HistLock: Efficient and Sound Hybrid Detection of Hidden Predictive Data Races with Functional Contexts [†]

Jialin Yang
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
jialiyang4@gapps.cityu.edu.hk

Chunbai Yang
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
chunbyang2@gapps.cityu.edu.hk

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

***Abstract*—State-of-the-art hybrid data race detectors combine the happens-before relation and the locking discipline to alleviate the imprecision problem incurred by lockset-based detector and the thread interleaving problem incurred by happens-before detectors. However, they incur high runtime overheads. In this paper, we present HistLock, a novel and sound hybrid dynamic race detector, which attains high precision, low slowdown and memory overheads, and thread insensitivity. It formulates a novel context-based strategy to phrase out non-redundant memory access events and check races to conserve both time and memory computation. It ensures each race warning to be the one violating the locking discipline in the original event history. Our experiment compared HistLock to FastTrack, AccuLock, and MultiLock-HB, which were a precise happens-before race detector, an imprecise hybrid race detector, and a precise hybrid race detector, respectively, on 13 benchmark subjects. HistLock was found to be of higher precision, 156% faster and 33% more memory-efficient than MultiLock-HB. It detected 59 more race warnings than AccuLock, attaining higher effectiveness, but ran slower by 43%. In most cases, in a single run, HistLock reported all the races detected by FastTrack in 100 runs.**

***Keywords*—Data Race Detection; Dynamic Analysis**

I. INTRODUCTION

A *data race* occurs when two threads in an execution trace access the same memory location without any protection of their access order by synchronization operations, and at least one of these accesses is a write [9]. Data race occurrences may lead to memory state corruptions or program crashes. The presence of data races may indicate the presence of other concurrency bugs, such as atomicity violations or linerizability errors in the program code [3][18].

Data race freedom is a correctness criterion applicable to analyze multithreaded programs. However, the thread interleaving space of a typical multithreaded program is often large, thread schedules to produce different execution traces are different, and yet data races usually occur only in some but not all traces even for the same input. They indicate a need of cost-effective approaches to data race detection.

There are two broad categories of program analysis techniques for data race detection. The first category is the *static* race detection techniques [1][7][10][29]. They provide a race detection coverage over the entire program by analyzing the program code to report race warnings. In general, these warnings can be real races or false alarms, and the latter is the vast majority. It is intricate to confirm real races automatically based on such warnings due to the absence of known inputs to trigger these races. The second category is dynamic race detection techniques [9][14][22][23][26][30]. They closely analyze the memory access and synchronization operations of observed traces, and are able to detect data races more precisely in the observed trace than static analysis techniques.

Almost all dynamic techniques (e.g., [9][23][30]) analyze traces without needing the whole trace generated beforehand. These techniques are differed by their incurred slowdown overhead, memory overhead cost, precision in detecting races. They can be broadly classified into three sub-categories: *happens-before* (HB) approach (e.g., [9]), *locking discipline* (LD) approach (e.g., [23]), and *hybrid* approach (e.g., [22]). Detectors based on the HB approach aim at sound race detection (i.e., every reported case is a real race) [9]. However, this HB approach is sensitive to thread interleaving, which makes the detectors blind to detect races only inferable from an observed trace. Techniques based on causal precede relation [27] or the maximal causal model [12] address this problem, but their time complexity is either polynomial [27] or exponential [12], and they still have difficulties to scale up well in handling long executions, which makes such improved HB detectors incurring significant slowdown overheads and difficult to be used as online detectors. Detectors based on the LD approach are efficient but imprecise (i.e., report false positives and miss some real races) [23]. They can efficiently detect some races originally hidden behind the HB relations [16] in the observed traces, whereas some of these races can also be detected by the above improved HB detectors but with a much higher slowdown overhead. Since they incur false positives, follow-up studies [24] to schedule execution to confirm a race reported by a LD detector to be a real race have been investigated.

If a race in an observed trace can be detected by an online HB detector, there is no need to use an online LD detector in parallel to detect the race again from the same trace. For ease of our presentation, we refer to a race detected by a HB detector in a given observed trace as a HB race (with respect to the trace). Thus, online LD detectors can focus on efficiently detecting race warnings that are *not* HB races, albeit imprecisely (i.e., discharging the detection of HB races to a HB detector and discharging the confirmation of other race warnings to an active

† This research is supported in part by the General Research Fund and Early Career Scheme of Research Grants Council of Hong Kong SAR (project numbers: 123512, 125113, 111313, 11201114, and 11200015).
‡ Correspondence author.

testing technique [24]). We refer to the use of both the LD and HB approaches in the same online detector as a hybrid detector.

Xie et al. [13][30] formulate a pair of interesting hybrid detectors: AccuLock and MultiLock-HB. Both detectors combine LD [23] and *epoch*-based HB [9] mechanisms (see Section III for more details). In theory, MultiLock-HB ensures to report at least one race warning on a memory location if some memory accesses violate LD and they does not form a hard (i.e., fork-join) synchronization order in the observed trace. It also never reports false positives with respect to the lockset discipline. AccuLock compromises the above theoretical guarantee. For each memory location, it merely keeps the latest historic read accesses of each thread and one historic write access among all threads, resulting in detecting fewer race warnings than MultiLock-HB but also incurring a lower slowdown overhead. Nonetheless, AccuLock may report false positives other than cases of LD violations.

This paper proposes HistLock, a novel hybrid online race detector. HistLock records histories of non-redundant memory accesses on each shared memory location for read and write accesses, respectively. It reports a warning if any memory access pair (i.e., read-write, write-read and write-write) within their corresponding histories are concurrent with each other and violate LD. Unlike MultiLock-HB which checks every memory access and then removes the redundant ones from the corresponding read or write list kept by it, HistLock only checks non-redundant memory accesses (and skip the redundant ones), puts these accesses into the read history or write history, and introduces a novel context-based strategy to adaptively update the history. All the above designs help HistLock to conserve memory consumption as well as improve checking efficiency, which makes HistLock feasible and effective. HistLock theoretically guarantees to report warnings on concurrent memory accesses that violate LD without any false positive (sound), but it is practically limited to a restrictive subset (incomplete). To the best of our knowledge, HistLock is the first work to formulate the notion of *access site* (AS) and *access context* (AC), and update the history based on those two notions (we refer to them as functional context) to achieve efficient concurrency bug detection.

We have implemented HistLock on Maple [17], and evaluated it on 13 benchmark subjects including two real-world applications. The experimental results showed that HistLock was more precise, 156% faster, and 33% more memory-efficient than MultiLock-HB. In most cases, by using a single run, HistLock reported the sum of all races detected by FastTrack in 100 runs. The experiment also confirmed that the efficiency gap between FastTrack (or AccuLock) and HistLock is significantly smaller than that between FastTrack (or AccuLock) and MultiLock-HB.

The main contribution of this paper is three-fold:

(1) This paper proposes a novel hybrid technique using functional contexts. The technique is efficient and effective with high race coverage.

(2) We show the feasibility of HistLock by implementing HistLock as a race detector for detecting data races in real-world C++ programs.

(3) It presents an experiment that validates HistLock, in multiple dimensions.

The rest of this paper is organized as follows: Section II reviews the preliminaries related to our work. Section III introduces a motivating example of our work. Section IV presents HistLock. Section V reports evaluation experiment about HistLock. Section VI discusses the closely related work. Section VII concludes this paper.

II. PRELIMINARIE

A. System Modeling

TABLE 1. THE MODEL OF OUR SYSTEM

Operation	$op := write(x) \mid read(x) \mid acquire(m) \mid release(m) \mid fork(u) \mid join(u) \mid wait(cv) \mid signal(cv) \mid barrier(b)$ $x \in$ Memory Location; $m \in$ Lock; $u \in$ Thread; $cv \in$ Condition Variable; $b \in$ Barrier
Event	$e := \langle t, c, op, f\rangle$ $t \in$ Thread; $c \in$ Clock; $op \in$ Operation; $f \in$ Function
Execution Trace	$\tau := e_1, e_2, e_3, \ldots, e_n$ $e \in$ Event

Table 1 shows our model of the system. A race detector monitors a set of events in a program execution, where each ***event*** *e* is a quadruple $\langle t, c, op, f\rangle$, in which *t* is the thread generating *e*, *c* is the timestamp of the thread generating *e*, *op* is the operation of *e* performed by *t*, and *f* is the function in which the *op*'s line of code resides. Each ***operation*** *op* has a type: *write*(*x*) and *read*(*x*) for write and read on memory location *x*, *acquire*(*m*) and *release*(*m*) for lock acquisition and release on lock *m*, *fork*(*u*) and *join*(*u*) for fork a thread *u* and join to *u*, *wait*(*cv*) and *signal*(*cv*) for wait and signal on a condition variable *cv*, and *barrier*(*b*) for a synchronization barrier *b*. We refer to the event of the first two operation types (i.e., *write*(*x*) and *read*(*x*)) as **memory access**, and refer to the event of the other seven operation types as **synchronization primitive**. Specially, we refer to the event of the last five operation types (i.e., *fork*(*u*), *join*(*u*), *wait*(*cv*), *signal*(*cv*), and *barrier*(*b*)) as **hard order synchronization primitive**, because they impose a hard order of event execution. An ***execution trace*** τ (or ***trace*** τ for short) is a sequence of events $e_1, e_2, e_3, \ldots, e_n$.

B. Happens-Before Relation and HB-Race

The ***happens-before*** (**HB**) relation [16], denoted by $\rightarrow_{hb}$, is a partial order of the events over a trace τ. It is defined by three rules: (1) If α and β are events generated by the same thread, and α precedes β, then $\alpha \rightarrow_{hb} \beta$. (2) If α and β are events generated by different threads, and at least one of α and β is a synchronization primitive, and α precedes β, then $\alpha \rightarrow_{hb} \beta$. (3) If $\alpha \rightarrow_{hb} \beta$ and $\beta \rightarrow_{hb} \gamma$, then $\alpha \rightarrow_{hb} \gamma$.

When two events α and β access the same memory location in a given trace, and neither $\alpha \rightarrow_{hb} \beta$ nor $\beta \rightarrow_{hb} \alpha$, then α and β are ***concurrent***. If they are concurrent and at least one of them is a write, then α and β form a **HB-race** [9]. HB-race is a *subset* of all the real races existing in a program [12].

The ***must happens-before*** (or ***mhb***) relation [12][13][15][30], denoted by $\Rightarrow_{\mathbf{mhb}}$, is a special kind of happens-before relation over a trace τ. It is defined as follows: (1) If α and β are two events generated by the same thread, and α precedes β, then $\alpha \Rightarrow_{\text{mhb}} \beta$. (2) If a thread *t* forks a thread *u* via event *α*, and β is the first event of *u*, then $\alpha \Rightarrow_{\text{mhb}} \beta$; similarly, if a thread *t* joins to a thread *u* via event *α*, and β is the last event of *t*, then $\alpha \Rightarrow_{\text{mhb}} \beta$. (3) If a thread *t* sends a signal via α, and a thread *u* was blocked at β until it receives the signal sent by α, then $\alpha \Rightarrow_{\text{mhb}} \beta$. (4) If α is the last event before a barrier in a thread *t*, and β is the first event after the same barrier in a thread *u*, then $\alpha \Rightarrow_{\text{mhb}} \beta$. (5) If $\alpha \Rightarrow_{\text{mhb}} \beta$ and $\beta \Rightarrow_{\text{mhb}} \gamma$, then $\alpha \Rightarrow_{\text{mhb}} \gamma$.

C. Locking Discipline and ∅-Race

If a thread *t* acquires a lock *m*, then *t* holds *m* until it releases *m*. We refer to the set of locks (i.e., *lockset*) being currently held by a thread *t* as $\boldsymbol{L_t}$. We also denote the lockset of a thread when it generates an event *e* by $\boldsymbol{L(e)}$.

The ***locking discipline*** **(LD)** [23] is an assertion: For every memory location *x*, there is a *non-empty* lockset CL_x such that $CL_x \subseteq L_t$ whenever thread *t* accesses *x*, and all threads share the same CL_x for the same location *x*.

For an online detector like Eraser [23], 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. On an event *e* generated by a thread *t* to access location *x*, the detector reduces CL_x to $CL_x \cap L_t$ followed by checking whether CL_x is empty (denoted by ∅). It reports a LD violation if the reduced CL_x is empty [23]. For ease of reference, we refer to this resultant **reduced lockset** on handling event *e* as ***ls*(*e*)**.

Suppose that in a given trace, two events α and β access the same shared memory location with α preceding β, and at least one is a write. Then, α and β form an empty-set race (denoted by **∅-race**) [13][30] if the following conditions are satisfied: (1) neither $\alpha \Rightarrow_{\text{mhb}} \beta$ nor $\beta \Rightarrow_{\text{mhb}} \alpha$, and (2) a LD violation occurs on β.

Since the inception of Eraser, a vast majority of detectors based on the locking discipline approach detect ∅-races between two consecutive memory accesses performed on the same memory location along a trace.

The set of ∅-races is a *superset* of HB-races in a program [13][30]. As introduced in Section I, there is another line of research to apply active testing (e.g., [8][24]) to detect whether an ∅-race is harmful.

D. Vector Clock and Epoch

A ***vector clock*** **(VC)** [19] is a tuple of timestamps to record a clock for an entity (e.g. thread). We refer to the vector clock being currently held by a thread *t* as $\boldsymbol{C_t}$. We also denote the vector clock of a thread when it generates an event *e* by $\boldsymbol{C(e)}$. The timestamp of a thread *u* that is aware by thread *t* in its vector clock C_t is denoted by $C_t[u]$. We denote the vector clock in which every timestamp is 0 by $\bot_{VC}$. When a thread starts, its vector clock is $\bot_{VC}$. We use $C_{t1} \sqcup C_{t2}$ to indicate a vector clock C_{t3} in which the timestamp $C_{t3}[u]$ is $max(C_{t1}[u], C_{t2}[u])$ for each thread *u*, and we refer to ⊔ as vector clock **join** operation.

FastTrack [9] uses the notation ***c@t*** to denote a pair of a timestamp *c* and a thread *t*, where $c = C_t[t]$, and refers to *c@t* as ***epoch*** [9]. We refer to the current *epoch* of thread *t* as $\boldsymbol{E_t}$. We also denote the *epoch* at the point an event *e* is generated by ***E*(*e*)**. Similar to prior work (e.g., [9][13][30]), we use the notation $\boldsymbol{c@t \preccurlyeq C_t}$ to denote $c \leq C_t[t]$ (or $\not\preccurlyeq$ otherwise).

E. Redundant Memory Accesses

Suppose that in a trace τ, three events α, β and χ access the same shared memory location *x* such that (1) α and β are performed by the same thread t_1 and χ is performed by another thread t_2, (2) α precedes β in τ, and β precedes χ in τ, and (3) if χ is in race with β whenever χ is in race with α; then, β is called ***redundant*** [11][21] with respect to α and the race detection on β with χ can be *skipped*.

III. MOTIVATING EXAMPLE

We use the example shown in Fig. 1 to motivate our work. Fig. 1 shows an execution trace τ_1. The trace consists of 3 threads (t_1 to t_3), 4 functions (f_1 to f_4), and 25 events labeled as $e_1, e_2, \ldots, e_{25}$. Functions f_1 to f_4 generate event sequences e_1–e_5, e_6–e_{11}, e_{12}–e_{17}, and e_{18}–e_{24}, respectively. Among these events, 11 of them access the shared memory location *x*. In this section, we only consider non-redundant memory accesses unless specified otherwise.

The trace τ_1 executes events as follows: Thread t_1 forks t_2 and t_3. Then, t_1 invokes f_1 and f_1 returns, and then t_1 invokes f_2 and f_2 returns. After that, t_2 invokes f_3 and f_3 returns. Finally, t_3 invokes f_4 and f_4 returns. For brevity, we simply refer to τ_1 as $f_1 \to f_2 \to f_3 \to f_4$.

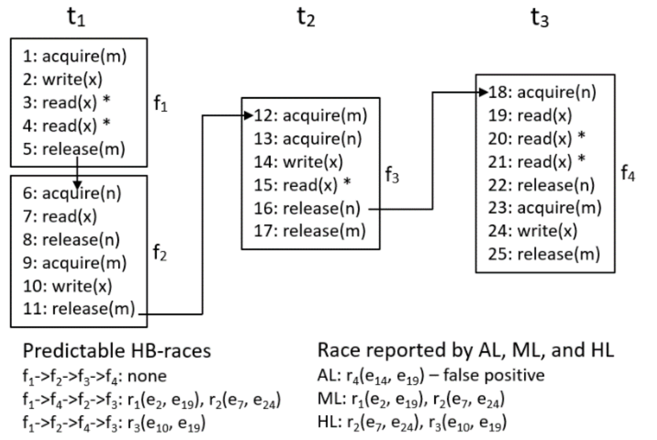

Fig. 1. The exemplified execution trace τ_1. Redundant memory accesses are marked with "*", and arrows indicate HB relations.

FastTrack [9]: There is no HB-race in τ_1 because the HB relations $e_{11} \to_{\text{hb}} e_{12}$ and $e_{16} \to_{\text{hb}} e_{18}$ prevent the accesses to *x* to be executed concurrently. FastTrack as a pure HB-race detector does not detect any race in τ_1.

Consider a trace τ_2 following the execution order of $f_1 \to f_4 \to f_2 \to f_3$. In this case, the HB relations over τ_2 are $e_5 \to_{\text{hb}} e_{23}$, $e_{22} \to_{\text{hb}} e_6$, and $e_{11} \to_{\text{hb}} e_{12}$. There are two HB-races: one between e_2 and e_{19} and another between e_7 and e_{24}, denoted by

$r_1(e_2, e_{19})$ and $r_2(e_7, e_{24})$, respectively, or r_1 and r_2 for short. Similarly, consider a trace τ_3 following the execution order of $f_1 \rightarrow f_2 \rightarrow f_4 \rightarrow f_3$. The HB relations over τ_3 are $e_8 \rightarrow_{\text{hb}} e_{18}$, $e_{11} \rightarrow_{\text{hb}} e_{23}$, $e_{25} \rightarrow_{\text{hb}} e_{12}$, and $e_{22} \rightarrow_{\text{hb}} e_{13}$. There is a HB-race between e_{10} and e_{19}, which is denoted by $r_3(e_{10}, e_{19})$, or r_3 for short.

On each event involving in r_1, r_2, and r_3, along trace τ_1, the locksets being held by t_1 are $\{m\}$, $\{n\}$, and $\{m\}$, respectively, and the locksets being held by t_3 are $\{n\}$, $\{m\}$, and $\{n\}$, respectively. Following the locking discipline, CL_x is initialized as an arbitrary lockset. For each of these racy pairs of accesses, we have $CL_x \cap \{m\} \cap \{n\} = \emptyset$. So, all of them are also $\emptyset$-races in τ_1.

AccuLock [13]: Consider the trace τ_1. AccuLock uses one context placeholder W_x to keep the context of the latest historic write event and three context placeholders $R_x[t_1]$, $R_x[t_2]$, and $R_x[t_3]$, for keeping the context of the latest historic read event of each thread. The context c for each event e is a tuple $\langle E(e), ls(e)\rangle$, and the elements are referred to as $c.epoch$ and $c.lockset$, respectively. On each event e generated by a thread t, AccuLock reports a race with W_x if $W_x.epoch \not\preccurlyeq C(e) \wedge W_x.lockset \cap L(e) = \emptyset$, and reports a race with $R_x[t]$ if $R_x[t].epoch \not\preccurlyeq C(e) \wedge R_x[t].lockset \cap L(e) = \emptyset$. Then, it sets W_x or $R_x[t]$ to the context of event e for write or read, respectively. Suppose W_x, $R_x[t_1]$, $R_x[t_2]$, and $R_x[t_3]$ are $\perp$ (empty) initially. On e_2, AccuLock sets W_x to the context of e_2. On e_3 and e_4, AccuLock skips them by identifying them as redundant accesses. On e_7, AccuLock sets $R_x[t_1]$ to the contexts of e_7. Whenever processing a write event, the algorithm resets all $R_x[t_1]$, $R_x[t_2]$, and $R_x[t_3]$ to $\perp$. Therefore, on e_{10} and e_{14}, W_x is set to the contexts of e_{10} and e_{14} respectively, and $R_x[t_1]$, $R_x[t_2]$, and $R_x[t_3]$ are reset to $\perp$. AccuLock skips e_{15} because it is a redundant access, and e_{20} and e_{21} are skipped in the same way. On e_{19}, only the context of e_{14} is kept in W_x, thus, both the races $r_1(e_2, e_{19})$ and $r_3(e_{10}, e_{19})$ are missed. Then, $R_x[t_3]$ is set to the context of e_{19}. On e_{24}, $R_x[t_1]$ is $\perp$ because of the reset operation on e_{14}, AccuLock cannot locate the context for e_7, failing to detect $r_2(e_7, e_{24})$. On the other hand, lockset $ls(e_{14})$ is a reduced lockset, which is defined as $L(e_{14}) \cap ls(e_{10}) = \{m, n\} \cap \{m\} = \{m\}$. Thus, on e_{19}, we have $ls(e_{14}) \cap L(e_{19}) = \{m\} \cap \{n\} = \emptyset$, and AccuLock reports $r_4(e_{14}, e_{19})$ as a race. Nonetheless, e_{14} and e_{19} are protected by the locksets $\{m, n\}$ and $\{n\}$, respectively. $r_4(e_{14}, e_{19})$ is not a race. In summary, AccuLock may miss to detect some $\emptyset$-races and report false positives with respect to $\emptyset$-races.

MultiLock-HB [30]: The technique uses a combination of strategies to address the problems incurred by AccuLock. Consider the trace τ_1. It uses three context placeholders $W_x[t_1]$, $W_x[t_2]$, and $W_x[t_3]$ to keep three lists of contexts of the historic write events and three context placeholders $R_x[t_1]$, $R_x[t_2]$, and $R_x[t_3]$ to keep three lists of contexts of the historic read events. The context c for each event e is a tuple $\langle E(e), L(e)\rangle$. Keeping all historic events is impractical. Rather than using AccuLock's reset-all strategy on R_x when processing a write event, MultiLock-HB selectively drops some contexts when processing events, which is illustrated as follows. On e_2, $W_x[t_1]$ is $\perp$, and MultiLock-HB adds the context of e_2 into $W_x[t_1]$. On e_3, the technique finds that there is an historic event (e_2 in $W_x[t_1]$ in this example, and in both W_x and R_x in general) sharing the same epoch with e_3 and the lockset of e_3 is not smaller than such an historic event kept. So, the context of e_3 is not kept in $R_x[t_1]$. Similarly, e_4 is not kept in $R_x[t_1]$. On e_7, the context of e_7 is added to $R_x[t_1]$. Unlike AccuLock which increases the thread's epoch on lock release, MultiLock-HB only increases the epoch of a thread on hard order synchronization primitives (e.g., fork or join). Therefore, all the accesses generated in f_1 and f_2 share the same thread epoch. So, on e_{10}, the context of e_{10} is *not* added to $W_x[t_1]$ because it shares the same context with e_2 which is already in $W_x[t_1]$. On e_{14}, e_{19}, and e_{24}, the contexts of e_{14}, e_{19}, and e_{24} are added to $W_x[t_2]$, $R_x[t_3]$, and $W_x[t_3]$, respectively. On the other hand, the technique identifies events e_{15}, e_{20}, and e_{21} as redundant reads, and decides not to keep them. Note that $R_x[t_2]$ is always empty for this trace. On e_{19}, because $W_x[t_2]$ keeps the context of e_{14} and $W_x[t_1]$ keeps the context of e_2 but not that of e_{10}, we only have $L(e_{19}) \cap L(e_{14}) = \{n\} \cap \{m, n\} = \{n\} \neq \emptyset$ and $L(e_{19}) \cap L(e_2) = \{n\} \cap \{m\} = \emptyset$, detecting $r_1(e_2, e_{19})$, missing $r_3(e_{10}, e_{19})$, and avoiding the report of $r_4(e_{14}, e_{19})$. On e_{24}, because $R_x[t_1]$ keeps the context of e_7, we have $L(e_{24}) \cap L(e_7) = \{m\} \cap \{n\} = \emptyset$, detecting $r_2(e_7, e_{24})$. Note that although e_3, e_4, e_{15}, e_{20}, and e_{21} are redundant reads, race detection is still performed when these events occur, and no race could be detected.

Apart from requiring more memory, MultiLock-HB runs significantly slower than AccuLock. This is because on each event e generated by a thread t, MultiLock-HB walks through the two lists kept by $R_x[t]$ and $W_x[t]$ to (1) identify all those events sharing the same epoch with e but each having a larger lockset than e (which requires a set interaction operation on the two corresponding locksets), and remove these identified contexts one by one from $R_x[t]$ if e is a read or from both $R_x[t]$ and $W_x[t]$ if e is a write, and (2) add e to either $R_x[t]$ if e is a read or $W_x[t]$ if e is a write.

Other techniques: **Causally-precedes** (CP) [27] can detect a superset of HB races in a trace. However, CP is unable to predict all these three races from τ_1 because all events in τ_1 are CP-ordered, and all the HB edges in τ_1 are also CP edges. The model of **RVPredict** [12] relies on a constraint solver to exhaustively enumerate all interleaving scenarios to detect HB-races in every such scenario. It can detect all three races with a huge slowdown overhead if RVPredict runs as an online technique.

HistLock: Our technique addresses the above mentioned problems by abstracting function call site (i.e., *access site*) from an event into the context, and uses two context placeholders R_x and W_x to keep lists of historic contexts for read and write events, separately. It leverages two lemmas (see Section IV.D) to safely skip checking on redundant accesses. Moreover, HistLock further introduces an elimination strategy based on function grouping to adaptively remove old contexts of memory accesses in the history. Consider the trace τ_1. On e_2, W_x is $\perp$, and HistLock adds the context of e_2 into W_x. On e_3 and e_4, HistLock skips them because they are redundant accesses, and it also skips e_{15}, e_{20} and e_{21} for the same reason. On e_7, R_x is $\perp$, and HistLock adds the context of e_7 into R_x. On e_{10}, the technique observed that e_{10} is from another *access site* (i.e., f_2) by thread t_1 other than f_1 of e_2 by the same thread t_1 in W_x, so it discards the context of e_2 and keeps the context of e_{10}. On e_{14}, e_{19} and e_{24}, HistLock simply adds their contexts into W_x or R_x for write or read respectively, because there is no new *access site* in t_2 and t_3. HistLock follows the VC updating rules of AccuLock, and checks current read with each context in W_x, or checks current write with each context in W_x and R_x, respectively. It detects $r_2(e_7, e_{24})$ by $L(e_{24}) \cap L(e_7) = \{m\} \cap \{n\} = \emptyset$, and detects $r_3(e_{10}, e_{19})$ by $L(e_{19}) \cap L(e_{10}) = \{n\} \cap \{m\} = \emptyset$. However, HistLock misses $r_1(e_2, e_{19})$ because it has discarded the context of e_2 on e_{19}. Same as MultiLock-HB, HistLock avoids the report of the false warning $r_4(e_{14}, e_{19})$. We note that HistLock is sound (see Theorem 1).

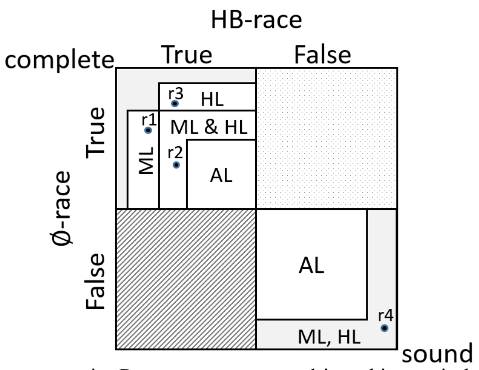

Fig. 2. The race matrix. Race r_1–r_4 are mapped into this matrix by type. The completeness and soundness of AccuLock (AL), MultiLock-HB (ML), and HistLock (HL) are also illustrated with the matrix.

For the coverage problem, owing to the keep of the historic reads and writes, HistLock not only achieves a larger coverage than AccuLock in terms of completeness, but also retains soundness (as MultiLock-HB is). Fig. 2 maps races $r_1 – r_4$ into a race matrix, and shows the detection capability of AccuLock, MultiLock-HB and HistLock with regard to completeness and soundness. Because no race can be mapped into the case of [HB-race: True; Ø-race: False] (the bottom-left area) according to the definition of HB-race and Ø-race, and there is no race in the case of [HB-race: False; Ø-race: True] (the top-right area) under this example, we only consider the cases of [HB-race: True; Ø-race: True] and [HB-race: False; Ø-race: False]. As Fig. 2 shows, AccuLock suffers from the false positive issue due to limited soundness, while MultiLock-HB and HistLock do not. For completeness, MultiLock-HB and HistLock have some common coverage that still larger than which of AccuLock, but they have their respective coverage that cannot be reached by the other.

TABLE 2. THE COUNT OF OPERATIONS

# of operations	AL	ML	HL
detect write-read race	2	10	3
detect read-write race	2	4	4
detect write-write race	3	4	3
update write context	4	4	4
update read context	2	7	2
Total	13	29	16

AL: AccuLock, ML: MultiLock-HB, HL: HistLock.

For the slowdown problem, resulting from redundancy elimination, HistLock dramatically reduces the number of both updating and detection operations (i.e., update algorithm variables and detect data races, respectively) when compared to MultiLock-HB, and just incurs a modest penalty when compared to AccuLock. Table 2 summarizes the count of operations in the three detectors in the running example. In terms of counts, HistLock only requires 55% of all the operations performed by MultiLock-HB, and it just requires 3 more detection operations than AccuLock, but achieves a more precise result than the latter.

IV. OUR TECHNIQUE

In this section, we present HistLock, a novel hybrid detector to detect Ø-races. HistLock is built on two empirical findings: Real-world races frequently occur in (cold) code regions that are not frequently executed) and rarely occur in (hot) code regions that are frequently executed [18]. Thus, the number of accesses in a trace for a data race bug tends to be small. Moreover, races tend to occur between memory accesses within a "short" distance [20].

A. Access History Model

HistLock uses the notion of *access site* and *access context* to adaptively update the memory access history.

Access Site: An *access site* (AS) is a tuple $\langle tid, fn\rangle$ for a memory access event e, where tid is the thread generating e, and fn is the function in which this memory access is performed. We refer to the current access site of thread t as $\boldsymbol{AS_t}$. We also denote the access site at the point of an event e by $\boldsymbol{AS(e)}$. In the running example, for trace τ_1, the access sites $AS(e_2)$, $AS(e_3)$, and $AS(e_4)$ are $\langle t_1, f_1\rangle$; the access sites $AS(e_7)$ and $AS(e_{10})$ are $\langle t_1, f_2\rangle$; the access sites $AS(e_{14})$ and $AS(e_{15})$ are $\langle t_2, f_3\rangle$; and the access sites $AS(e_{19})$, $AS(e_{20})$, $AS(e_{21})$, and $AS(e_{24})$ are $\langle t_3, f_4\rangle$.

Access Context: An *access context* (AC) is a set of tuples, and each such tuple is $\langle as, opNum\rangle$, where as is an access site and $opNum$ is the number of memory accesses generated by the thread $as.tid$ inside the body of the function $as.fn$. In the running example, for trace τ_1, the access contexts for access sites $\langle t_1, f_1\rangle$, $\langle t_1, f_2\rangle$, $\langle t_2, f_3\rangle$, and $\langle t_3, f_4\rangle$ at execution point generating event e_{25} are $\langle\langle t_1, f_1\rangle, 3\rangle$, $\langle\langle t_1, f_2\rangle, 2\rangle$, $\langle\langle t_2, f_3\rangle, 2\rangle$, and $\langle\langle t_3, f_4\rangle, 4\rangle$, respectively.

We define $AC_{|t}$ as $\{as \in AC \mid as.tid = t\}$, which represents the projection of AC on the thread t. The ***depth*** of an access context AC is $\boldsymbol{n}$ (denoted as $\mathbf{AC^n}$) if and only if (1) for every thread t, the projection $AC_{|t}$ contains at most n access sites and (2) at least one projection, say $AC_{|t}$, contains exactly n access sites. In the running example, AC^2 is sufficient to keep all the access contexts for τ_1. With respect to AC^2, $AC^2_{|t1}$ at the execution point generating event e_{25} is $\{\langle\langle t_1, f_1\rangle, 3\rangle, \langle\langle t_1, f_2\rangle, 2\rangle\}$.

We further propose the memory access history model **$\mathbf{AC^n}$-history**: An AC^n-history records all the memory accesses belong to AC^n for the latest n functions invoked by each thread. Each access event e is recorded as a triple $\langle AS(e), E(e), L(e)\rangle$, which is called an ***event context***.

In the running example, AC^2-history is sufficient to keep all the access events in τ_1. AC^1-history only keeps the accesses for the access context $\langle\langle t_1, f_1\rangle, 3\rangle$ until the first event e_7 in f_2 occurs (i.e., until $\langle\langle t_1, f_2\rangle, 1\rangle$ is constructed). In both model instances, access contexts $\langle\langle t_2, f_3\rangle, 2\rangle$, and $\langle\langle t_3, f_4\rangle, 4\rangle$ are still available when e_{25} occurs.

In Fig. 1, we illustrate the result of HistLock in the AC^1-history model, which misses to detect $r_1(e_2, e_{19})$ because e_2 is not kept in AC^1-history. If HistLock is configured with AC^2-history, race $r_1(e_2, e_{19})$ is detected. In the sequel, we present HistLock in the AC^1-history model. Extending HistLock for other AC^n-history is simple.

B. *Algorithm Variables*

For each thread t, there is a vector clock C_t and a lockset L_t. For each shared memory location x, there are two lists R_x and W_x recording the triple $\langle as, epoch, lockset\rangle$ as *event context* for each non-redundant read or non-redundant write performed by all threads. The *as* is $AS(e)$, the *access site* that we have defined in the last subsection; the *epoch* is $E(e)$, which is the clock when the event e for the memory access to x is generated by a thread; and the *lockset* is $L(e)$, indicating the set of locks protecting the shared memory location x during each memory access. Also, each shared memory location x keeps two sequences RC_x and WC_x as *access context* for non-redundant read and write events, respectively.

C. *Updating Algorithm Variables*

As shown in Figure 1, our system monitors a set of events. In this section, we present the handling on monitoring the occurrence of each event in the trace online.

1) *Fork and Join*

On a thread t forking a new thread u, an empty lockset is initialized for thread u, the vector clock C_u is joined with C_t, and finally $C_t[t]$ is increased by 1:

1. $L_u := \emptyset$
2. $C_u := C_u \sqcup C_t$
3. $C_t[t] := C_t[t] + 1$

On a thread u joining a thread t, the vector clock C_t is joined with C_u (i.e., $C_t := C_t \sqcup C_u$).

2) *Acquire and Release*

On acquiring a lock m, the lockset L_t of thread t is updated accordingly: $L_t := L_t \cup \{m\}$.

On releasing a lock m by a thread t, m is removed from the lockset L_t of thread t, and $C_t[t]$ is increased by 1.

1. $L_t := L_t - \{m\}$
2. $C_t[t] := C_t[t] + 1$

3) *Wait and Signal*

For the *wait-signal* synchronization primitives, a thread u is blocked until u receives a signal sent by a thread t. A condition variable cv is used to model each signal sent by t. S_{cv} is the vector clock held by cv to record the latest timestamp of t.

Before t sends the signal on cv, S_{cv} is updated to the latest timestamp among cv and t, and $C_t[t]$ is increased by 1.

1. $S_{cv} := S_{cv} \sqcup C_t$
2. $C_t[t] := C_t[t] + 1$

After u has received the signal on cv, C_u is updated to the latest timestamp among u and cv: $C_u := C_u \sqcup S_{cv}$.

4) *Enter Barrier and Exit Barrier*

When a thread t enters a barrier b before proceeding the next operation, the vector clock of t is updated to the latest timestamp of all the arrived threads at the barrier b right after it passes the barrier b. Also, the timestamp of t is increased by 1 to indicate the synchronization. We use B_b to denote the vector clock held at barrier b to record the latest timestamp from all threads that enter b. Specifically, on a thread t entering a barrier b, HistLock performs the following operation: $B_b := B_b \sqcup C_t$. On a thread t existing a barrier b, HistLock performs the following operations:

1. $C_t := C_t \sqcup B_b$
2. $C_t[t] := C_t[t] + 1$

5) *Read and Write*

Algorithm 1: Read [HistLock]: thread *t* reads variable *x*:

```
1.  $lastRead := R_x.last$
2.  $lastWrite := W_x.last$
3.  if $lastRead.epoch \neq epoch(t) \wedge lastWrite.epoch \neq epoch(t)$ then
4.    for each $\langle as, epoch, lockset\rangle \in W_x$ do
5.      if $epoch \not\preccurlyeq C_t \wedge lockset \cap L_t = \emptyset$ then
6.        report a write-read race warning
7.      end if
8.    end for
9.    if $as_c \neq RC_{x|t}.as$ then
10.     $R_x$.removeAll($RC_{x|t}.as$)
11.     $RC_{x|t} := \langle as_c, 0\rangle$
12.   end if
13.   $R_x$.addToLast($\langle as, epoch(t), lockset_t\rangle$)
14.   $RC_{x|t}.opNum := RC_{x|t}.opNum + 1$
15.   if $RC_{x|t}.opNum >$ MAX_NUM then
16.     $R_x$.removeOldest($RC_{x|t}.as$)
17.   end if
18. end if
```

On thread t reading from a shared memory location x via the event e, HistLock performs the following, which is summarized in Algorithm 1. If both the *epoch* of the last event context in R_x and the *epoch* of the last event context in W_x are different from the epoch of e (at line 3), HistLock checks whether there is a LD violation (lines 4-8) by checking whether $x.epoch \not\preccurlyeq C_t$ where $x.epoch$ is the *epoch* of some event context in W_x. (i.e., A date race warning is reported when the current read is not ordered with any such write and the corresponding write-read pair is not protected by any lock in common.) In addition, if the access site AS_t of e is different from the one recorded in the history $RC_{x|t}.as$ of thread t, the algorithm calls the function $R_x.removeAll()$ with the parameter $RC_{x|t}.as$, which is to remove all the accesses having the access site $RC_{x|t}.as$ from R_x (line 10). Then, the algorithm resets $RC_{x|t}$ (line 11). Finally, the algorithm appends the event context of e to R_x (line 13), and increases $RC_{x|t}.opNum$ by 1 to count the number of accesses associating with the access site AS_t (line 14). In practice, the number of accesses associated to the same access site can be huge. To manage the size of R_x, the algorithm removes the first event context in R_x which belongs to AS_t via the function *removeOldest*() if the number of event contexts (via $RC_{x|t}.opNum$) reaches a threshold (lines 15-16). Note that the algorithm performs race detection and maintains R_x and RC_x only if the current access *e* is non-redundant via the checking at line 3.

Algorithm 2: Write [HistLock]: thread *t* writes variable *x*:

```
1.  $lastWrite := W_x.last$
2.  if $lastWrite.epoch \neq epoch(t)$ then
3.      for each $\langle as,\ epoch,\ lockset\rangle \in W_x$ do
4.          if $epoch \npreccurlyeq C_t \wedge lockset \cap L_t = \emptyset$ then
5.              report a write-write race warning
6.          end if
7.      end for
8.      for each $\langle as,\ epoch,\ lockset\rangle \in R_x$ do
9.          if $epoch \npreccurlyeq C_t \wedge lockset \cap L_t = \emptyset$ then
10.             report a read-write race warning
11.         end if
12.     end for
13.     if $as(t) \neq WC_{x|t}.as$ then
14.         for each $\langle as,\ epoch,\ lockset\rangle \in W_x$ do
15.             if $as = WC_{x|t}.as$ then
16.                 $W_x$.remove($\langle as,\ epoch,\ lockset\rangle$)
17.             end if
18.         end for
19.         $WC_{x|t} := \langle as, 0\rangle$
20.     end if
21.     $W_x$.addLast($\langle as,\ epoch(t),\ lockset_t\rangle$)
22.     $WC_{x|t}.opNum := WC_{x|t}.opNum + 1$
23.     if $WC_{x|t}.opNum >$ MAX_NUM then
24.         $W_x$.removeOldest($as$)
25.     end if
26. end if
```

Algorithm 2 summarizes how HistLock handles write events, which is similar to Algorithm 1. On thread *t* writing to a shared memory location *x* via the event *e*, if the *epoch* of the last event context in W_x is different from the *epoch* of *e* (line 2), the algorithm checks whether there is a LD violation against both the read history and write history (lines 4-12). Lines 13-21 of this algorithm are the same as lines 9-17 of Algorithm 1 except that R_x and $RC_{x|t}$, which are to maintain the current read access, are replaced by W_x and $WC_{x|t}$, which are to maintain the current write access. Similar to Algorithm 1, Algorithm 2 filters out redundant writes from race checking and the maintenance of W_x and WC_x via line 2.

D. Theoretical Guarantee

1) Eliminating Redundant Accesses

HistLock records an AC-history. It keeps non-redundant memory accesses and performs race detections on these accesses only without a miss, which are guaranteed by Lemma 1 and Lemma 2.

Lemma 1 (redundant read) [13]: Suppose that *a*, *b*, and *c* are three memory accesses on the same shared memory location with the above appearance order in a trace. If (1) *b* is a read, (2) *a* and *c* form a race, and (3) the epoch $E(b)$ is as same as the epoch $E(a)$, then *b* is a redundant read with respect to *a* and *c*.

Proof: Given that $E(a) = E(b)$, events *a* and *b* should be generated by the same thread *t*. Moreover, we know that *a* and *c* are concurrent with each other because they are in race and *a* appears before *b* in the trace. Thus, we can infer that *b* and *c* are concurrent with each other. Case (1): Suppose that *c* is a write, then *b* and *c* form a read-write race. In other words, the two conditions that (1) there is either a read-write race or a write-write race between *a* and *c* and (2) *c* is write imply that *b* and *c* form a read-write race. Case (2): Suppose that *c* is a read. If *a* is a read, then there is no race between *a* and *c*, which contradicts to the given condition that *a* and *c* form a race. If *a* is a write, then *a* and *c* form a write-read race. As *b* is a read, there is no race between *b* and *c*. In other words, if we know *a* and *c* forming a write-read race, then there is no race between *b* and *c*. In summary, we can always correctly determine both whether there is a race between *b* and *c* and the type of race between *b* and *c* if there is a race between *a* and *c*. ☐

Lemma 2 (redundant write) [13]: Suppose that *a*, *b*, and *c* be three memory accesses on the same shared memory location with the above appearing order in a trace. If (1) *a* and *b* are both writes, (2) *a* and *c* form a race, and (3) the epoch $E(b)$ is the same as the epoch $E(a)$, then *b* is a redundant write with respect to *a* and *c*.

Proof: Similar to the proof of Lemma 1, given $E(a) = E(b)$, we can infer that *b* and *c* are concurrent with each other. Case (1): Suppose that *c* is a write. Then *b* and *c* form a write-write race. In other words, the two conditions that (1) there is write-write race between *a* and *c* and (2) *c* is write imply that *b* and *c* form a write-write race. Case (2): Suppose that *c* is a read, similar to Case (1), *b* and *c* form a write-read race. In summary, we can always correctly determine the type of race between *b* and *c* if there is a race between *a* and *c*. ☐

As shown in Algorithm 1 (line 3) and Algorithm 2 (line 2), HistLock skips to keep reads and writes that share the same epoch as the latest accesses (say *e*) on the same location kept by it, and thus also skips the race detection operation on these skipped accesses.

2) Soundness

Theorem 1: Every race warning reported by HistLock is a ∅-race.

Proof: When a (non-redundant) memory access event *e* is generated, HistLock records the exact lockset $L(e)$ protecting *e* in both Algorithms 1 and 2, and adds the corresponding event context $\langle AS(e), E(e), L(e)\rangle$ into R_x and W_x by Algorithm 1 (line 13) and Algorithm 2 (line 21), respectively. No reduced lockset $ls(e)$ is kept in R_x and W_x. According to the race detection algorithm of HistLock (Algorithm 1 line 5 and Algorithm 2 line 4 and 9), every reported racy event pair (1) has no *mhb* relation, and (2) experiences a LD violation on *e*. Thus, every reported race by HistLock is ∅-race. ☐

V. EXPERIMENTAL EVALUATION

We present an evaluation of HistLock on race detection effectiveness, runtime slowdown, and memory consumption.

The results are also compared with FastTrack, AccuLock, and MultiLock-HB, which is a precise happens-before race detector, an imprecise hybrid race detector, and a precise hybrid race detector, respectively. Note that HistLock is a precise hybrid race detector, and both MultiLock-HB and HistLock are sound for detecting ∅-races.

A. *Experimental Setup*

We implemented HistLock (HL), FastTrack (FT) [9], AccuLock (AL) [13] and MultiLock-HB (ML) [30] in the same framework on Maple [17]. Our framework was based on the source code of Djit in Maple. For detecting data races, we instrumented synchronization primitives (e.g., fork and join), and memory access (i.e., read and write) by inserting callback analysis function probes. We chose 5 as the threshold of AS-history of HistLock (see MAX_NUM in Algorithms 1 and 2). This is because, according to the prior study [28] , the number of distinct locks used in an *access site* does not exceed 5, and lock releases (resulting in new *epochs* in HistLock) beyond 5 are consecutive acquire and release on the same lock (e.g., in a loop), which a detector needs not to record all of them but can still retains precision to detect the program code location incurring a data race bug. All implemented detectors reported racy pairs at statement level. In other words, if a data race was detected by any of our implemented detector, it reported the location of racy statement in the source code (our benchmark was compiled with debugging information).

For the benchmark subjects, we chose PARSEC benchmark suite 2.1, as well as the real-world programs Apache HTTP server 2.0.48, and MySQL 4.0.12 to evaluate HistLock. The PARSEC benchmark suite was a set of C/C++ multithreaded programs which is also used in previous experiments (e.g., [4][6][14]) on race detection. It contained 13 subjects: *blackscholes*, *bodytrack*, *canneal*, *dedup*, *facesim*, *ferret*, *fluidanimate*, *freqmine*, *raytrace*, *streamcluster*, *swaptions*, *vips*, and *x264*. Because *freqmine* was implemented by OpenMP API [5] and our tool only supported POSIX Threads API, and *facesim* crashed when we ran it under the Maple without our framework, we excluded these two subjects from our evaluation. We used all the remaining 11 subjects in our experiment and executed them with the *simdev* input test [2]. Apache HTTP server (*httpd*) and MySQL (*mysql*) were open source web server and database server, respectively, widely used in practice. The inputs of *httpd* and *mysql* are test cases *Apache #25520* and *MySQL #791* in [17], respectively.

Our experiment was performed on Ubuntu 12.04 Linux (64-bits) with 2.9GHz E5-2690 processor (4 cores assigned) and 16GB physical memory managed by Windows Server 2012 Hyper-V Platform. Each benchmark subject was run for 100 times, and on each run, we applied each of the four detectors to report races. We measured the sum of all distinct reported races (the same racy pair was only counted once), the mean of runtime and memory consumption of each detector on each benchmark subject.

Our tool and benchmark were compiled by GCC 4.6.3. Both time measurement and the races reported may be different if the framework or the benchmark is going to be compiled by other compilers. To assure the correctness of our implementation, we compared the races detected by our detector with those detected by other detectors (e.g., [4]). With reference to Maple [17] and our previous tools, our framework implemented some data structures (e.g., *lockset*, ⟨*epoch*, *lockset*⟩ tuple, memory access history) directly using the C++ STL library.

B. *Data Analysis*

1) *Race Warnings and Detection Effectiveness*

Table 3 shows the number of data race warnings reported by each detector. Column "# of Worker Threads" means the actual number of working threads allocated by the subject, and the harness thread (e.g. main thread) that starts the test case is not counted. Column "Total # of Race Warnings" shows the sum of distinct race warnings in 100 runs. In *blackscholes*, *canneal*, *swaptions*, and *x264*, no race warning was issued by all the four detectors. In *dedup* and *vips*, all the three hybrid detectors reported race warnings, but FT did not. This is because AL, ML, and HL are hybrid detectors targeting on detecting ∅-races, while FT is a happens-before detector

TABLE 3. NUMBER OF RACE WARNINGS REPORTED BY FOUR DETECTORS ON EACH BENCHMARK SUBJECT

	Subject	Application Domain	# of Worker Thread	Total # of Race Warnings (100 runs)				Mean # of Race Warnings (single run)			
				FT	*AL*	*ML*	*HL*	*FT*	*AL*	*ML*	*HL*
PARSEC	blackscholes	Financial Analysis	4	0	0	0	**0**	-	-	-	**-**
	bodytrack	Computer Vision	5	9	18	19	**18**	6	13 (8)	15 (9)	**11 (9)**
	canneal	Engineering	4	0	0	0	**0**	-	-	-	**-**
	dedup	Enterprise Storage	12	0	9	14	**9**	-	-	-	**-**
	ferret	Similarity Search	18	4	4	5	**4**	4	4 (4)	5 (4)	**4 (4)**
	fluidanimate	Animation	4	33	33	50	**57**	20	21 (21)	47 (33)	**45 (33)**
	raytrace	Rendering	4	1	1	1	**1**	-	-	-	**-**
	streamcluster	Data Mining	8	90	105	138	**137**	70	74 (74)	135 (90)	**112 (90)**
	swaptions	Financial Analysis	4	0	0	0	**0**	-	-	-	**-**
	vips	Media Processing	3	0	15	21	**15**	-	-	-	**-**
	x264	Media Processing	5	0	0	0	**0**	-	-	-	**-**
Real-world Application	httpd	Web Server	3	27	28	29	**29**	18	19 (19)	25 (24)	**21 (21)**
	mysql	Database Server	3	173	254	267	**256**	118	171 (134)	227 (148)	**216 (142)**
	Total			337	467	544	**526**	236	302 (260)	454 (308)	**409 (299)**

Figures in parentheses are the numbers of HB-races. FT is a HB detector, others are hybrid detectors.

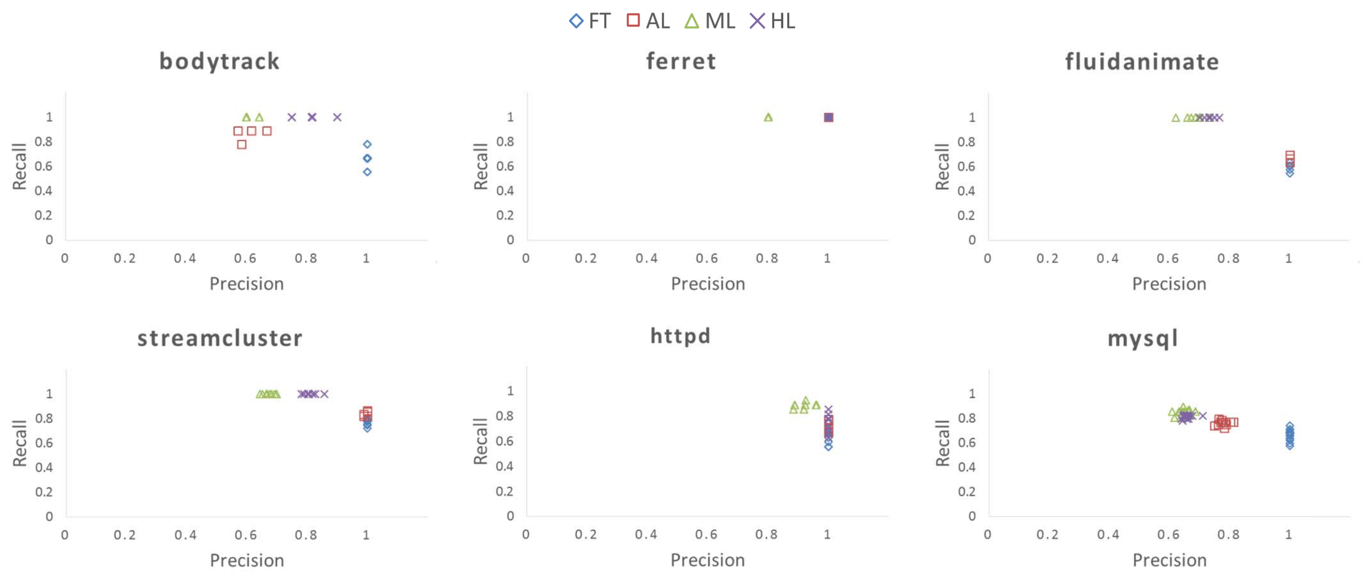

Fig. 3. The recall and precision rate of FT, AL, ML, and HL on selected benchmark subjects.

targeting on detecting HB-races. As we discussed in Section II, ∅-race is the superset of HB-races. Moreover, despite that AL reports false positives of ∅-races (see our illustration in Section III), ML and HL still reported significantly more race warnings than AL (by 77 and 59, respectively). ML reported more race warnings than HL by 38. However, there is no guarantee that the extra races reported by hybrid detectors are all real races, in the worst case, they may all be false positives. Since confirming whether those extra races are false positives is difficult and time consuming (e.g., all possible interleaving need to be explored), we leave this task as our future work.

To further evaluate the effectiveness of HL and the other three detectors, we selected 6 subjects (i.e., *bodytrack*, *ferret*, *fluidanimate*, *streamcluster*, *httpd*, and *mysql*) that have race warnings reported by all the four detectors, and applied FT, AL, ML, and HL to the same execution trace of those subjects within a single run. Column "Mean # of Race Warnings" of Table 3 shows the mean number of race warnings in single run by running 100 times. Additionally, figures in parentheses are the numbers of HB-races detected by the three hybrid detectors. Because finding all real races in a program is still a challenging research topic, for each subject, we use the set of HB-races detected by FT in 100 runs as the approximate whole set of real races, and we refer to it as set Ω. We firstly define the notion of true positive, false positive, and false negative in our evaluation.

- True Positive (TP): HB-races reported within a single run.
- False Positive (FP): Extra races other than HB-races reported within a single run.
- False Negative (FN): The missed HB-races within a single run by referencing set Ω.

We then formulate the recall rate and precision rate as below:

- Recall = TP / (TP + FN)
- Precision = TP / (TP + FP)

Fig. 3 shows the distribution of recall and precision rate on the 6 selected benchmark subjects for the four detectors. In *bodytrack*, the results were distributed by few discrete points. In *ferret*, in every run, all FT, AL and HL achieved 100% recall and precision rate, and ML attained 100% recall rate but incurred precision rate at 80%. Except the above two subjects, the other 4 subjects showed clusters for each detector. Because FT is a pure HB detector and only detect HB-races, the precision rate of FT is always 100% in all subjects. However, it had a lowest recall rate compared with the other three hybrid detectors. For hybrid detectors, AL had a higher (or at least the same) precision rate than both ML and HL in all subjects except *bodytrack*, however, it never achieved a high recall rate. On the other hand, ML and HL achieved very high recall rates (100% in most cases except *httpd* and *mysql*); but they incurred lower precision rates compared to AL in all subjects. Between ML and HL, HL always performed better than ML in terms of precision, and just incurred a slightly lower recall rate than ML in *httpd* and *mysql*.

2) *Execution Time and Runtime Slowdown*

Table 4 shows the evaluation result on comparing the execution time of four implemented detectors. The "Base" time is the execution time that the subject runs on Maple with dummy instrumentation, so the runtime overhead caused by the Maple framework itself can be factored out. The slowdown factor is shown below the total execution time of each detector by taking "Base" time as the normalization base.

From the column "Time" in Table 4, the three hybrid detectors AL, ML, and HL incurred overall runtime slowdowns of 1.6x, 5.9x, and 2.3x, respectively. Among them, both AL and HL were significantly faster than ML by 269% and 156%, respectively. As a HB detector, FT incurred 1.5x slowdown, AL and HL were 6.7% and 53% slower than FT, respectively. In 9 out of 13 benchmark subjects (i.e., *bodytrack*, *dedup*, *ferret*, *fluidanimate*, *streamcluster*, *swaptions*, *vips*, *x264*, and *mysql*), HL ran faster than ML by at most as 373% (in *dedup*).

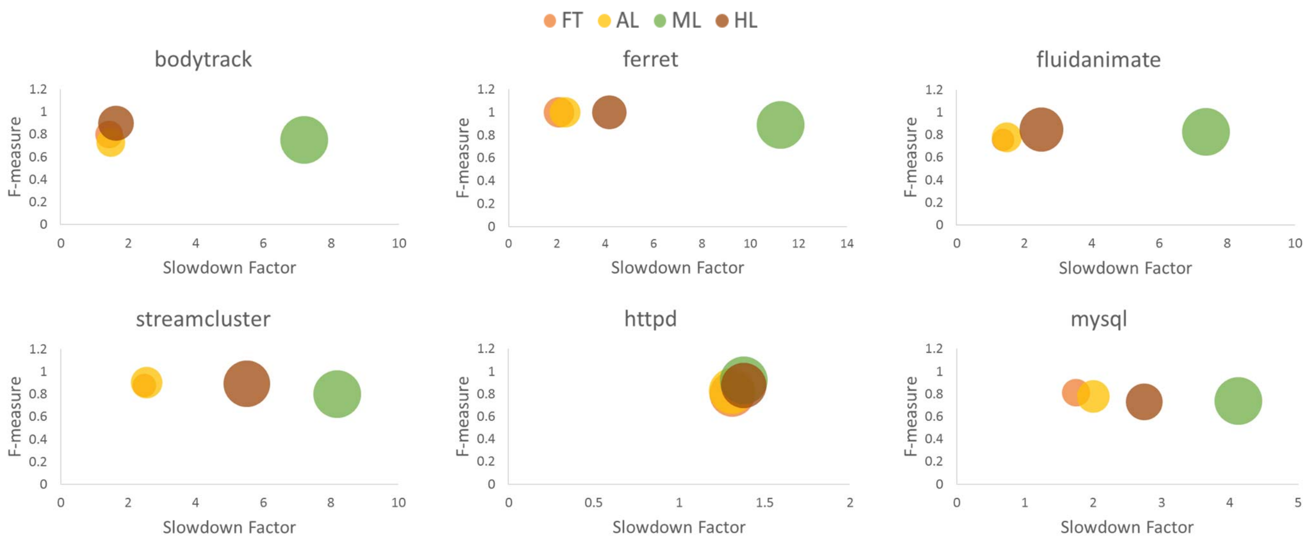

Fig. 4. The comparison on F-measure, slowdown factor and memory consumption on selected benchmark subjects. The size of bubbles indicate memory consumption (smaller slowdown factor is better, higher F-measure is better, smaller bubble is better).

In the rest subjects (i.e., *blackscholes*, *canneal*, *raytrace*, and *httpd*), HL ran as fast as ML. As a hybrid detector, although HL is significantly faster than ML in most cases, it is still slower than AL by 43%. Nonetheless, as we presented in the last subsection, AL had very limited coverage on detecting predictable HB-races, and HL outperformed AL greatly with respect to the recall rate.

TABLE 4. EXECUTION TIME ON EACH BENCHMARK SUBJECT

	Subject	Time (in second)				
		Base	*FT*	*AL*	*ML*	*HL*
PARSEC	blackscholes	0.5	0.6	0.8	0.9	**0.9**
	bodytrack	39	56	58	281	**64**
	canneal	1.1	1.1	1.2	1.4	**1.3**
	dedup	33	68	69	406	**85**
	ferret	12	25	28	135	**50**
	fluidanimate	69	95	103	509	**173**
	raytrace	91	106	108	111	**113**
	streamcluster	31	77	79	254	**171**
	swaptions	0.8	1.1	1.2	3.8	**1.3**
	vips	6.2	7.3	7.8	21.6	**8.8**
	x264	2.2	2.4	2.6	3.3	**2.8**
Real-world Application	httpd	2.9	3.8	3.8	4	**4**
	mysql	8	14	16	33	**22**
	Total	296	458	479	1764	**698**
	Slowdown	1x	1.5x	1.6x	5.9x	**2.3x**

3) *Memory Consumption*

The memory consumption of FT, AL, ML, and HL on each benchmark subject is listed as Table 5. By directly comparing from the table, in total, we observe that HL consumes 33% less memory than ML. By comparing the memory consumption between HL and ML, HL cost less memory than ML in 9 out of 13 benchmark subjects (i.e., *bodytrack*, *canneal*, *ferret*, *fluidanimate*, *swaptions*, *vips*, *x264*, *httpd*, and *mysql*), and cost as same memory as ML in the rest 4 subject (i.e., *blackscholes*, *dedup*, *raytrace*, and *streamcluster*). Although HL consumes more memory than FT and AL (by 100% and 57%, respectively), it has higher coverage than the latter two detectors in detecting HB-races as we present in the first subsection.

TABLE 5. MEMORY CONSUMPTION

	Subject	Memory Consumption (in MB)			
		FT	*AL*	*ML*	*HL*
PARSEC	blackscholes	95	95	95	**96**
	bodytrack	1,447	1,632	4,400	**2,402**
	canneal	125	127	155	**130**
	dedup	2,614	2,723	2,813	**2,897**
	ferret	880	891	2,147	**1,109**
	fluidanimate	1,708	3,068	7,764	**6,596**
	raytrace	185	185	186	**187**
	streamcluster	780	1,431	3,185	**3,062**
	swaptions	149	155	221	**171**
	vips	342	376	1,671	**770**
	x264	252	260	468	**303**
Real-world Application	httpd	177	180	193	**176**
	mysql	888	1,223	2,603	**1,522**
	Total	9,642	12,346	25,901	**19,421**

4) *Overall Evaluation*

To comprehensively evaluate our technique, besides runtime slowdown and memory consumption, we further formulate F-measure as a dimension to evaluate the effectiveness of each detector.

- F-Measure = 2 (Precision × Recall) / (Precision + Recall)

Fig. 4 shows the comparison on F-measure, slowdown factor and memory consumption on the 6 selected benchmark subjects (i.e., *bodytrack*, *ferret*, *fluidanimate*, *streamcluster*, *httpd*, and *mysql*) for FT, AL, ML, and HL. The x-axis represents slowdown factor, the y-axis represents F-measure, and the size of bubble presents the memory consumption. Smaller slowdown factor is better, higher F-measure is better, and smaller bubble is better. From Fig. 4, HL achieved higher (or at least the same level of) F-measure than the other three detectors in most cases (except *httpd* and *mysql*). However, it incurred more runtime slowdown and consumed more memory than FT and AL. On the other hand, to compare the two sound ∅-race detectors HL and ML, (1) HL achieved higher F-measure than ML in 3 subjects (i.e., *bodytrack*, *ferret*, and *streamcluster*), attained same level of F-measure of ML in *fluidanimate* and *mysql*, and incurred lower F-measure than ML in *httpd*; (2) HL significantly outperformed ML with respect to runtime slowdown in 5 out of 6 subjects except *httpd*, plus 4 subjects (i.e., *dedup*, *swaptions*, *vips*, and *x264*) not shown in Fig. 4; (3) HL consumed less memory than ML in 5 out of 6 subjects except *streamcluster*, plus 4 subjects (i.e., *canneal*, *swaptions*, *vips*, and *x264*) not shown in Fig. 4. It seems to us that HL achieved an asymmetric trade-off among effectiveness, runtime slowdown, and memory consumption in terms of sound ∅-race detection, and it superior to ML with respect to the three dimensions we evaluated.

VI. RELATED WORK

Hybrid Data Race Detector. Before the inception of FastTrack, there were several hybrid race detectors attempting to combine lockset and HB for race detection [14][21][22][26][33]. These detectors can be classified into two broad categories. The first category is to detect HB races with the help of lockset algorithms to reduce tracking overheads, and MultiRace [22] is a representative example. It always applies lockset based race detection, and when a LD violation is detected, a HB-based race detection is performed to confirm whether there is a HB race for the pair of accesses forming the LD violation. The second category is to detect LD violations with the help of HB-based detectors to lower the amount of false positives, and HYBRID [21] is an example. It employs a subset of happens-before relation (i.e., the *mhb* relation) to filter out LD violations that are actually properly ordered by hard order synchronization primitives. However, all those detectors still use heavy vector clock comparison to determine happens-before relations.

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 [13] is the first work in epoch-based hybrid detection of LD violations, but is unsound. MultiLock-HB [30] is the first work in sound epoch-based hybrid detection of LD violations. It incurs significantly heavier slowdown overheads than AccuLock. Same as MultiLock-HB, HistLock is a sound LD violation detector. It also significantly outperforms MultiLock-HB in terms of slowdown and memory overhead.

Lowering the overhead is a trend in the research of hybrid detector. SimpleLock [31] is an interesting attempt, which aims at lowering the overhead incurred by epoch-based LD violation detectors at the expense of soundness. Compared to MultiLock-HB, SimpleLock only keeps the number of locks being held by a thread at an epoch rather than the identity of each of those locks. It is efficient, but cannot be generalized to detect LD violations involving different locks protecting the same memory location correctly. SimpleLock+ [32] simplifies SimpleLock by recording both the latest memory access information of each thread and whether the memory location has a lock to protect at each epoch only. SimpleLock+ is more lightweight than SimpleLock, but does not address the soundness problem incurred by SimpleLock.

***Sound Predictive Race Detection*.** HB detectors are sound but interleaving-sensitive in detecting HB races. A class of techniques aims at inferring alternative and valid interleaving scenarios from the current trace to detect additional races (without false positives) from these alternative scenarios. *Causally-precedes* (CP) [27] is a refinement of the HB relation, which has been applied to detect a small superset of HB races detected by HB detectors from the same non-deadlocking trace fragment in polynomial time. On analyzing a deadlocking trace, CP cannot distinguish a race from a deadlock occurrence, but either case indicates a concurrency bug. RVPredict [12] refines the maximal causal model [25] with additional value constraints to ensure branch decisions correctly encoded in detecting additional races from trace fragments. The huge overall time overhead to analyze a long trace prevents these two techniques to be general online detectors.

Redundancy Elimination. Redundancy elimination techniques aim to simplify the tracking of causal orders among events in a trace. LOFT [4] safely skips operations on VC tracking when processing a non-interleaved sequence of lock acquisitions and releases performed by each thread in a trace, and has been applied to a HB race detector. IFRit [6] is a region of a trace within which shared variables cannot be updated by other threads. IFRit [6] detects the overlap of such regions accessing the same shared memory location by different threads. RedCard [11] detects races by only checking the first access to a shared memory location in a sequence of events that involves no lock release, treating all other events in the sequence as redundant.

Active Testing. A LD violation may not be a HB race. RaceFuzzer [24] controls the schedule of a concrete execution with the attempt to manifest the accesses involved in a LD violation in a prior trace into a concurrent pair of accesses. Racageddon [8] iteratively applies concolic execution to generate a new schedule for yielding such a concurrent pair of accesses. After finding possible schedules that produce real races, Racageddon further swaps the order of the accesses involving in these races to find additional races. DrFinder [34] infers function calls that may lead to the generation of access events of each memory location in execution points not far later than the corresponding function invocation points, and then generates new schedules based on such look-ahead information.

VII. CONCLUSION

This paper has presented a novel dynamic hybrid data race detector HistLock for the detection of ∅-races. HistLock is sound in its detection precision on ∅-races, novel in its design to keep and check accesses against current memory accesses, and effective in adaptively phrasing out old non-redundant memory accesses through a strategy based on functional contexts. In the experiment, it outperformed FastTrack, AccuLock, and MultiLock-HB in effectiveness, showed an asymmetric trade-off among effectiveness, runtime slowdown, and memory consumption in terms of sound ∅-race detection, and superior to ML with respect to the three dimensions we evaluated. HistLock was validated to be of higher precision, 156% faster and 33% more memory-efficient than MultiLock-HB. It also detected 59 more race warnings than AccuLock, attained higher effectiveness, but ran slower by 43%. In most cases, in a single run, HistLock reported those races detected by FastTrack in 100 runs.

REFERENCES

[1] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: static race detection for java," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 28(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 *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques (PACT'08)*, pp. 72–81, 2008.

[3] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: proportional detection of data races," In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'10)*, pp. 255−268, 2010.

[4] Y. Cai and W. K. Chan, "Lock trace reduction for multithreaded programs," IEEE Transactions on Parallel and Distributed Systems (TPDS), vol.24(12), pp. 2407−2417, 2013.

[5] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," Computational Science & Engineering, vol. 5(1), pp. 46–55, 1998.

[6] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H. J. Boehm, "IFRit: Interference-free regions for dynamic data-race detection," In *Proceedings of the 2012 ACM international conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*, pp. 467–484, 2012.

[7] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," In *Proceedings of the 19th ACM symposium on Operating Systems Principles (SOSP'09)*, pp. 237–25, 2003.

[8] M. Eslamimehr and J. Palsberg, "Race directed scheduling of concurrent programs," In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPOPP'14)*, pp. 301–314, 2014.

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

[10] C. Flanagan, and S. N. Freund, "Type-based race detection for Java," *In Proceedings of the 2000 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'00)*, pp. 219–232, 2000.

[11] C. Flanagan, and S. N. Freund, "Redcard: redundant check elimination for dynamic race detectors," ECOOP 2013–Object-Oriented Programming, LNCS 7920, pp. 255−280, 2013.

[12] J. Huang, P. O. N. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," In *Proceedings of the 2014 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'14)*, pp. 337−348, 2014.

[13] X. Xie, J. Xue, "AccuLock: accurate and efficient detection of data races," In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*, pp. 201−212, 2011.

[14] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: an efficient dynamic race detector," In *the 2009 IEEE International Parallel & Distributed Processing Symposium (IPDPS'09)*, pp. 1–13, 2009.

[15] Z. Lai, S. Cheung, and W.K. Chan, "Detecting atomic-set serializability violations in multithreaded programs through active randomized testing," In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pp. 235–244, 2010.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, vol. 21(7), pp. 558–565, 1978.

[17] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: a coverage-driven testing tool for multithreaded programs," In *Proceedings of the 2012 ACM international conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)*, pp. 485–502, 2012.

[18] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: effective sampling for lightweight data-race detection," In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'09)*, pp. 134−143, 2009.

[19] F. Mattern, "Virtual time and global states of distributed systems," Parallel and Distributed Algorithms, vol. 1(23), pp. 215–226, 1989.

[20] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pp. 329−339, 2008.

[21] R. O'Callahan and J. D. Choi, "Hybrid dynamic data race detection," In *Proceedings of the 9th ACM symposium on Principles and Practice of Parallel Programming (PPOPP'03)*, pp. 167−178, 2003.

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

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," ACM Transactions on Computer Systems (TOCS), vol. 15(4), pp. 391−411, 1997.

[24] K. Sen, "Race directed random testing of concurrent programs," In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'08)*, pp. 11−21, 2008.

[25] T. F. Serbanuta, F. Chen, and G. Rosu, "Maximal causal models for sequentially consistent systems," Runtime Verification, LNCS 7687, pp. 136−150, 2013.

[26] K. Serebryany, T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice," *In Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA'09)*, pp. 62–71, 2009.

[27] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'12)*, pp. 387−400, 2012.

[28] R. Xin,, Z. Qi, S. Huang, C. Xiang, Y. Zheng, Y. Wang, and H. Guan, "An automation-assisted empirical study on lock usage for concurrent programs," In *the 29th IEEE International Conference on Software Maintenance (ICSM'13)*, pp. 100−109, 2013.

[29] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," In *Proceedings of the 2007 Joint Meeting on Foundations of Software Engineering (ESEC/FSE'07)*, pp. 205–214, 2007.

[30] X. Xie, J. Xue and J. Zhang, "AccuLock: accurate and efficient detection of data races," Software: Practice and Experience, vol. 43(5), pp. 543−576, 2013.

[31] M. Yu, S. K. Yoo, and D. H. Bae, "Simplelock: fast and accurate hybrid data race detector," In *Proceedings of the 2013 international conference on Parallel and Distributed Computing (PDCAT'13)*, pp. 50−56, 2013.

[32] M. Yu and D. H. Bae, "SimpleLock+: fast and accurate hybrid data race detection," The Computer Journal, doi: 10.1093/comjnl/bxu119, 2014.

[33] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," In *Proceedings of the 20th ACM symposium on Operating Systems Principles (SOSP'05)*, pp. 221−234, 2005.

[34] Y. Cai, and L. Cao, "Effective and precise dynamic detection of hidden races for Java programs," In *Proceedings of the 2015 Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, pp. 450−461, 2015.