory location without synchronization, such as setting the same value multiple times to a flag variable. Those WRITE–WRITE races have no influence on the behavior of a program. For instance, *ferret* initializes the *input_end* flag to 0, and set this flag to 1 when finish loading a file. The program will do further process if any file is loaded. Multiple threads may load files and set the flag concurrently, but it incurs no bug in the program. Another case is that *pbzip2* initializes the *allDone* flag to 0, and set this flag to 1 when it completes or aborts the process. Multiple threads may set the flag to 1 concurrently without unintended behavior.

Ad hoc synchronization: Many programs implement their own synchronization primitives by using polling loop to periodically check a shared condition variable asynchronously updated by other threads (see Fig. 9). However, the race on the condition variable (i.e., *flag* in Fig. 9) in the abovementioned ad hoc synchronization usually introduce no bug. In our case, PARSEC implements its own barrier with polling loops, and the barrier is used by *ferret* and *fluidanimate*. Moreover, *pbzip2*, *httpd*, and *mysql* also implement their own ad hoc synchronizations with polling loops. Those races on the condition variables in their ad hoc synchronizations are benign.

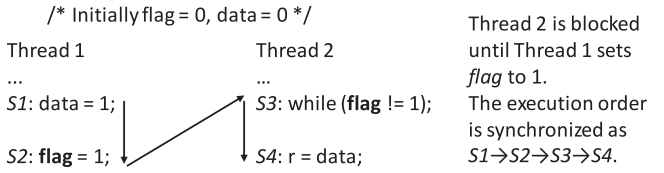

Fig. 9. Ad hoc synchronization implemented by polling loop on condition variable *flag*. There is a benign race on *flag* (between statements $S2$ and $S3$), and a false race on *data* (between statements $S1$ and $S4$).

Caching: Some programs use a shared memory location as cache, and update the cache concurrently. Since cache miss or keeping multiple copies in the cache is not regarded as incorrect in many cases, the synchronization for updating the cache is sometimes carefully removed to improve the efficiency. We found a race in *httpd* which is intentionally designed to boost the cache performance.

Unused value: An incorrect value (due to data race) is discarded and never used by some programs. We found *raytrace* will recompute the result of atomic addition if the value in register is different from the current value in memory for the same input parameter, and the race causing inconsistency between register and memory is harmless.

C. False Races

Some programmers implement their own synchronization mechanisms without using standard *pthreads* synchronization. However, our tool prototype used in the experiment is only aware of *pthreads* function calls. Therefore, those customized synchronization mechanisms cannot be well recognized if they are implemented by limited (or without) *pthreads* functions.

One case is that *streamcluster* employs PARSEC-implemented barrier, which only use *pthreads* mutex without *pthreads* barrier. As a result, our tool cannot instrument these barrier events, and most reported races (both HB-races and $\emptyset$-races) are false races in this benchmark. Similarly, MySQL implements its own mutex mechanisms which cannot be well handled by our tool; thus, it reported a large number of false races in *mysql*. We note that these kinds of false races are caused by the limitation of Maple framework but not our algorithm (i.e., HistLock+).

There are also some false races caused by ad hoc synchronization. As discussed in the previous section, ad hoc synchronization may cause benign races on the condition variable (i.e., *flag* in Fig. 9). Further, since no synchronization primitive protecting the shared memory location to be synchronized inside the polling loop, hybrid detectors will report a $\emptyset$-race on that shared memory location (i.e., *data* in Fig. 9). In our experiment, some additional false races were reported in *ferret*, *fluidanimate*, *pbzip2*, *httpd*, and *mysql* by this reason.

We note that the difference between the numbers of false races reported by MultiLock-HB and HistLock+ was insignificant. Since our work focuses on reducing lockset comparisons in hybrid detectors, we leave eliminating those false races as our future work.

D. Active Testing

As discussed above, a reported race in the experiment may be a benign race or a false race. Active testing (e.g., [25], [41]) can be used to complement race detection to confirm whether a reported race is real and harmful. For example, RaceFuzzer [25] takes the potential races reported by a race detector as input, and tries to schedule the interleaving to trigger the race and checks whether any generic error (e.g., crash, hang, exception, etc.) can be observed after triggering the race. Because pure happens-before race detectors have limited coverage (i.e., thread interleaving sensitive), and pure lockset-based detectors may report massive false positives and false races [24], it appears to us that hybrid race detectors are promising by reporting $\emptyset$-races as the input of race-guided active testing to confirm those harmful races.

E. Long Traces

As an online race detector, HistLock+ may still run out of memory when analyzing long trace due to the increasing number of memory accesses to be kept for retaining completeness. One heuristic strategy to alleviate the issue is to set up a size window for each list of memory accesses and remove those older memory access kept in the analysis state if all the slots within the size window have been occupied. This class of strategies is based on the empirical findings that memory accesses have higher possibility to be in race with more recent accesses than older ones in the history [18]. Another class of strategy is to clean up the history access events when certain triggering conditions (e.g., different function calls [34], thread context switching, reaching the threshold of targeted events, etc.) are met. For instance, sampling techniques (e.g., [3], [19], [40]) can be employed to reduce the memory consumption. However, all above approaches compromise the completeness of hybrid detectors. It is interesting to know how far a complete (online) race detector can handle long traces.

VII. Related Work

Eraser [24] is a pure lockset-based race detector. It introduced a finite state machine and refined the lockset algorithm. There are four states for every shared memory location: *virgin*, *exclusive*, *shared*, and *shared-modified*. Eraser updates the lockset and state accordingly and reports a race warning if a LD violation occurred in *shared-modified* state. Eraser sometimes reports warnings that are not real races. It may also miss to detect race in certain situations (i.e., false negatives).

FastTrack [10] is a pure happens-before race detector. It referred to a pair of a clock c and a thread t as an epoch. Because FastTrack uses epoch to record the last WRITE performed by last thread, it is only able to detect the latest WRITE–READ race between the adjacent WRITE and READ events and may miss races. Also, owing to the thread interleaving sensitive nature of happens-before detection, FastTrack cannot predict potential races from the given trace.

Before the inception of FastTrack, there were several hybrid race detectors attempting to combine the lockset and HB

mechanisms for race detection [14], [21], [22], [27], [39]. After the concept of epoch introduced by FastTrack, there are several hybrid race detectors employing the idea of epoch along with novel lockset algorithms to reduce the false positives or improve the performance of previous hybrid race detectors. AccuLock [32] is the first epoch-based hybrid race detector, but AccuLock is incomplete and incurs the problem of reporting race warning that is not Ø-races. MultiLock-HB [33] is the first complete epoch-based hybrid race detector. It incurs significantly heavier runtime overhead than AccuLock. HistLock [34] is a sound hybrid race detector. Although it has higher coverage than AccuLock by keeping more historical events, HistLock is still incomplete, otherwise it will degenerate into an impractical detector.

SimpleLock [35] is an interesting attempt, which aims at lowering the overhead of hybrid race detectors at the expense of soundness. Compared to MultiLock-HB, SimpleLock only keeps the number of locks being held by a thread rather than the identity of each of those locks. It is efficient but cannot be generalized to correctly detect LD violations involving multiple locks. SimpleLock+ [36] simplifies SimpleLock by only recording whether the last memory access to a memory location has a lock to protect. SimpleLock+ is more lightweight than SimpleLock, but it does not address the soundness problem incurred by SimpleLock.

Some techniques aim at inferring alternative and valid interleaving scenarios from a given trace to detect the additional races (without false positives) in these alternative scenarios. *Causally-precedes* (CP) [28] is a refinement of the happens-before relation, which has been applied to detect a small superset of HB-races from nondeadlocking trace fragment in polynomial time. Some CP-based race detectors [15], [23] are recently proposed which reduce the time complexity of CP detection from polynomial to linear. RVPredict [13] refines the maximal causal model [26] with additional value constraints to ensure branch decisions correctly encoded in detecting additional races from trace fragments. The huge overhead to analyze a long trace prevents this technique to be general online detector.

Redundancy elimination techniques aim to simplify the tracking of causal orders among events in a trace. LOFT [5] safely skips operations on vector clock tracking when processing a noninterleaved sequence of lock acquisitions and releases performed by each thread in a trace, and has been applied to a happens-before race detector. Interference-free region (IFR) [8] is a region of a trace within which shared variables cannot be updated by other threads. IFRit [8] detects the overlap of such regions accessing the same shared memory location by different threads. RedCard [12] proposes the idea to keep the first access to a shared memory location in a sequence of events that involves no lock release, and treats all other events in the sequence as redundant. It statically identifies those redundant accesses, which lack of the runtime information (such as epoch) from the execution trace, and need a dynamic detector to do further detection. Unlike RedCard, HistLock+ dynamically identifies skippable memory accesses from a trace, and divides the sequence (i.e., EL-span) by the combination of epoch and LRC. Thus, HistLock+ achieves both redundancy elimination and race detection in single phase while guarantees no loss in detection precision.

VIII. Conclusion

This paper has investigated to what extent a complete hybrid race detector achieves without lockset comparison in memory access maintenance. It has presented HistLock+ for this purpose. HistLock+ divides the sequence of memory access events generated by the same thread into thread segments based on each combination of epoch and LRC. Moreover, for each memory location, HistLock+ discards all but the first WRITE in every such thread segment if no READ precedes this first WRITE; otherwise, it additionally keeps the first READ. The design of HistLock+ makes the elimination on skippable memory accesses involving no lockset comparison and conserves memory in maintaining those events being kept. HistLock+ guarantees to detect at least one Ø-race, if any, on each memory location, and never report false positive of Ø-race. It reports *exactly one* racy memory access for either READ or WRITE on each memory location at the EL-span level. In the experiment, HistLock+ was found to be 122% faster and 28% more memory-efficient than the previous state-of-the-art complete hybrid race detector. Moreover, HistLock+ achieved the highest effectiveness in race detection among all evaluated race detectors.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for java," *ACM Trans. Programm. Lang. Syst.*, vol. 28, no. 2, pp. 207–255, 2006.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 72–81.

[3] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: Proportional detection of data races," in *Proc. 31st ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2010, pp. 255–268.

[4] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-up: A complete and automatic linearizability checker," in *Proc. 31st ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2010, pp. 330–340.

[5] Y. Cai and W. K. Chan, "Lock trace reduction for multithreaded programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 12, pp. 2407–2417, Dec. 2013.

[6] Y. Cai, and L. Cao, "Effective and precise dynamic detection of hidden races for Java programs," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 450–461.

[7] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, 1998.

[8] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H. J. Boehm, "IFRit: Interference-free regions for dynamic data-race detection," in *Proc. ACM Int. Conf. Object Oriented Programm. Syst. Lang. Appl.*, 2012, pp. 467–484.

[9] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *Proc. 19th ACM Symp. Oper. Syst. Principles*, 2003, pp. 237–25.

[10] C. Flanagan and S. N. Freund, "FastTrack: Efficient and precise dynamic race detection," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2009, pp. 121–133.

[11] C. Flanagan and S. N. Freund, "Type-based race detection for Java," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2000, pp. 219–232.

[12] C. Flanagan and S. N. Freund, "Redcard: Redundant check elimination for dynamic race detectors," in *Proc. Eur. Conf. Object-Oriented Programm.*, 2013, pp. 255–280.

[13] J. Huang, P. O. N. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2014, pp. 337–348.

[14] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2009, pp. 1–13.
[15] D. Kini, U. Mathur, and M. Viswanathan, "Dynamic race prediction in linear time," in *Proc. 38th ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2017, pp. 157–170.
[16] Z. Lai, S. Cheung, and W. K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 235–244.
[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[18] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Archit. Support Programm. Lang. Oper. Syst.*, 2008, pp. 329–339.
[19] D. Marino, M. Musuvathi, and S. Narayanasamy. "LiteRace: Effective sampling for lightweight data-race detection," in *Proc. 30th ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2009, pp. 134–143.
[20] F. Mattern, "Virtual time and global states of distributed systems," *Parallel Distrib. Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.
[21] R. O'Callahan and J. D. Choi, "Hybrid dynamic data race detection," in *Proc. 9th ACM Symp. Principles Practice Parallel Programm.*, 2003, pp. 167–178.
[22] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs," in *Proc. 9th ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2003, pp. 179–190.
[23] J. Roemer and M. D. Bond, "An online dynamic analysis for sound predictive data race detection," Ohio State University, Columbus, OH, USA, Tech. Rep. OSU-CISRC-11/16-TR05, 2016.
[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.
[25] K. Sen, "Race directed random testing of concurrent programs," in *Proc. 29th ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, 2008, pp. 11–21.
[26] T. F. Serbanuta, F. Chen, and G. Rosu, "Maximal causal models for sequentially consistent systems," in *Proc. Int. Conf. Runtime Verification*, pp. 136–150, 2013.
[27] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proc. Workshop Binary Instrum. Appl.*, 2009, pp. 62–71.
[28] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proc. 39th Annu. ACM SIGPLAN-SIGACT Symp. Principles Programm. Lang.*, 2012, pp. 387–400.
[29] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using schedule bounding: An empirical study," in *Proc. 19th ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2014, pp. 15–28.
[30] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static race detection on millions of lines of code," in *Proc. Joint Meeting Found. Softw. Eng.*, 2007, pp. 205–214.
[31] S. Wu, C. Yang, C. Jia, and W. K. Chan, "ASP: Abstraction subspace partitioning for detection of atomicity violations with an empirical study," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 724–734, Mar. 2016.
[32] X. Xie and J. Xue, "AccuLock: Accurate and efficient detection of data races," in *Proc. 9th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2011, pp. 201–212.
[33] X. Xie, J. Xue, and J. Zhang, "AccuLock: Accurate and efficient detection of data races," *Softw., Practice Experience*, vol. 43, no. 5, pp. 543–576, 2013.
[34] J. Yang, C. Yang, and W.K. Chan, "HistLock: Efficient and sound hybrid detection of hidden predictive data races with functional contexts," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Security.*, 2016, pp. 13–24.
[35] M. Yu, S. K. Yoo, and D. H. Bae, "Simplelock: Fast and accurate hybrid data race detector," in *Proc. Int. Conf. Parallel Distrib. Comput.*, 2013, pp. 50–56.
[36] M. Yu and D. H. Bae, "SimpleLock+: Fast and accurate hybrid data race detection," *Comput. J.*, vol. 59, no. 6, pp. 793–809, 2016.
[37] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 325–336.
[38] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proc. ACM Int. Conf. Object Oriented Programm. Syst. Lang. Appl.*, 2012, pp. 485–502.
[39] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: Efficient detection of data race conditions via adaptive tracking," in *Proc. 20th ACM Symp. Oper. Syst. Principles*, 2005, pp. 221–234.
[40] K. Zhai, B. Xu, W. K. Chan, and T. H. Tse, "CARISMA: A context-sensitive approach to race-condition sample-instance selection for multithreaded applications," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 221–231.
[41] W. Zhang, C. Sun, and S. Lu, "ConMem: Detecting severe concurrency bugs through an effect-oriented approach," in *Proc. 15th Edition ASPLOS Archit. Support Programm. Lang. Oper. Syst.*, 2010, pp. 179–192.

Jialin Yang received the B.Eng. degree in software engineering from the School of Software, Nanchang University, China, in 2012. He received the M.Sc. degree in computer and information science from SUNY Polytechnic Institute, Albany, NY, USA, in 2014. He is currently working toward the Ph.D. degree with the Department of Computer Science, City University of Hong Kong, Hong Kong.

His current research interest includes dynamic program analysis on detection of concurrency bugs in multithreaded programs.

Bo Jiang (M'10) received the Ph.D. degree in software engineering from The University of Hong Kong, Hong Kong in 2011.

He is currently an Associate Professor with the School of Computer Science and Engineering, Beihang University, Beijing, China. His current research interests include software engineering in general, and embedded systems testing as well as program debugging.

Dr. Jiang received the best paper awards from COMPSAC'08, COMPSAC'09, QSIC'11, and QRS'17. His research results have been reported in many international journals and conferences such as TSC, JSS, IST, ASE, WWW, ICWS, COMPSAC, and QSIC.

W. K. Chan (M'04) received the B.Eng., M.Phil., and Ph.D. degrees from The University of Hong Kong, Hong Kong. He is an Associate Professor with the Department of Computer Science, City University of Hong Kong. His current research interests are program analysis and testing of large-scale software systems.

Prof. Chan is an Editor of the *Journal of Systems and Software*, general and program co-chairs of SETA of COMPSAC 2015-2017, the Area Chair of ICWS 2015–2016, Guest Editors of a few journal special issues as well as PC/RC members of many international conferences including ICSE, FSE, QSIC, COMPSAC, ICWS, ICSOC, and QRS. His research results have been reported in more than 100 major international journals and conferences including TOSEM, TSE, TPDS, TAAS, TSC, TRel, CACM, Computer, ICSE, FSE, ISSTA, ASE, and ICDCS.