ACCULOCK: Accurate and Efficient Detection of Data Races

Xinwei Xie and Jingling Xue
Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales
NSW, Australia 2052
{xinweix, jingling}@cse.unsw.edu.au

***Abstract*—Happens-before detectors are precise but can be too conservative to detect certain data races in repeated test runs as they are sensitive to thread interleaving. By making the opposite tradeoffs, lockset detectors can detect more races but are not precise (by reporting false positives). For both types of detectors, happens-before detectors run more slowly as they use expensive vector clocks. Existing hybrid race detectors (combining lockset and happens-before) alleviate some of the limitations in both analysis techniques at the cost of additional analysis overhead.**

Recently, due to FASTTRACK, epoch-based happens-before and lockset detectors now exhibit comparable performance. It is the time to rethink how to design a hybrid race detector to balance precision and coverage, by leveraging the lightweightness of epoch clocks. ACCULOCK is the first such a solution.

ACCULOCK analyzes a program by reasoning about the subset of the happens-before relation observed with lock acquires and releases excluded, thereby reducing its sensitivity to thread interleaving. When such a weaker happens-before relation is violated, ACCULOCK applies a new efficient lockset algorithm to enforce a lock-based synchronization discipline by distinguishing the locks protecting reads and writes. The key motivation behind is to ensure that ACCULOCK can improve happens-before detectors by discovering also data races in alternate thread interleavings when analyzing one program execution while limiting false warnings thus incurred in a controlled manner. In addition, ACCULOCK achieves these objectives by maintaining comparable performance as FASTTRACK, the fastest happens-before detector.

All these properties of ACCULOCK are validated and confirmed by comparing it against six other detectors, all implemented in Jikes RVM using 11 benchmark programs.

I. INTRODUCTION

The ubiquity of multicore processors is clearly increasing software complexity by driving the need for multithreaded applications. A *data race* occurs in a multithreaded program when at least two different threads access the same memory location without an ordering constraint enforced between the accesses, such that at least one of the accesses is a write [1]. Data races themselves are not necessarily errors; but they often introduce serious hard-to-find, crash-causing concurrency-related software defects. Therefore, tools for automatic detection of data races are invaluable.

Ultimately, data races should be detected with a range of tools used in stages, including both static and dynamic detectors. Static analysis techniques are (statically) sound [2], [3], [4], [5], [6], [7] but imprecise (by producing many false positives). In contrast, dynamic analysis techniques [8], [9], [10], [11], [12], [13], [14], [15], [16] produce false negatives but can be precise or imprecise. This work presents a new dynamic race detector that combines a new lockset analysis with happens-before analysis in a novel way by leveraging the recent advantages made in the FASTTRACK work [12].

A. Related Work

There are a number of dynamic detectors reported, with *lockset* and *happens-before* sitting at the two ends of the spectrum. Lockset race detection, as exemplified by ERASER [9], analyzes a program by enforcing a locking discipline and reports a race if two threads access a shared memory location without holding a common lock. Lockset detectors are insensitive to thread interleaving and run with low performance overhead, but are imprecise (by reporting false positives) because they ignore the ordering of events in program executions.

Happens-before race detection tracks the happens-before relation, a casual relationship induced by program order and synchronization order during an execution, represented using vector clocks (VCs) [17]. Happens-before detectors are sensitive to thread interleaving and dynamically sound and precise for one particular execution only. Earlier examples include TRADE [14] and DJIT$^+$ [18]. VCs are expensive to implement, both in time and space. Recently, FASTTRACK [12] has reduced most VC-based operations from $O(n)$ to $O(1)$, where n is the number of threads, i.e., size of a vector clock, by using scalar clocks called *epochs* whenever possible. In their implementation [12], FASTTRACK achieves about the same performance as the lockset detector ERASER, at the expense of being dependent on thread interleaving. PACER [19] lowers its overhead by applying statistical sampling.

Goldilocks [20] captures the happens-before relation using a unified lockset containing locks, threads and volatile variables. Although it is dynamically sound and precise, the overhead of traversing its global synchronization list is much higher than FASTTRACK in a high-performance JVM, as shown in [12].

In the pre-FASTTRACK era, there were two kinds of attempts on combining lockset and happens-before race detection to detect data races. One is to use the lockset information to improve the efficiency of VCs, as in MULTIRACE [18], by limiting VC operations to accesses to a shared location with an empty lockset. The other is to use the happens-before

Thread T1	Thread T2	Thread T1	Thread T2	Thread T3	Thread T1	Thread T2	Thread T1	Thread T2

= x Ⓐ
lock l_1
= x
unlock l_1
	lock l_1
	x = Ⓑ
	unlock l_1
		lock l_2
		= x Ⓐ
		unlock l_2
			lock l_1
			lock l_2
			= x
			unlock l_2
			unlock l_1
				lock l_1
				x = Ⓑ
				unlock l_1
					lock l_2
					= x Ⓐ
					unlock l_2
					lock l_1
					= x
					unlock l_1
						lock l_1
						x = Ⓑ
						unlock l_1
							// create the list of
							// new nodes in list1 Ⓐ
							lock l_1
							channel = list1
							unlock l_1
								lock l_1
								list2 = channel
								unlock l_1
								// process the nodes
								// data in list2 Ⓑ

(a) Race (A, B) **always** reported by ACCULOCK but **possibly** by FASTTRACK.		(b) Race (A, B) **always** reported by ACCULOCK but **possibly** by FASTTRACK.			(c) Race (A, B) reported by ACCULOCK **if and only if** reported by FASTTRACK.		(d) A false positive, (A, B), **always** reported by ACCULOCK but **never** by FASTTRACK, caused by shared channels [21].	

Fig. 1. An illustration of the design philosophy behind ACCULOCK compared to FASTTRACK given the **same thread interleaving**.

information to reduce false positives in a lockset detector like ERASER. "HYBRID" [21] does this by reporting the same true positives as ERASER while RACETRACK [10] may report less to trade precision for efficiency. However, these earlier hybrid detectors are often (significantly) slower than lockset and happens-before detectors as VCs are expensive to maintain. The only exception is that RACETRACK has about the same performance as ERASER but is less precise. These results are validated by their authors, partly in the FASTTRACK work [12] and more extensively in our experiments.

Independently, THREADSANITIZER [22] has also recently been designed to combine lockset and happens-before to dynamically detect data races. However, it differs from ACCULOCK in two key aspects. First, THREADSANITIZER still uses VCs to reason about happens-before while ACCULOCK adopts lightweight epochs. Second, THREADSANITIZER keeps track of multiple locksets for concurrent writes to a shared location to increase its chances in detecting races caused by the multiple protecting lock idiom while ACCULOCK maintains only the lockset for the last write. However, according to the authors of THREADSANITIZER [22], such races rarely occur in real-world programs. Due to the above two differences, THREADSANITIZER suffers no less analysis overhead than earlier hybrid detectors such as HYBRID and MULTIRACE. (Using caching in VC-based detectors can speed up only some VC operations as caching is not overhead-free and all VC operations on cold and conflict cache misses are still $O(n)$.)

B. Overview of the Idea

1) Motivation: Due to FASTTRACK, an epoch-based happens-before detector has nearly closed the performance gap with a lockset detector. It is the time to reconsider how to design a dynamic detector that combines happens-before and lockset to obtain improved precision and coverage, under the conditions that the detector achieves comparable performance and limits the number of false positives reported compared to FASTTRACK. ACCULOCK is the first such a solution.

The four key design objectives for ACCULOCK, as illustrated in Figure 1 and explained below, are as follows:

1) To increase coverage of data races in a happens-before detector by detecting also races in alternate thread interleavings when analyzing a particular program execution;
2) To reduce the sensitivity of a happens-before detector to thread interleaving caused also due to the use of varying numbers of threads in the same program;
3) To limit false positives incurred in a controlled manner;
4) To achieve comparable performance as FASTTRACK with more or less the same memory overhead.

The motivations for these objectives are discussed below. To motivate Objective (2) for now, we have tested FASTTRACK (both our implementation and the version of PACER [19] with its sampling rate set as 100%) on `xalan` from `DaCapo`. FASTTRACK reports a particular race, as discussed in Section V-B1, depending on thread interleavings caused by using varying numbers of threads. For example, FASTTRACK never reports the race in 500 runs tested when the number of threads is 8, but ACCULOCK catches it in all 500 runs.

2) Solution: ACCULOCK leverages the framework of FASTTRACK but with this new set of design objectives to meet.

FASTTRACK is dynamically sound and precise since it uses the true happens-before relation, denoted $\xrightarrow{hb}$, induced by program order and synchronization order. In Figure 1, FASTTRACK will report a race between `A` and `B` in (a) and (c) if `T2` acquires lock l_1 before `T1` does since `B` $\xrightarrow{hb}$ `A` does not hold, but chooses to be silent (in order to be 100% dynamically precise) if the lock acquisition order is reversed, in which case, `A` $\xrightarrow{hb}$ `B` holds. In (b), FASTTRACK does not report the racy pair (`A`, `B`) when the thread interleaving is either `T1` → `T2` → `T3` or `T3` → `T2` → `T1`. In (d), FASTTRACK will never report a race. This last example provides an abstraction of `shared channels` [21], in which accesses to `channel` are synchronized but accesses to the transmitted data (i.e., the nodes in the two lists) need not be.

ACCULOCK achieves the four design objectives by (1) using $\xrightarrow{accu-hb}$, a *thread-interleaving-less-sensitive* subset of $\xrightarrow{hb}$, obtained with all lock acquires and releases excluded and (2) applying a new lockset algorithm that distinguishes the locks protecting reads and writes when enforcing a locking discipline. By comparing with FASTTRACK in Figure 1, ACCULOCK always reports the races in (a) and (b) since the two unordered accesses `A` and `B` in each case are not protected by a common lock (to satisfy Objectives (1) and (2)). In (c), which provides an abstraction of multiple protecting locks, ACCULOCK behaves exactly the same as FASTTRACK (to achieve Objective (4)). Otherwise, any lockset algorithm may have to use sets of sets of locks instead of just sets of locks [9, p. 409] (to satisfy Objective (3)), but this can be costly.

In addition, ACCULOCK also tries to fulfill Objective (4) by leveraging the lightweight epoch representation of $\xrightarrow{accu-hb}$ to provide constant-time fast paths for most reads and writes in program order, as in FASTTRACK and by avoiding $O(n)$ vector clock operations on lock acquires and releases (due to the use of $\xrightarrow{accu-hb}$ rather than $\xrightarrow{hb}$). In (d), ACCULOCK reports a potential race between A and B to the data transmitted via the channel, which turns out to be a false positive (discovered only by further analysis), but FASTTRACK does not (as it only reports a race actually seen).

Note that neither FASTTRACK nor ACCULOCK understands the semantic differences among all the four cases (not to mention shared channels, in particular).

Definition 1 ($\emptyset$**-Races).** A potential data race detected between two concurrent accesses to a location x in a program execution is called a *$\emptyset$-race* if they do not access x with a common lock (i.e., with the set of common locks being $\emptyset$) in the execution.

We argue that $\emptyset$-races such as the one illustrated in Figure 1(d) should be flagged for further analysis due to the detrimental effects of data races on the reliability of multi-threaded software. Alternatively, such false positives can be eliminated with user annotations so that the missing happens-before relationship is thus established [21].

By using the new lockset algorithm proposed, ACCULOCK is expected to report usually $\emptyset$-races in real code. In fact, in the absence of multiple protecting locks, as is common in practice, all races reported by ACCULOCK are $\emptyset$-races (Theorem 3). In our experiments, all races reported by ACCULOCK for 10 out of the 11 benchmark programs used are $\emptyset$-races.

C. Contributions

- We introduce a new dynamic race detector, ACCULOCK, with all properties discussed in Section I-B (Section III). We provide a new lockset algorithm that enables a seamless integration of the lockset and happens-before mechanisms to achieve a fine balance between precision and coverage of data races reported.
- We have implemented ACCULOCK and six other dynamic detectors, ERASER [9], DJIT$^+$ [18], RACETRACK [10], MULTIRACE [18], "HYBRID" [21] and FASTTRACK [12] in Jikes RVM and validated ACCULOCK's fulfilment of its design objectives using 11 benchmarks, the largest Java programs ever used as a collection in the dynamic analysis literature (Section IV).
- We have analyzed all these detectors (in terms of performance, memory requirement, precision and coverage) to provide insights for further studies (Section V). In particular, ACCULOCK is capable of finding more data races than FASTTRACK when looking for $\emptyset$-races while maintaining comparable analysis overhead.

II. BACKGROUND

We first review vector clocks (VCs) and how a generic $O(n)$ (time and space) VC-based happens-before detector works, where n is the number of threads (Section II-A). We then describe how FASTTRACK uses epoch clocks to reduce most $O(n)$ VC operations to $O(1)$ (Section II-B). Finally, we review the basic LOCKSET algorithm and touch upon ERASER, the classic lockset algorithm, on which many others are based (Section II-C).

A. VCs and VC-based Happens-Before Detection

VC detectors soundly and precisely track the (true) happens-before relation $\xrightarrow{hb}$, which is the transitive closure of its (intra-thread) program order and (inter-thread) synchronization order (induced by, e.g., forks, joins, lock acquires and releases). By performing dynamic analysis on all synchronization, read and write operations, they detect concurrent variable accesses and report a data race if one is a write.

A vector clock $VC : Tid \rightarrow Nat$ records a clock for each thread in the program. VCs are partially ordered ($\sqsubseteq$) point-wise with a minimum element $(0, \ldots, 0)$ and a join operation ($\sqcup$), which is defined to be a point-wise maximum.

1) Synchronization Operations: Accesses to synchronization objects (*threads*, *locks* and *volatile variables* in Java) are always ordered and never raced. Each synchronization object has its own clock. Each thread t keeps a vector clock C_t such that for any thread u, the entry $C_t[u]$ records the clock for the last operation of u that happens before the current operation of thread t. Similarly, the analysis maintains a vector clock C_l (C_v) for each lock l (each volatile variable v).

These VCs are updated on synchronization operations that affect $\xrightarrow{hb}$. For example, when a thread t releases lock l, the analysis updates C_l with C_t (by copying the contents of C_t into C_l) and then increments the entry t in t's vector clock. When a thread t subsequently acquires lock l, the analysis updates C_t to be $C_t \sqcup C_l$, since all subsequent operations of t happen after that release operation. Obviously, a join or copy takes $O(n)$ in time and space.

2) Variable Reads and Writes: For each shared variable, i.e., memory location x, which can be an object field or an object itself depending the level of granularity used, the analysis keeps two vector clocks, R_x and W_x, such that the entries $R_x[t]$ and $W_x[t]$ record the clock values of the last read and write to x by thread t, respectively. At each read, the analysis checks that prior writes happen before the current thread t's VC, C_t, by verifying $W_x \sqsubseteq C_t$ and then updates $R_x[t]$ with $C_t[t]$. At each write, the analysis checks for data races with prior reads and writes by verifying $W_x \sqsubseteq C_t$ and $R_x \sqsubseteq C_t$ and then updates $W_x[t]$ with $C_t[t]$. Again, all these happens-before checks take $O(n)$ time each.

B. Epochs and FASTTRACK

FASTTRACK has reduced most $O(n)$ VC operations to $O(1)$, by exploiting the following insights: (1) In a race-free program, all writes to a variable are totally ordered by $\xrightarrow{hb}$, and on encountering a write, all previous reads must happen before the write by $\xrightarrow{hb}$, and (2) the analysis must keep track of all concurrent reads since they potentially race with a subsequent write. As a result, FASTTRACK replaces the write vector clock W_x with an *epoch* $c@t$, which records the clock value c at

Algorithm 1 Read [FASTTRACK]: thread t reads variable x

```
if $R_x \neq$ epoch($t$) then                    {If same epoch, no action}
    check $W_x \preccurlyeq C_t$
    if $|R_x| = 1 \wedge R_x \preccurlyeq C_t$ then
        $R_x \leftarrow$ epoch($t$)                {Overwrite read map}
    else
        $R_x[t] \leftarrow C_t[t]$                 {Update read map}
    end if
end if
```

Algorithm 2 Write [FASTTRACK]: thread t writes variable x

```
if $W_x \neq$ epoch($t$) then                    {If same epoch, no action}
    check $W_x \preccurlyeq C_t$
    if $|R_x| \leqslant 1$ then
        check $R_x \preccurlyeq C_t$
    else
        check $R_x \sqsubseteq C_t$                {$O(1)$ amortized time}
    end if
    $R_x \leftarrow empty$
    $W_x \leftarrow$ epoch($t$)                    {Update write epoch}
end if
```

which thread t performed the last write to x. When reads are ordered by $\xrightarrow{hb}$, FASTTRACK uses an epoch for the last read. Otherwise, it uses VCs for reads.

Some notations are introduced and used later in presenting our ACCULOCK algorithm. The function **epoch**(t) is a shorthand for $c@t$, where $c = C_t[t]$. In addition, $c@t \preccurlyeq VC$ iff $c \leqslant VC[t]$, where VC is a vector clock.

For comparison purposes later with ACCULOCK, Algorithms 1 and 2 show the core part of FASTTRACK in handling reads and writes but is formulated more compactly according to [19]. In [19], read epochs and VCs are unified into a *read map*, which maps zero or more threads to clock values. Thus, a read map is an epoch if it has one entry, the initial state is epoch $0@t$ if it has zero entries, and a VC otherwise.

Following [12], gray shading indicates operations that take $O(n)$ time each, where n is the number of threads.

At a read, FASTTRACK does nothing if the read map R_x is an epoch equal to the current thread's time. Otherwise, it checks whether the last write races with the current read. Finally, it either replaces R_x with an epoch if R_x is an epoch and happens before the current read or updates R_x's t entry.

At a write, FASTTRACK also does nothing if the variable's write epoch is the same as the thread's epoch. Otherwise, it checks to see if the current write races with the last write. Finally, it checks for races with prior reads and clears the read map. In FASTTRACK, the read map is cleared this way because for each prior read in the read map, one of the following statements holds: (1) it races with the current write, in which case, the race has been detected and reported, or (2) it happens before the current write, in which case, both accesses do not race. The shaded check takes $O(|R_x|) \leqslant O(n)$ time but it is amortized over the last $|R_x|$ analysis steps that take $O(1)$ time each. By being able to clear the read map, FASTTRACK can adaptively switch between epochs and VCs so that the number of $O(n)$ VC operations is greatly reduced.

Algorithm 3 Access [LOCKSET]: threads t reads or writes x

```
$L_t \leftarrow$ set of locks held now
if $x$ is a read then
    $L_t \leftarrow L_t \cup \{readers\_lock\}$
end if
$CL_x \leftarrow CL_x \cap L_t$                  {Initialized with set of all locks}
check $CL_x = \emptyset$                         {Check for races}
```

C. *Locksets and* ERASER

The basic LOCKSET algorithm, as depicted in Figure 3, detects violations of a locking discipline without considering the happens-before information. LOCKSET requires that every shared location be protected consistently by at least one common lock on each access (read or write) to it.

For each thread t, L_t holds the set of all locks acquired by t at any time. For each shared location x, the *candidate set*, CL_x, records the set of all locks, known as *lockset*, that have consistently protected every access to x so far. The use of a "fake lock" in [18], denoted *readers_lock*, serves to suppress false warnings on concurrent reads to x without holding a common lock. However, any later write to x will cause *readers_lock* to be removed from CL_x.

By ignoring $\xrightarrow{hb}$, LOCKSET may result in excessive false positives. To alleviate this, ERASER uses a state machine to handle unsoundly thread-local and read-shared data. For the reasons regarding the unsoundness, we refer the reader to [9].

III. ACCULOCK

We describe our ACCULOCK algorithm that detects data races dynamically by taking advantage of the lightweight epoch representation of $\xrightarrow{accu-hb}$ (a thread-interleaving-less-sensitive subset of $\xrightarrow{hb}$) and by also deploying an efficient yet more precise new lockset algorithm (than LOCKSET). This combination enables ACCULOCK to achieve a better coverage of data races than FASTTRACK and a better precision than ERASER. ACCULOCK achieves these objectives as well as the others about maintaining comparable performance as FASTTRACK and limiting its data races reported to be mostly $\emptyset$-races (Definition 1), as discussed in Section I-B,

Algorithms 4 – 13 give the algorithmic core of ACCULOCK, with Algorithms 10 and 11 being ACCULOCK's counterparts of FASTTRACK's Algorithms 1 and 2.

The notations, **epoch**(t) (the current epoch of thread t), $\preccurlyeq$ (on an epoch and a VC) and $\sqcup$ (on two VCs), as in FASTTRACK, and L_t as in LOCKSET, are used as before.

Below we introduce the components of ACCULOCK by functionality. We explain the design decisions and tradeoffs made in order for ACCULOCK to meet its design objectives. Section III-A discusses how to track $\xrightarrow{accu-hb}$. Section III-B describes how ACCULOCK approximates the lock-subset condition [18], [21] to both eliminate some redundant race checks and catch more data races than FASTTRACK. Section III-C contains the key contribution of the work. It describes how ACCULOCK detects data races by combining $\xrightarrow{accu-hb}$ and our new lockset algorithm. Section III-D characterizes the data races reported by ACCULOCK with respect to FASTTRACK.

A. Tracking $\xrightarrow{accu-hb}$

Like all happens-before detectors, ACCULOCK keeps a vector clock C_t for every thread t. However, ACCULOCK uses these vector clocks to track $\xrightarrow{accu-hb}$ rather than $\xrightarrow{hb}$.

In Algorithms 6 – 9, ACCULOCK tracks the inter-thread synchronization order induced by `fork`, `join`, `notify` and `notifyAll` as in Java as well as the intra-thread program order by incrementing clock C_t's t entry or C_u's u entry. Note that lock acquires and releases in Algorithms 4 and 5 do not affect the synchronization order. However, in a lock release, the clock of thread t is incremented merely to facilitate the lock-subset optimization described in Section III-B.

The shaded $O(n)$ VC operations for tracking the synchronization order induced by `fork`, `join`, `notify` and `notifyAll` are needed in all happens-before detectors. However, these events happen infrequently compared to lock acquires and releases, which are tracked in $O(n)$ time and space in FASTTRACK but ignored in ACCULOCK.

As for volatile reads and writes, there are two choices, depending a client's need for a particular program. If Algorithms 12 and 13 are incorporated, then ACCULOCK behaves exactly as FASTTRACK by including the effects of volatile variables on $\xrightarrow{accu-hb}$. Otherwise, ACCULOCK proceeds by treating volatile variables just as lock objects.

B. Approximating the Lock-Subset Condition

Traditionally, the lock-subset optimization [18], [21] serves to reduce the number of accesses participating for race checks.

Definition 2 (Redundant Accesses)**.** Let there be two accesses a and b to the same shared location x made by the same thread. We say that b is *redundant* with respect to a if (1) a and b are both reads, a and b are both writes, or a is a write and b is a read, and (2) a's lockset is a subset of b's lockset.

Thus, a future access to x that races with b must also race with a. As a result, b can be ignored in race detection.

Eliminating all redundant checks would be counter-productive as subset operations are costly. ACCULOCK approximates the lock-subset condition so that an elimination of some redundant accesses serves to both bring a performance gain and increase the number of data races detected.

Note that in Algorithms 4 and 5, the vector clock C_t in thread t ticks only at a lock release but not at a lock acquire. Thus, the results stated in Lemmas 1 and 2 are immediate.

Lemma 1 (Redundant Reads)**.** *Let P_x be the epoch of a prior access (read or write) to x. If P_x = **epoch**(t), then a future read at **epoch**(t) is redundant (w.r.t. the prior access).*

Lemma 2 (Redundant Writes)**.** *Let P_x be the epoch of a prior write to x. If P_x = **epoch**(t), then a future write at **epoch**(t) is redundant (w.r.t. the prior write)*

Consider thread `T1` in Figure 1(a). If P_x represents epoch of the first read to `x` in the code, then the second read to `x` is

Algorithm 4 Acquire: thread t acquires lock m
```
$L_t \leftarrow L_t \cup \{m\}$
```

Algorithm 5 Release: thread t releases lock m
```
$L_t \leftarrow L_t - \{m\}$
$C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 6 Fork: thread t forks thread u
```
$L_u \leftarrow$ set of all possible locks
$C_u \leftarrow C_u \sqcup C_t$
$C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 7 Join: thread t joins thread u
```
$C_t \leftarrow C_t \sqcup C_u$
$C_u[u] \leftarrow C_u[u] + 1$
```

Algorithm 8 Notify: thread t notifies thread u
```
$C_u \leftarrow C_u \sqcup C_t$
$C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 9 NotifyAll: thread t wakes up all waiting threads
```
for all threads $u$ waiting for thread $t$ do
  $C_u \leftarrow C_u \sqcup C_t$
end for
$C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 10 Read: thread t reads variable x
```
if epoch$(t) \notin \{\mathcal{R}_x[t].epoch, \mathcal{W}_x.epoch\}$ then   {If same epoch, no action}
  $\mathcal{R}_x[t].epoch \leftarrow$ epoch$(t)$                            {Update read epoch}
  $\mathcal{R}_x[t].lockset \leftarrow L_t$                                {Update read lockset}
  if $\mathcal{W}_x.epoch \not\preccurlyeq C_t$ then
    check $\mathcal{R}_x[t].lockset \cap \mathcal{W}_x.lockset = \emptyset$   {Check with prior write}
  end if
end if
```

Algorithm 11 Write: thread t writes variable x
```
if epoch$(t) \neq \mathcal{W}_x.epoch$ then                   {If same epoch, no action}
  if $\mathcal{W}_x.epoch \not\preccurlyeq C_t$ then
    $\mathcal{W}_x.lockset \leftarrow \mathcal{W}_x.lockset \cap L_t$
    check $\mathcal{W}_x.lockset = \emptyset$                  {Check with prior write}
  else
    $\mathcal{W}_x.lockset \leftarrow L_t$                    {Update write lockset}
  end if
  $\mathcal{W}_x.epoch \leftarrow$ epoch$(t)$                  {Update write epoch}
  for all threads $t'$ in read map $\mathcal{R}_x$ do          {$O(1)$ amortized time}
    if $\mathcal{R}_x[t'].epoch \not\preccurlyeq C_t$ then
      check $\mathcal{R}_x[t'].lockset \cap L_t = \emptyset$   {Check with prior reads}
    end if
  end for
  $\mathcal{R}_x \leftarrow empty$                             {Clear $\mathcal{R}_x$}
end if
```

Algorithm 12 Volatile Read: thread t reads volatile variable x
```
$C_t \leftarrow C_t \sqcup C_x$
```

Algorithm 13 Volatile Write: thread t writes volatile variable x
```
$C_x \leftarrow C_x \sqcup C_t$
$C_t[t] \leftarrow C_t[t] + 1$
```

Code in a Thread	Code in a Thread
lock l_1	lock l_1
= x	x =
lock l_2	lock l_2
= x	x =
unlock l_2	unlock l_2
lock l_3	lock l_3
= x	x =
unlock l_3	unlock l_3
unlock l_1	unlock l_1
(a) Lemma 1	(b) Lemma 2

Fig. 2. An illustration of the nonnecessity of the lock-subset conditions stated in Lemmas 1 and 2 to enable for the lock-subset optimization.

redundant by Definition 2 (with respect to the first) and also by Lemma 1 as both are in the same epoch. Consider thread `T1` in Figure 1(c). The second read to `x` is not redundant (with respect to the first read to `x`) and also concluded so by Lemma 1 as both reads have different epochs.

However, the lock-subset condition in either lemma is sufficient but not necessary, as illustrated in Figure 2. To see why the condition in Lemma 1 is not necessary, consider the code sequence executed in a thread given in Figure 2(a). The last two reads are redundant with respect to the first by Definition 2. However, in ACCULOCK, the first read shares the same epoch as the second but that epoch is different from the epoch of the third read. So ACCULOCK can ignore the second but must analyze the third. The nonnecessity of the lock-subset condition in Lemma 2 is illustrated similarly using Figure 2(b).

In summary, ACCULOCK removes some redundant accesses in $O(1)$ time in order to keep its performance comparable as FASTTRACK. How such redundancy elimination also helps ACCULOCK detect more races will be clear below.

C. Detecting Data Races

We are now ready to introduce our new $\xrightarrow{accu-hb}$-aware lockset algorithm and examine how ACCULOCK applies it to detect three kinds of data races for concurrent accesses (w.r.t. $\xrightarrow{accu-hb}$): *write-read* (a write concurrent with a later read), *write-write* (a write concurrent with a later write) and *read-write* (a read concurrent with a later write).

The core part of ACCULOCK for race detection is given in Algorithms 10 and 11. As in LOCKSET, L_t holds the set of all locks acquired by thread t at any time, according to Algorithms 4 and 5. ACCULOCK maintains the following two metadata structures for each shared location x:

- $\mathcal{R}_x$ is a read map that maps zero or more threads to $(epoch, lockset)$ pairs for all concurrent reads to x (w.r.t. $\xrightarrow{accu-hb}$) with at most one read from each thread. For each thread t, $\mathcal{R}_x.epoch$ is the epoch for the last non-redundant read in thread t (with some redundant reads to x being removed by Lemma 1) and $\mathcal{R}_x.lockset$ is the lockset protecting x in thread t.
- $\mathcal{W}_x$ is a single $(epoch, lockset)$ pair, where $\mathcal{W}_x.lockset$ records the lockset for x that has consistently protected all concurrent writes to x (w.r.t. $\xrightarrow{accu-hb}$) so far and $\mathcal{W}_x.epoch$ gives the epoch for the last non-redundant write to x among all threads (with some redundant writes to x being removed by Lemma 2).

1) Write-Read Races: At a read in a thread t, as shown in Algorithm 10, ACCULOCK does nothing if the current read is redundant (Lemma 1). Otherwise, ACCULOCK records the epoch and lockset of the current read in $\mathcal{R}_x$ for thread t, by overwriting the prior read, if any. As a result, $\mathcal{R}_x$ keeps the $(epoch, lockset)$'s for all concurrent reads to x to be checked for races with a later write to x.

If the current read is not ordered with the last write made at $\mathcal{W}_x.epoch$, then ACCULOCK checks to see if the current read races with one of the prior writes implicitly represented by their common lockset in $\mathcal{W}_x.lockset$. A data race warning is reported when the current read and one of the prior writes are not protected by a common lock.

2) Write-Write and Read-Write Races: At a write in a thread t, as in Algorithm 11, ACCULOCK does nothing if the current write is redundant by Lemma 2. Otherwise, ACCULOCK checks for a potential race with a prior write. If the current write and the last write made at $\mathcal{W}_x.epoch$ are unordered (by $\xrightarrow{accu-hb}$), ACCULOCK updates $\mathcal{W}_x.lockset$ and reports a race (between the current write and one of the prior writes) when $\mathcal{W}_x.lockset$ becomes empty. If both writes are ordered, then the current write happens after the last one (by $\xrightarrow{accu-hb}$). In this case, $\mathcal{W}_x.lockset$ is reset to L_t, i.e., the lockset protecting the current write. In either case, $\mathcal{W}_x.epoch$ is updated with the current epoch (for the current write).

Afterwards, ACCULOCK checks for races by looping over all reads in $\mathcal{R}_x$ that happen concurrently with the current write protected by the lockset L_t (w.r.t. $\xrightarrow{accu-hb}$). As in Algorithm 2 of FASTTRACK, this **for** loop takes $O(|R_x|) \leqslant O(n)$ time but is amortized over the last $|\mathcal{R}_x|$ analysis steps that take $O(1)$ amortized time each, using the efficient lockset implementation [9], [10]. If no races are detected, the current write happens after all reads in $\mathcal{R}_x$ (in the sense of $\xrightarrow{hb}$). In both cases (whether the current write races with any prior read or not), $\mathcal{R}_x$ is cleared. We reset $\mathcal{R}_x$ this way in order to ensure that ACCULOCK and FASTTRACK can be on a par in terms of analysis overhead incurred. As a result, some races caused by the multiple protecting locks idiom may go undetected but these are rare according to [22] and our experiments.

3) Examples: Let us revisit the four examples in Figure 1.

Figure 1(a). Suppose `T1` acquires lock l_1 before `T2`. In this case, FASTTRACK will not find the race between `A` and `B`. At the first read to `x` in `T1`, ACCULOCK stores the current epoch and the empty lockset for the read into $\mathcal{R}_x[\texttt{T1}]$ so that $\mathcal{R}_x[\texttt{T1}].lockset = \emptyset$. The second read is redundant and thus ignored. When the write in `T2` is analyzed, its protecting lockset is $L_t = \{l_1\}$. (So $\mathcal{W}_x.lockset$ is updated to be $\{l_1\}$.) ACCULOCK detects the race (`A`, `B`) because $\mathcal{R}_x[\texttt{T1}].lockset \cap L_t = \emptyset$, implying that `x` is not consistently locked by a common lock.
If the lock acquisition order is reversed, ACCULOCK will also succeed in detecting the same data race.

- **Figure 1(b).** AccuLock can detect the race between A and B in all six interleavings but FastTrack fails with T1 $\rightarrow$ T2 $\rightarrow$ T3 and T3 $\rightarrow$ T2 $\rightarrow$ T1. Consider T1 $\rightarrow$ T2 $\rightarrow$ T3. The two reads in T1 and T2 are concurrent by $\xrightarrow{accu-hb}$ (but not by $\xrightarrow{hb}$), $\mathcal{R}_x$ records the epochs and locksets for the two reads so that $\mathcal{R}_x[\texttt{T1}].lockset = \{l_2\}$ and $\mathcal{R}_x[\texttt{T2}].lockset = \{l_1, l_2\}$. At the later write in T3, $L_t = \{l_1\}$ holds. AccuLock detects the race as the accesses A and B are not protected by a common lock.
- **Figure 1(c).** AccuLock behaves exactly as FastTrack in order to be fast and avoid false warnings as explained below. Both detectors regard the two reads to x in T1 as happening in program order. If T2 acquires lock l_1 before T1, both detectors will discover the race (A, B). If the lock acquisition order is reversed, both detectors keep only the second read (lines 4 and 6 in Algorithm 1 and lines 2 and 3 in Algorithm 10). So both will miss the race.
- **Figure 1(d).** AccuLock detects this $\emptyset$-race similarly as in Figure 1(a), except that it is a false warning, confirmed later only by the programmer or other means. However, FastTrack does not.

4) AccuLock's Lockset Mechanism: AccuLock aims to find more data races than FastTrack by being mindful about alternate thread interleavings when analyzing a given program execution. It must do so by maintaining comparable performance as FastTrack and limiting false warnings in a controlled manner. Thus, AccuLock is designed to miss a data race (known later by further analysis) that is reported by LockSet or Eraser if detecting the race only causes it to be buried in a flood of accomplishing false warnings.

Therefore, AccuLock exploits the program order included in $\xrightarrow{accu-hb}$ to mimic FastTrack whenever necessary. In Algorithm 10, only the last non-redundant read in each thread is recorded in $\mathcal{R}_x$. In Algorithm 11, only the last non-redundant write among all threads is recorded in $\mathcal{W}_x$, and in addition, on seeing two writes in a row that are ordered by $\xrightarrow{accu-hb}$, AccuLock resets $\mathcal{W}_x.lockset$ to L_t (line 6). Moreover, AccuLock also exploits implicitly the synchronization order induced by lock acquires and releases by clearing $\mathcal{R}_x$ at each write to x to improve both time and space efficiency. This is because the current write either races with some of the prior reads in $\mathcal{R}_x$ or happens after all of them in the sense of $\xrightarrow{hb}$. Finally, by distinguishing the locks protecting reads and writes using $\mathcal{R}_x$ and $\mathcal{W}_x$ and approximating the lock-subset condition efficiently, AccuLock performs the amortized $O(1)$ lockset operations in the **for** loop of Algorithm 11 only infrequently, i.e., on a write when $\mathcal{R}_x$ has $O(n)$ entries.

D. Characterizing Data Races

We give a few properties about AccuLock to show its fulfilment of our design requirements. We supplement this analysis by providing experimental evidence in Section V.

Theorem 1 (Compared with LockSet)**.** *AccuLock reports no more data races than LockSet in any thread interleaving.*

Proof: Let M_a (M_f) be the set of shared memory locations checked for races by AccuLock (LockSet). Due to the use of $\xrightarrow{accu-hb}$ (and also $\xrightarrow{hb}$ in Algorithm 11) in AccuLock, then $M_a \subseteq M_f$ holds. For any $x \in M_a$, if AccuLock detects a race to x, so will LockSet, because AccuLock distinguishes reads and writes to x but LockSet does not when finding the common locks held for x. ■

However, AccuLock may report some (real) data races that Eraser does not since the latter is unsound in its handling of thread-local and read-shared data. A formal comparison between the two can be involved and is thus omitted.

AccuLock misses no races detected by FastTrack when looking for more in alternate thread interleavings.

Theorem 2 (Compared with FastTrack)**.** *Consider a fixed program execution (with the same thread interleaving). If* FastTrack *reports a pair of racy accesses on a shared location x during this execution,* AccuLock *will also report a (not necessarily identical) pair of racy accesses on x.*

Proof: Let there be a racy pair (a, b) on x from FastTrack (implying that either a or b is a write). Then a and b are not ordered by $\xrightarrow{hb}$, and consequently, not by $\xrightarrow{accu-hb}$. Let a' (b') be a (b) or an earlier non-redundant access in the same epoch. Being racy by FastTrack, a and b are not protected by a common lock. Nor are a' and b' according to Lemmas 1 and 2. So (a', b') is racy by AccuLock. ■

In the absence of multiple protecting locks, AccuLock reports only the potential races that it is designed to find.

Theorem 3 ($\emptyset$-Races)**.** *Suppose each location is protected by a fixed lock (or none). Then* AccuLock *reports only $\emptyset$-races.*

Proof: Suppose that AccuLock detects a pair of racy accesses to a shared location. Then one of the two accesses must not be protected by a common lock using a simple case analysis. The rest of the proof follows from Definition 1. ■

In the presence of multiple protecting locks, however, as shown in Figure 1(b), AccuLock may report false warnings that are not $\emptyset$-races, just like LockSet and Eraser. As discussed in [9, p. 409], all such false warnings can be avoided by using sets of sets of locks instead of just sets of locks, which may be costly in practice (if used improperly).

Consider a program given in Figure 3 that is adapted from Figure 1(c) so that (1) the first read to x in T1 is now a write instead, and (2) the write to x in T2 is guarded by not only l_1 but also l_2. Suppose that the write in T2 is made between the two accesses in T1 in the modified program. After the two writes, $\mathcal{W}_x.lockset = \{l_2\}$. At the read in T1, AccuLock will report a false warning, (A, B) for x, since its lockset is $\mathcal{R}_x[\texttt{T1}].lockset = \{l_1\}$, implying that $\mathcal{R}_x[\texttt{T1}].lockset \cap \mathcal{W}_x.lockset = \emptyset$. This false warning is not an $\emptyset$-race as the second read in T1 and the write in T2 are protected by the lock l_1.

The lockset intersection, $\mathcal{W}_x.lockset \leftarrow \mathcal{W}_x.lockset \cap L_t$,

Thread T1	Thread T2
lock l_2	lock l_1
x =	lock l_2
unlock l_2	x = Ⓑ
lock l_1	unlock l_2
= x Ⓐ	unlock l_1
unlock l_1	

Fig. 3. An illustration of a false warning (A, B) that is not an $\emptyset$-race.

performed in Algorithm 11 is the culprit for such false warnings. However, it trades off such imprecision for efficiency. In practice, however, for the 11 benchmarks used in our experiments, ACCULOCK reports only a few non-$\emptyset$-races.

IV. IMPLEMENTATION

We have implemented ACCULOCK and six other dynamic race detectors all in the Jikes JVM. The six other detectors are: ERASER [9] (a well-known imprecise detector based on LOCKSET), RACETRACK [10] (an imprecise hybrid lockset/VC detector), "HYBRID" [21] (a hybrid Lockset/VC detector), DJIT$^+$ [18] (a high-performance VC-based detector), MULTIRACE [18] (a hybrid Lockset/DJIT$^+$ detector), and FASTTRACK [12] (the fastest happens-before detector).

To ensure reliable comparisons, all algorithms were implemented on top of EMPTY as similarly as possible in order to reuse the same data structures such as vector clocks and locksets. EMPTY performs no analysis and is used to measure the instrumentation overhead at compile time as well as the overhead of associating metadata with each monitored object and synchronization object at run time). It is implemented inside the Jikes RVM (version 3.1.0), a high-performance Java-in-Java virtual machine. Jikes RVM's performance is competitive with commercial VMs as of November 2009.

A. Metadata

There are three kinds of metadata for ACCULOCK concerning reads/writes, VCs and locksets. We handle the first two as in PACER [19] and the last as in ERASER [9] and RACETRACK [10].

- *Two words* are added to the header of each object. The first word points to the read/write metadata for each instrumented field x: $\mathcal{R}_x$ and $\mathcal{W}_x$. The second points to the synchronization data, i.e., its VC for a synchronization object. Similarly, a word is added per static field for read/write metadata. If volatile variables are handled by applying Algorithms 12 and 13, a word per (object or static) volatile field is also added for synchronization metadata.
- As in ERASER and RACETRACK, a lockset table is used to record all distinct locksets ever created and to identify a lockset uniquely by its index into the table. Lookups in the table are lock-free while inserts are serialized.

The other six detectors are implemented similarly.

B. Instrumentation

There are two dynamic compilers to translate Java bytecode into native code in the Jikes RVM. The *baseline* compiles each method initially when it first executes. When a method becomes *hot*, the *optimizing* compiler recompiles it at successively higher optimization levels. Our implementation modifies both compilers to add instrumentation to at each interesting program point, such as a synchronization operation, read or write. Only the application code of a program loaded at run time is instrumented. In the optimizing compiler, we use its mostly static intraprocedural escape analysis to filter out thread-local accesses.

C. Reporting Races

All detectors report at most one race for each field monitored. The racy pairs reported for a shared location by different detectors may be different.

V. EXPERIMENTAL EVALUATION

We validate the fulfilment of its design objectives by ACCULOCK by comparing it against six other dynamic detectors using 11 benchmarks, the largest programs ever used as a collection in the literature. Our results show that ACCULOCK is capable of reporting more (real) data races than FASTTRACK, while maintaining comparable analysis overhead (in performance and memory overhead) and limiting the data races reported to be mostly $\emptyset$-races (Definition 1). In addition, neither of the other detectors meets all our design objectives.

A. Methodology

1) Platform: We performed all experiments on a 3.0GHz quad-core Intel Xeon machine running Redhat Enterprise Linux 5 (kernel version is 2.6.18) with 16GB of memory.

2) Benchmark Configuration: We have selected 11 benchmarks that expose different runtime structures and patterns in the following way. We have used all four multithreaded programs in the latest release of the `DaCapo` benchmark suite (`9.12-bach`) [23] that can compile under the Jikes JVM: `xalan`, a test tool for the xerces library to transform XML documents into HTML, `lusearch`, a benchmark using `lucene` to index a set of documents, `avrora`, a simulator running AVR microcontrollers, and `sunflow`, a render processing images using a ray-tracing algorithm. We also include the two multithreaded programs in an older version of `DaCapo` (version `2006-10-MR2`): `hsqldb`, a JDBCbench-like in-memory benchmark and `eclipse`, a (non-GUI) JDT performance test tool for the Eclipse IDE. The other five benchmarks are: `hedc`, a tool to access astrophysics data from Internet [11], `mtrt`, a multithreaded ray-tracing program from SPEC JVM98, `jspider`, a highly configurable and customizable web spider engine [24], `cache4j`, a cache system for Java objects with a simple API and fast implementation [25], and `jcs`, a distributed caching system [26].

For the six `DaCapo` benchmarks, the inputs with default sizes were used (as some of these benchmarks run out of memory on larger sizes). For `mtrt`, the largest input size was

						#Instrumented Times (Slowdowns)								#Race Warnings						
Program	Size (LOC)	#Classes Loaded	#Methods Compiled	#Threads	Base Time (secs)	EMPTY	ERASER	RACETRACK	HYBRID	DJIT$^+$	MULTIRACE	FASTTRACK	ACCULOCK	ERASER	RACETRACK	HYBRID	DJIT$^+$	MULTIRACE	FASTTRACK	ACCULOCK
xalan	265,897	360	2,199	64	4.65	2.34	4.81	4.59	10.19	14.1	14.2	5.58	6.03	24	24	24	6[16]	6[16]	6[16]	36[29]
lusearch	110,960	100	505	64	6.89	2.05	4.26	3.76	5.04	6.24	6.37	3.84	3.75	0	0	0	1[12]	1[12]	1[12]	1[12]
hsqldb	148,481	113	1,012	16	2.74	3.36	7.82	7.78	15.92	–	–	7.73	8.24	9	6	5	–	–	3[4]	3[4]
eclipse	165,366	1,230	9,580	16	27.1	3.06	10.06	10.24	18.4	–	–	9.29	9.62	139	53	87	–	–	17[3]	67[30]
avrora	136,756	397	1,785	6	13.6	1.69	3.55	3.48	4.66	4.5	4.52	3.23	3.4	37	2	3	3	3	3	4[1]
sunflow	108,962	121	986	16	5.56	5.05	41.27	41.67	102.8	79.6	80.1	54.1	51.9	4	3	4	19[7]	19[7]	19[7]	19[24]
mtrt	11,317	38	243	20	1.53	2.73	4.95	4.83	8.31	15.8	15.8	4.81	4.88	12	6	5	6[1]	6[1]	6[1]	6[1]
cache4j	5,061	9	65	64	49.1	1.32	2.29	2.24	3.69	4.27	4.28	2.47	2.47	1	1	1	2	2	2	2
jcs	66,944	70	364	64	32.4	1.7	4.0	3.83	8.71	8.03	7.92	4.69	4.83	3	3	3	3	3	3	5[3]
hedc*	24,924	38	140	30	1.24	1.06	1.07	1.08	1.07	1.08	1.08	1.08	1.09	2	0	1	3[2]	1	3[2]	3[2]
jspider*	18,826	304	1,630	15	33.4	1.05	1.08	1.08	1.07	1.08	1.08	1.07	1.08	8	2	6	7[4]	7[4]	7[4]	7[4]
Average						2.6	9.2	9.1	20.0	18.9	19.0	10.3	10.9	239	100	139	92[42]	88[42]	119[49]	257[130]

TABLE I
BENCHMARK RESULTS. THE TWO MARKED WITH '*' ARE NOT COMPUTE-BOUND AND ARE THUS EXCLUDED WHEN COMPUTING AVERAGE SLOWDOWNS. DJIT$^+$ AND MULTIRACE RAN OUT OF MEMORY ON `hsqldb` AND `eclipse` DUE TO THE 4GB HEAP LIMITATION IN JIKES JVM. THE WARNINGS INSIDE THE BRACKETS ARE GENERATED IF CLASS INITIALIZERS AND OBJECT CONSTRUCTORS ARE ALSO INSTRUMENTED. IN EACH OF THE "#RACE WARNINGS" COLUMNS, THE "AVERAGE" REPRESENTS THE TOTAL NUMBER OF WARNINGS REPORTED BY THAT DETECTOR.

enabled with the option "`-s100`". For `jspider`, it was set up to run on a randomly chosen `URL` using `google`. For `cache4j` and `jcs`, their benchmark inputs were used.

3) Computing Time and Space Overheads: These measurements are the average of 10 runs. The time spent on analyzing a program by a detector does not include the time for recording and printing the stack traces for each racy pair of accesses reported. The benchmarks marked with '*' in Table I are not compute-bound and are excluded when computing the average performance slowdown for a detector.

4) Counting Race Warnings: Dynamically detecting races is challenging as some races occur infrequently. For each program, we report all distinct warnings found in the 10 runs by a detector to ensure a reliable comparison with others.

5) Analysis Configuration: ACCULOCK provides a number of analysis switches, controlling whether to analyze memory locations at the level of fields or objects, whether to distinguish the elements of an array or not, and whether to include the events of volatile reads/writes in $\xrightarrow{accu-hb}$ or not.

Due to space limitations, we will restrict ourselves to the fine-grain analysis performed at the field level. Volatile variables are handled by Algorithms 12 and 13. Finally, all array elements are individually monitored.

We used the default generational mark-region collector with the options as `-Xmx4000M -X:processors=all`.

B. Results and Analysis

Table I lists the size, the number of classes, the number of methods, the number of threads and uninstrumented running times for each program examined. In addition, the "Instrumented Times" columns show the running times of each program under each of the detectors, reported as the ratio to the uninstrumented running time. The variations in slowdowns for different programs are common for different dynamic detectors. The "#Race Warnings" columns give the number of warning produced by each detector. Like ERASER, both HYBRID and RACETRACK suppress "initialization warnings" using ERASER's unsound state machine in handling thread-local and read-shared data. To achieve an apples-to-apples comparison with the other four detectors, DJIT$^+$, MULTIRACE, FASTTRACK and ACCULOCK, class initializers and object constructors are not instrumented. Otherwise, all initialization warnings reported are given inside the brackets.

Here are some observations about the following four detectors when compared to ACCULOCK directly or indirectly:

DJIT$^+$. Like FASTTRACK, ACCULOCK is faster than DJIT$^+$, which always reports the same warning as FASTTRACK as both differ only in how the happens-before relation is represented (by VCs vs. VCs + epochs).

MULTIRACE. This detector has about the same overhead and behaves exactly the same as DJIT$^+$ except that only accesses with an empty lockset concluded by ERASER are checked using VC operations. Due to ERASER's unsound state machine used as discussed below, MULTIRACE may miss real races and report false warnings, as already discussed in [12].

HYBRID. This detector uses what is similar as $\xrightarrow{accu-hb}$ to filter out some potential races from ERASER that are ordered by $\xrightarrow{accu-hb}$. While being effective in some programs, such as `eclipse` and `avrora`, HYBRID can be up to 3$\times$ (in `sunflow`) slower than ERASER and inherits the same imprecise state machine from ERASER. The aggressive lock-subset optimization [18], [21] used in HYBRID for removing redundant accesses can be expensive for some programs.

RACETRACK. By making the opposite tradeoff as HYBRID, this detector runs as fast as ERASER but can be very imprecise since it starts looking for races on a memory location only after it has "observed" some racy evidence or missed some racy accesses regarding the location. By comparing the "RACETRACK" and

"FASTTRACK" columns tallying the warnings found (even they may represent different warnings)), we find that RACETRACK often detects only a small subset of races detected by FASTTRACK in a program (e.g., `sunflow`). On the other hand, ACCULOCK detects all what FASTTRACK does (by Theorem 2 and in practice).

Given the above discussions, it suffices to analyze our results by comparing ACCULOCK and FASTTRACK. Afterwards, we compare ACCULOCK and ERASER only briefly.

1) FASTTRACK Comparison: We first compare the instrumentation overheads incurred by FASTTRACK and ACCULOCK and then examine both detectors in terms of extra race conditions discovered by ACCULOCK.

a) Instrumentation Overheads: Table I shows that ACCULOCK has slightly higher analysis overhead (about 5.8% on average more) than FASTTRACK, when implemented in the same EMPTY framework. Note that ACCULOCK is slightly faster for `lusearch` and `sunflow`. ACCULOCK achieves such comparable performance by leveraging the lightweight epoch representation of VCs as in FASTTRACK and the fast lockset operations as in ERASER. As shown in Table I, ERASER remains to be the fastest of all detectors evaluated.

Program	Memory (MB)	Memory Overhead		
		ERASER	FASTTRACK	ACCULOCK
xalan	106.5	3.98	4.79	4.79
lusearch	73.1	4.89	4.48	4.19
hsqldb	94.1	6.84	6.87	6.9
eclipse	156.5	5.21	5.39	5.54
avrora	48.1	4.89	4.63	5
sunflow	48.1	10.25	7.72	7.75
mtrt	48.5	5.93	6.94	6.7
cache4j	34.1	2.52	2.62	2.21
jcs	59.7	2.36	2.67	2.71
hedc	19.7	1.42	1.43	1.46
jspider	37.8	1.28	1.28	1.28
Average	66.0	4.51	4.44	4.47

TABLE II
COMPARING MEMORY OVERHEAD, WHICH IS THE RATIO OF THE MAXIMUM HEAP SPACE USED DURING ANALYSIS TO THE MAXIMUM HEAP SPACE USED UNDER UNINSTRUMENTED EXECUTION.

Table II shows that ACCULOCK has more or less the same memory overhead as FASTTRACK. Compared with ERASER, both detectors also have similar memory requirements.

Both ACCULOCK and FASTTRACK keep the same set of instrumentation states for a location x. There are three states for reads: Same-Epoch, Exclusive when $|R_x| = 1$ in FASTTRACK or $|\mathcal{R}_x| = 1$ in ACCULOCK, and Read-Shared when $|R_x| > 1$ or $|\mathcal{R}_x| > 1$. There are two states for writes: Same-Epoch and Exclusive (with $|W_x| = 1$ in FASTTRACK or $|\mathcal{W}_x| = 1$ in ACCULOCK always).

Table III gives the number of times each state is entered by all instrumented locations in FASTTRACK and the number of $O(n)$ VC operations performed on synchronization objects. Table IV presents similar statistics for ACCULOCK, together with those for lockset operations. ACCULOCK checks more frequently for races between a write and earlier concurrent reads than FASTTRACK (as shown in the "Exclusive" columns in the two tables) because lock acquire and release ordering events are ignored in $\xrightarrow{accu-hb}$ (but included in $\xrightarrow{hb}$). On the other hand, as shown in the "#VC Ops on Sync Objects" columns, ACCULOCK reduces significantly the number of $O(n)$ VC operations on synchronization objects performed by FASTTRACK. In `jcs`, nearly all synchronization events are volatile reads. Such reduction can be more pronounced on affecting their relative analysis times when the number of threads, n, increases.

In general, ACCULOCK is slightly slower than FASTTRACK in analyzing a program when the number of lockset operations or the number of times the instrumented locations stay in the Read-Shared state or both are relatively high (as in `xalan` and `hsqldb`). For the ray-tracing application `sunflow`, ACCULOCK is faster FASTTRACK since ACCULOCK stays in the same epoch more often. Note that ACCULOCK needs to record the lockset for each non-redundant read. For the two caching applications, `cache4j` and `jcs`, the extra overhead incurred by ACCULOCK over FASTTRACK is slightly higher in `jcs` than `cache4j` as ACCULOCK stays in the Read-Shared state more often in `jcs`. Finally, ACCULOCK is slightly faster than FASTTRACK on `lusearch` because the ratio of the number of $O(n)$ VC operations performed on synchronization objects in FASTTRACK to the number of lockset operations performed by ACCULOCK is relatively high.

b) Effectiveness of Data Race Detection: ACCULOCK is more effective than FASTTRACK in the sense that (1) it detects all real races reported by FASTTRACK on every benchmark used (over 10 runs), as shown in the last three columns of Table V, (2) it reports only $\emptyset$-races in 10 out of the 11 benchmarks used, and (3) it finds more real races among the extra race warnings reported (relative to FASTTRACK).

By Theorem 2, ACCULOCK always finds a superset of races found by FASTTRACK given the same thread interleaving. This condition may or may not hold in two separate runs for the two detectors. However, this theorem holds for the 11 benchmarks used in our experiments (as shown by Column "−F" in Table V), as ACCULOCK uses $\xrightarrow{accu-hb}$, which is less sensitive to thread interleaving than $\xrightarrow{hb}$.

We have analyzed the extra warnings reported by ACCULOCK (in the "+F" columns) for `xalan`, `eclipse`, `avrora` and `jcs` using sets of sets of locks instead of just sets of locks for 10 runs. Only three for `eclipse` are found to be false warnings that are removable using sets of sets of locks. All the rest are $\emptyset$-races (Definition 1), which are the potential races that ACCULOCK is designed to flag for further analysis, as motivated in Section I-B.

Let us examine the $\emptyset$-races listed in the last column of Table V. First of all, ACCULOCK and FASTTRACK report the same set of real races in seven of the 11 programs tested, showing that ACCULOCK is usually precise by refraining from reporting false warnings. We have manually analyzed all $\emptyset$-races reported in three of the remaining four benchmarks,

Program	#Instrumentation States Entered						#VC Ops on Sync Objects $[O(n)]$
	Reads			Writes			
	Same Epoch	Exclusive	Read Shared	Same Epoch	Exclusive $\lvert R_x \rvert = 1$	Exclusive $\lvert R_x \rvert > 1$	
xalan	0.43B	0.16B	43.8M	27.8M	46.3M	4	8.94M
lusearch	0.76B	0.11B	9.84M	0.23B	49.2M	0	3.51M
hsqldb	80.6M	0.13B	47430	1.85M	24.8M	18	9.71M
eclipse	3.3B	0.34B	99.8M	0.75B	0.14B	352	4.9M
avrora	0.82B	0.11B	5.03M	0.34B	42.1M	0.1M	3.8M
sunflow	1.2B	0.22B	2.36B	0.35B	0.35B	6	1642
mtrt	0.17B	3.0M	1.11M	6.34M	18.5M	41	9626
cache4j	29.5M	0.13B	9.5M	0	71.1M	65	44.8M
jcs	26.5M	0.14B	0.32B	29.2M	0.11B	66	0.22B
hedc	32712	37462	1717	7995	2312	0	528
jspider	0.65M	0.11M	5984	0.26M	55633	11	4035

TABLE III
STATISTICS ABOUT FASTTRACK ANALYSIS OPERATIONS.

Program	Instrumentation States Entered						#VC OPs on Sync Objects $[O(n)]$	#Lockset Ops		
	Reads			Writes						
	Same Epoch	Exclusive	Read Shared	Same Epoch	Exclusive $\lvert \mathcal{R}_x \rvert = 1$	Exclusive $\lvert \mathcal{R}_x \rvert > 1$		Lookups	Intersects	Inserts
xalan	0.45B	0.16B	22.3M	26.9M	46.1M	0.03M	131	0.28B	5.42M	0.02M
lusearch	0.73B	0.14B	4.14M	0.23B	51.6M	65	0.85M	6.61M	446	1094
hsqldb	79.8M	0.14B	2.85M	1.74M	25.0M	0.02M	3.07M	0.14B	0.34M	2652
eclipse	3.38B	0.33B	11.7M	0.75B	0.14B	0.01M	1.25M	23.4M	0.26M	8412
avrora	0.82B	0.11B	7.0M	0.34B	40.3M	0.22M	0.43M	8.27	3.61M	10
sunflow	2.85B	0.23B	0.87M	0.35B	0.35B	4	34	1.62M	498	18
mtrt	0.19B	3.94M	0.07M	7.92M	16.5M	42	27	1.61M	194	22
cache4j	29.4M	61.1M	79.9M	0	71.8M	0.02M	64	0.16B	14.6M	3
jcs	0.14B	87.5M	0.26B	29.2M	0.11B	1.42M	0.21B	533	0.21B	13
hedc	0.03M	0.03M	789	7746	2296	0	154	396	38	67
jspider	0.69M	0.11M	3746	0.28M	55633	20	1047	0.16M	16	240

TABLE IV
STATISTICS ABOUT ACCULOCK ANALYSIS OPERATIONS.

Program	-E	+E		-F	+F	
		FP	$\emptyset$-races		FP	$\emptyset$-races
xalan	0	0	19	0	0	30
lusearch	0	0	1	0	0	0
hsqldb	3	0	0	0	0	0
eclipse	108	3	50	0	3	41
avrora	34	0	3	0	0	1
sunflow	0	0	22	0	0	0
mtrt	5	0	1	0	0	0
cache4j	0	0	1	0	0	0
jcs	0	0	2	0	0	2
hedc	1	0	4	0	0	0
jspider	3	0	2	0	0	0

-E/+E: fewer/more than ERASER -F/+F: fewer/more than FASTTRACK
FP: false positives (warnings) removable using sets of sets of locks

TABLE V
COMPARING DATA RACES REPORTED.

`xalan`, `avrora` and `jcs`, as follows:

`jcs`. Both are false warnings that warrant such further analysis in order to eliminate all potential software defects. One warning is related to unprotected accesses to the field `_cache` of an `jcs` object. Both are synchronized by an intervening user-defined barrier followed by a lock acquire. The other is caused by accesses to the field `attr` of a `CacheElement` object via object pooling, for the same reason as demonstrated in Figure 1(d). Both warnings can be suppressed with user annotations to ACCULOCK. How to automate detection of idioms such as object pooling and shared channels remains open.

`avrora`.This is a real race on some elements of an array `Medium$Transmitter$Ticker:transmission.data`, which is always detected by ACCULOCK using both the default input (6 threads) and the large input (26 threads). However, the race is missed by FASTTRACK (and also by PACER [19], another implementation of FASTTRACK with its sampling rate set at 100%) when the default input is used but is detected only with the large input, due to its sensitivity to thread interleaving.

`xalan`. All these are false warnings on 26 object fields, including the field `m_lastFetched` of an object `LocPathIterator`, due to the use of a shared `iterator pool`, which is synchronized itself. However, there is a real race on the field `m_attrs` of an object `ElemDesc` that is detected in all 10 runs by ACCULOCK but only in 4 of the 10 runs

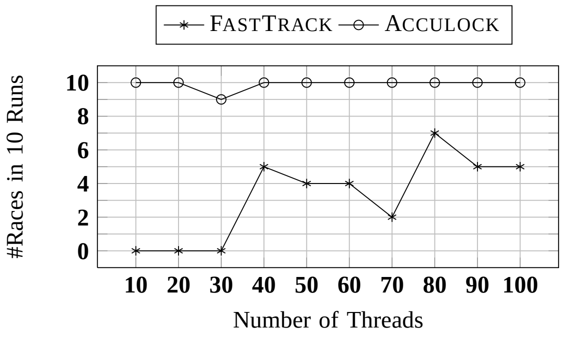

Fig. 4. Sensitivity of AccuLock and FastTrack to thread interleaving on the racy accesses to the field `m_attrs` of an object `ElemDesc` in `xalan`.

by FastTrack, despite that the race is counted for FastTrack in Table I. (Thus, this race is not included in the 30 ∅-races shown in the last column for this benchmark.) Figure 4 demonstrates further that AccuLock is significantly less sensitive to thread interleaving than FastTrack in hunting this race condition. In addition, in a separate experiment running `xalan` with 8 threads for 500 runs, FastTrack fails to detect the race in all the runs but AccuLock succeeds in reporting it in all 500 runs.

2) Eraser *Comparison:* While being the fastest among all seven detectors compared in Table I, Eraser is known to issue more warnings and also miss real races due to its unsound handling of thread-local and read-shared data.

Looking at Table I again, Eraser does not produce many false warnings compared to AccuLock in a few benchmarks. This is because Eraser has succeeded in suppressing unsoundly many false warnings from Lockset (not given in here). However, AccuLock has eliminated them soundly.

Table V also gives the extra race warnings reported by AccuLock relative to Eraser in the "+E" columns. AccuLock happens to also report only three race warnings that are not ∅-races (for `eclipse`). In addition, Eraser did not report the two real races found by AccuLock discussed above in `avrora` and `xalan`. Finally, we list the number of real races missed by Eraser but found by both FastTrack and AccuLock for the 11 programs in the order in which they appear in Table I: 2, 0, 3, 14, 3, 4, 7, 3, 2, 0 and 5.

VI. Conclusion

This paper presents a new dynamic race detector that can detect more data races than FastTrack, the fast happens-before detector, while maintaining comparable performance as FastTrack. The key innovation is to leverage the lightweight epoch representation of vector clocks in FastTrack and deploy a new lockset algorithm to achieve a fine balance of coverage and precision in race detection. These design objectives are met as validated against FastTrack and six other dynamic detectors using 11 benchmarks.

The basic idea behind AccuLock is not tied to FastTrack; it can be incorporated into any future faster happens-before detector to allow a good balance between speed, memory requirement, coverage and precision to be made.

Acknowledgment

Thanks to the anonymous reviewers for feedback on this work. This research is supported by the Australian Research Council (ARC) grants, DP0665581 and DP0987236.

References

[1] R. H. Netzer and B. P. Miller, "What are race conditions? - some issues and formalizations," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 74–88, 1992.

[2] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: context-sensitive correlation analysis for race detection," in *PLDI '06*, 2006, pp. 320–331.

[3] C. Flanagan and S. N. Freund, "Type-based race detection for Java," in *PLDI '00*, 2000, pp. 219–232.

[4] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller, "Automated type-based analysis of data races and atomicity," in *PPoPP '05*, 2005, pp. 83–94.

[5] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," *SOSP' 03*, vol. 37, no. 5, pp. 237–252, 2003.

[6] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *PLDI '06*, 2006, pp. 308–319.

[7] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: static race detection on millions of lines of code," in *ESEC-FSE '07*, 2007, pp. 205–214.

[8] A. Dinning and E. Schonberg, "Detecting access anomalies in programs with critical sections," in *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, ser. PADD '91. ACM, 1991, pp. 85–96.

[9] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Transactions on Computer System*, vol. 15, no. 4, pp. 391–411, 1997.

[10] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: efficient detection of data race conditions via adaptive tracking," in *SOSP '05*, 2005, pp. 221–234.

[11] C. von Praun and T. R. Gross, "Object race detection," in *OOPSLA '01*, 2001, pp. 70–82.

[12] C. Flanagan and S. Freund, "FastTrack: efficient and precise dynamic race detection," *PLDI '09*, Jun 2009.

[13] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded c++ programs," in *PPoPP '03*, 2003, pp. 179–190.

[14] M. Christiaens and K. De Bosschere, "TRaDe, a topological approach to on-the-fly race detection in java programs," in *JVM' 01*, 2001, pp. 15–24.

[15] E. Schonberg, "On-the-fly detection of access anomalies," in *PLDI '89*, 1989, pp. 285–297.

[16] S. L. Min and J.-D. Choi, "An efficient cache-based access anomaly detection scheme," in *ASPLOS' 91*, 1991, pp. 235–244.

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

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

[19] M. D. Bond, K. E. Coons, and K. S. McKinley, "Pacer: proportional detection of data races," in *PLDI '10*, 2010, pp. 255–268.

[20] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: a race and transaction-aware java runtime," in *PLDI '07*. ACM, 2007, pp. 245–255.

[21] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *PPoPP '03*, 2003, pp. 167–178.

[22] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," ser. WBIA '09. ACM, 2009, pp. 62–71.

[23] S. Blackburn, R. Garner, and C. Hoffmann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06*, 2006.

[24] JavaCoding.net, "Jspider, a highly configurable and customizable web spider engine," http://j-spider.sourceforge.net/, 2003.

[25] Y. Stepovoy, "Cache4j, cache for java objects," http://cache4j.sourceforge.net/, 2006.

[26] A. S. Foundation, "Jcs, a distributed caching system written in java," http://jakarta.apache.org/jcs/, 2009.