ACCULOCK: Accurate and Efficient Detection of Data Races

Xinwei Xie[1][2], Jingling Xue[1]*, Jie Zhang[3]

[1] *Programming Languages and Compilers Group, School of Computer Science and Engineering,*
University of New South Wales, NSW, Australia 2052
[2] *School of Computer Science, National University of Defense Technology Changsha, Hunan 410073, China*
[3] *College of Information Science and Technology, Beijing University of Chemical Technology, Beijing, China*

SUMMARY

This paper introduces a new dynamic data race detector, ACCULOCK, to detect data races in Java programs. ACCULOCK is the first hybrid detector that combines lockset and *epoch-based* happens-before for race detection. ACCULOCK analyzes a program execution by reasoning about the subset of the happens-before relation observed with lock acquires and releases excluded, thereby making it less sensitive to thread interleaving than pure happens-before detectors. When this relaxed happens-before relation is violated, ACCULOCK applies a new lockset algorithm to verify the locking discipline by distinguishing reads and writes, thereby making it more immune to false positives than pure lockset detectors. In addition, ACCULOCK is designed to achieve these design objectives by maintaining comparable instrumentation overheads (in both time and space) as FASTTRACK, the fastest happens-before detector available (at least for Java programs). All these properties of ACCULOCK have been validated and confirmed by comparing it against FASTTRACK and five other (pure happens-before, pure lockset or hybrid) detectors, which are implemented in Jikes RVM using a collection of large benchmark programs from a variety of applications. Furthermore, porting ACCULOCK and FASTTRACK to a different platform, RoadRunner, and repeating our experiments yields similar observations in terms of their effectiveness in race detection and instrumentation overheads. Copyright © 2011 John Wiley & Sons, Ltd.

KEY WORDS: data races, happens before, lockset, dynamic analysis

1. INTRODUCTION

Multithreading has become a common programming technique due to the widespread adoption of multicore processors. However, reasoning about the behaviour and correctness of multithreaded programs is notoriously difficult due to non-deterministic thread interleaving. Concurrent accesses to shared data must be synchronized properly; otherwise concurrent errors will unavoidably emerge. 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 can be made (statically) sound [2, 3, 4, 5, 6, 7] but the resulting solutions are 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.

*Correspondence to: Programming Languages and Compilers Group, School of Computer Science and Engineering, University of New South Wales, NSW, Australia 2052
†Email: jingling@cse.unsw.edu.au

Dynamic race detection comes in three flavours: *(pure) happens-before*, *(pure) lockset* and some hybrid of both approaches. Happens-before race detection tracks the happens-before relation [17], a causal relationship induced by program order and synchronization order during a program execution. Earlier examples include TRADE [14] and DJIT$^+$ [18]. Happens-before race detection is sensitive to thread interleaving as it is dynamically sound and precise for one particular execution *only*, As a result, happens-before detectors report all *real (or actual) races* with no false positives in the underlying thread interleaving revealed by one particular test run but may miss races despite repeated test runs. Lockset race detection, as exemplified by ERASER [9], analyzes a program by verifying the locking discipline and reports a *potential race* if two threads access a shared memory location without holding a common lock. A race thus flagged is said to be potential as it may or may not feasibly occur in a program execution. By ignoring the ordering of events during a test run, not only can ERASER find real races that occur in its underlying thread interleaving, it can also report *potential data races* corresponding to other test runs, some of which can be identified to be real races by some further analysis. Thus, lockset detectors are less insensitive to thread interleaving but can be imprecise when many potential races reported turn out to be false positives.

Traditionally, vector clocks (VCs) have been used to represent the happens-before relation. As a result, VC-based happens-before detectors run more slowly than lockset detectors. 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 ERASER. Therefore, FASTTRACK has improved the performance of VC-based happens-before detectors but still retained their precision and sensitivity to thread interleaving. In contrast, ERASER is less sensitive to thread interleaving but suffers from reporting excessively high false positives.

In this paper, we introduce a new dynamic race detector, ACCULOCK, to detect data races in Java programs. ACCULOCK is a hybrid detector that hunts for data races by combining lockset and happens-before. ACCULOCK will report all real races reported by FASTTRACK in a program execution as well as some potential races that may be real races in some other program executions. There are already some hybrid detectors reported, including MULTIRACE [18], HYBRID [19] and RACETRACK [10]. However, these earlier attempts are all VC-based, suffering from either excessive analysis overhead as in MULTIRACE and HYBRID or precision loss as in RACETRACK, as confirmed by their authors, evaluated in Section 4 and further discussed in Section 6. Unlike these existing hybrids, ACCULOCK combines a new lockset analysis with an epoch-based happens-before analysis in a novel way, enabling it to strike a better balance between precision and coverage at comparable performance as FASTTRACK. Validation using benchmarks in production-quality Java virtual machines shows that ACCULOCK can find more real races than pure happens-before detectors (more easily) while reporting significantly fewer false positives than lockset detectors.

```
               Thread T1                  |            Thread T2
------------------------------------------+----------------------------------
                                          | synchronized(queue){
                                          |     ... // not access to
 synchronized(queue){                     |     ... // lastcheckPoint
   lastcheckPoint = currentTime;          | }
 }                                        | if(lastcheckPoint > 0){
                                          |     ...
                                          | }
```

Race field: `weblech.spider.Spider.java:lastcheckPoint`

Figure 1. Data race in the real-world program `weblech`.

Figure 1 illustrates a real data race on the field `lastcheckPoint` in the method `run` of the class `Spider` found in the real-world benchmark `weblech`. This race can always be detected by ACCULOCK but infrequently by any happens-before detector like FASTTRACK. In this program, threads `T1` and `T2` share the same `run` method and the two code fragments are two different parts of `run`. `T1` tries to write to `lastcheckPoint` exclusively once it has acquired the lock

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

(a) Race (A, B) **always** reported by AccuLock but **possibly** by FastTrack.

Thread T1	Thread T2	Thread T3
	lock l_1	
lock l_2	*lock* l_2	*lock* l_1
= *x* Ⓐ	= *x*	*x* = Ⓑ
unlock l_2	*unlock* l_2	*unlock* l_1
	unlock l_1	

(b) Race (A, B) **always** reported by AccuLock but **possibly** by FastTrack.

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

(c) Race (A, B) reported by AccuLock **if and only if** reported by FastTrack.

Thread T1	Thread T2
// create the list of	*lock* l_1
// new nodes in *list*1 Ⓐ	*list*2 = *channel*
lock l_1	*unlock* l_1
channel = *list*1	// process the nodes
unlock l_1	// data in *list*2 Ⓑ

(d) A false positive, (A, B), **always** reported by AccuLock but **never** by FastTrack, caused by shared channels [19].

Figure 2. An illustration of the design philosophy behind AccuLock compared to FastTrack given the **same thread interleaving**.

`queue`. However, `T2` mistakenly tries to read from the variable without attempting to acquire the same lock. This race will occur when `T1` acquires the lock `queue` after `T2` has released it, but not conversely. By tracking the happens-before order (including the synchronization order), FastTrack will detect the race when it is actually seen but this rarely happens in repeated test runs. In contrast, AccuLock does not track the synchronization order. When the read and write to `lastcheckPoint` are unordered by AccuLock's relaxed notion of the happens-before relation (introduced in Section 3.1), AccuLock proceeds to verify the locking discipline, succeeding in detecting the race in each test run (independently of thread interleavings encountered).

1.1. *Overview of the Idea*

1.1.1. Motivation Due to FastTrack, an epoch-based happens-before detector has nearly closed the performance gap with a lockset detector. This has motivated us to design a hybrid detector that combines happens-before and lockset to obtain improved precision and coverage, under the conditions that the detector achieves comparable performance as FastTrack and limits the number of false positives reported compared to Eraser. AccuLock is the first such a solution.

The four design objectives for AccuLock, as illustrated in Figure 2 and explained below, are given 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 by thread scheduling policies used, even when a program is analyzed with varying numbers of threads at different test runs (as illustrated in Figure 8);
3. To limit false positives incurred in a controlled manner;

4. To achieve comparable instrumentation overheads (in time and space) as FastTrack, the fastest happens-before detector (that we know for Java programs).

The motivations for these design objectives are discussed in Section 1.1.2 below. Note that the rationale behind Objective (2) is discussed earlier using the example in Figure 1. To elaborate on this objective further for now, we have tested FastTrack on `xalan` from `DaCapo`. FastTrack reports a particular race, as discussed in Section 4.3.1, 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.

1.1.2. 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. Let us compare and contrast the two detectors using the four examples given in Figure 2.

FastTrack reports all and only real races detected in the thread interleaving induced by a given program execution. In (a) and (c), if `T2` acquires lock l_1 before `T1` does, the racy pair (`A`, `B`) is reported since `B` $\xrightarrow{hb}$ `A` does not hold. If the lock acquisition order is reversed, then `A` $\xrightarrow{hb}$ `B` becomes true. In this alternative thread interleaving, FastTrack will be silent as `A` and `B` do not race. In (b), there are six possible thread interleavings among `T1`, `T2` and `T3`. When the underlying thread interleaving is either `T1` $\rightarrow$ `T2` $\rightarrow$ `T3` or `T3` $\rightarrow$ `T2` $\rightarrow$ `T1`, FastTrack will not report the racy pair (`A`, `B`) that will occur only in one of the remaining four thread interleavings. In (d), FastTrack will never report a race. The middle two examples provide an abstraction of the `multiple protecting lock idiom`, whereby a memory location such as x may be protected by some locks from a collection of multiple locks at its different accesses. The last example provides an abstraction of `shared channels` [19], 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 $\xrightarrow{accu-hb}$ (defined in Section 3.1), 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 the locking discipline. AccuLock finds all real races that FastTrack does during one test run as well as some potential data races that may be likely real races in other test runs. Let us consider Figure 2 again. 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), AccuLock behaves exactly the same as FastTrack (to achieve Objective (4)). Otherwise, in order to catch the race (`A`, `B`) caused by the multiple protecting lock idiom, any lockset algorithm may have to use sets of locksets instead of just locksets [9, p. 409] (to satisfy Objective (3)), but this can be costly and useful only occassionally as validated empirically in Section 3.5. 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).

We define below the potential data races that AccuLock is designed to find in a program.

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 the location x protected by a common lock (i.e., with the set of common locks being $\emptyset$) in that program execution.

We argue that $\emptyset$-races such as the one illustrated in Figure 2(d) should be flagged for further analysis due to the detrimental effects of data races on the reliability of multithreaded software.

Alternatively, such false positives can be eliminated with user annotations so that the missing happens-before relationship is thus established [19].

By using the new lockset algorithm proposed in this paper, ACCULOCK is expected to report usually only $\emptyset$-races in real code. In principle, when multiple protecting locks are not used, as is common in real code, all races reported by ACCULOCK are $\emptyset$-races (Theorem 4). In practice, all races reported by ACCULOCK in our experiments are $\emptyset$-races except for the three false warnings that are not $\emptyset$-races reported from the `eclipse` benchmark.

1.2. Contributions

- We introduce a new hybrid race detector, ACCULOCK, with all properties discussed in Section 1.1. We describe a new lockset algorithm that enables a seamless integration of the lockset and happens-before mechanisms in our hybrid detector 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” [19] and FASTTRACK [12] in Jikes RVM and validated ACCULOCK’s fulfillment of its design objectives using 11 benchmarks, the largest Java programs ever used as a collection in the dynamic analysis literature.
- We have analyzed all these detectors (in terms of performance, memory requirement, precision and coverage) to provide insights for further studies. In particular, ACCULOCK is capable of finding more real data races than FASTTRACK (more easily) when looking for $\emptyset$-races while maintaining comparable analysis overheads.
- We have also ported ACCULOCK and FASTTRACK from Jikes RVM to another Java VM platform, RoadRunner [20]. Repeating our experiments and analysis yields similar observations in terms of their effectiveness in race detection and analysis overheads.
- We also introduce for the first time an epoch-based lockset detector, MULTILOCK-HB, that uses sets of locksets instead of just locksets [9] to detect data races caused by the use of multiple protecting lock idiom (Figures 2(b) and (c)). Replacing the lockset algorithm used in ACCULOCK with MULTILOCK-HB results in only three false warnings in `eclipse` to be suppressed at the expense of a factor-of-three performance slowdown (on average). This discovery is significant for two reasons. First, the ACCULOCK design is justified as the races caused by the multiple protecting lock idiom are rare and thus unnecessarily expensive to detect with MULTILOCK-HB. Second, MULTILOCK-HB can be selectively deployed for certain applications (e.g., `eclipse`) that contain potentially such races.

The rest of this paper is organised as follows. Section 2 provides the background for this work. Section 3 introduces our ACCULOCK algorithm. Section 4 evaluates our ACCULOCK design in Jikes RVM. Section 5 evaluates our ACCULOCK design further in RoadRunner. Section 6 discusses the related work. Section 7 concludes the paper.

2. BACKGROUND

We 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 in the program (Section 2.1). We then describe how FASTTRACK uses epoch clocks to reduce most $O(n)$ VC operations to $O(1)$ (Section 2.2). Finally, we review the basic LOCKSET algorithm and touch upon ERASER, the classic lockset algorithm, on which many other detectors are based (Section 2.3).

2.1. VCs and VC-based Happens-Before Detection

VC detectors soundly and precisely track the (true) happens-before relation $\xrightarrow{hb}$ between two operations A and B (denoted as $A \xrightarrow{hb} B$), where $\xrightarrow{hb}$ is defined as follows:

Program order: If A executes before B in the same thread, then $A \xrightarrow{hb} B$.

Synchronization order: If A and B are synchronization operations from two different threads such that A precedes B (e.g., A releases a lock and B subsequently acquires it), then $A \xrightarrow{hb} B$.

Transitive closure: If $A \xrightarrow{hb} B$ and $B \xrightarrow{hb} C$, then $A \xrightarrow{hb} C$.

In other words, $\xrightarrow{hb}$ for a program execution 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 tracking $\xrightarrow{hb}$ among all synchronization, read and write operations, VC detectors identify *concurrent* accesses to a shared variable 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. More specifically, we have:

$$\begin{array}{rcl} V_1 \sqsubseteq V_2 & \text{iff} & \forall t. V_1(t) \leq V_2(t) \\ V_1 \sqcup V_2 & = & \lambda t.\text{max}(V_1(t), V_2(t)) \end{array}$$

2.1.1. Synchronization Operations Accesses to synchronization objects (*threads*, *locks* and *volatile variables* in Java) are always ordered and never race. 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. Initially, $C_t[t] = 1$ and $C_t[u] = 0$ if $u \neq t$. Similarly, the analysis maintains a vector clock C_l for each lock l and a vector clock C_v for each volatile variable v. Such vector clocks are initialized to $(0, \ldots, 0)$.

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.

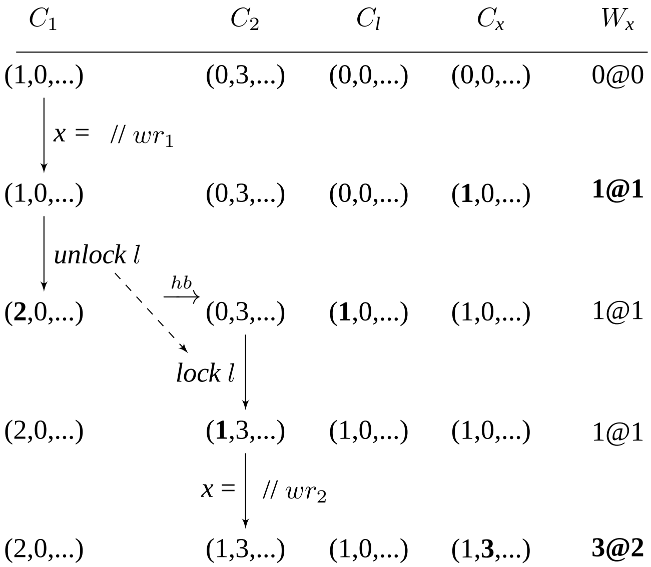
2.1.2. Variable Reads and Writes For each shared variable, i.e., memory location x, which can be an object field or an object itself depending the level of granularity used, the analysis keeps two vector clocks, R_x and W_x, such that the entries $R_x[t]$ and $W_x[t]$ record the clock values of the last 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.

2.1.3. Example Consider the code fragment given in Figure 3, depicting the relevant portion of the instrumentation state of a VC detector. C_1 and C_2 are the vector clocks associated with threads `T1` and `T2`, respectively. C_l and C_x are the vector clocks of lock *l* and variable x, respectively. W_x will be referred to later. The state transitions with respect to the current operation are shown in bold.

At the write wr_1, the analysis updates C_x with the current clock C_1 of `T1`. At the release operation *unlock l*, the analysis updates C_l with C_1 and increments the first entry of C_1. At the acquire operation *lock l*, C_2 is joined with C_l to obtain the up to date clock values, thus capturing the dashed release-acquire happens-before edge as shown. At the subsequent write wr_2, the analysis compares the vector clocks with $\sqsubseteq$ in $O(n)$ time and space complexity:

$$C_x = (1, 0, \ldots) \sqsubseteq (1, 3, \ldots) = C_2$$

Since the check succeeds, the two writes performed by two different threads are not concurrent. So no data race is reported. Clearly, it takes $O(n)$ time and space to keep track of and verify the happens-before relation $\xrightarrow{hb}$ for n concurrent running threads.



C_1	C_2	C_l	C_x	W_x
(1,0,...)	(0,3,...)	(0,0,...)	(0,0,...)	0@0
(1,0,...)	(0,3,...)	(0,0,...)	(**1**,0,...)	**1@1**
(**2**,0,...)	(0,3,...)	(**1**,0,...)	(1,0,...)	1@1
(2,0,...)	(**1**,3,...)	(1,0,...)	(1,0,...)	1@1
(2,0,...)	(1,3,...)	(1,0,...)	(1,**3**,...)	**3@2**

Figure 3. An execution trace for a happens-before detector.

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

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

2.2. *Epochs and* FastTrack

FastTrack has reduced most $O(n)$ VC operations to $O(1)$, by exploiting the following two insights. First, 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 $\xrightarrow{hb}$. Second, 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 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.

Algorithm 3 Read/Write Access [Lockset]: threads t reads or writes x

```
// $CL_x$ is initialized with set of all locks
$L_t \leftarrow$ set of locks held now
if $x$ is a read then
    $L_t \leftarrow L_t \cup \{readers\_lock\}$          {Insert fake lock}
end if
$CL_x \leftarrow CL_x \cap L_t$          {Update lockset}
assert $CL_x \neq \emptyset$          {Check for races}
```

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, $c@t \preccurlyeq VC$ if and only if $c \leqslant VC[t]$, where VC is a vector clock. Following [12], gray shading indicates operations that take $O(n)$ time each, where n is the number of threads.

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 [21]. In [21], 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.

At a read, FastTrack does nothing if the read map R_x is an epoch equal to the current thread's time. Otherwise, it asserts that the last write happens before the current read. Otherwise, the two accesses race. 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 assert takes $O(|R_x|) \leqslant O(n)$ time, which is proportional to the number of entries inserted into R_x by prior reads, 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.

Let us revisit the example in Figure 3. W_x denotes the write epoch of variable x and is initialized to 0@0. At the first write wr_1 to x, FastTrack updates W_x with 1@1. Then FastTrack proceeds as discussed earlier to analyze the lock release and acquire operations on lock l. At the second write wr_2 to x, FastTrack compares W_x with the current thread `T2`'s VC in $O(1)$ time:

$$W_x = 1@1 \preccurlyeq (1, 3, \dots) = C_2$$

As the check passes, the second write wr_2 happens after the first write wr_1. Then W_x is updated with 3@2 to indicate that the last write is performed by `T2` when its clock value is 3.

2.3. *Locksets and* Eraser

The basic Lockset algorithm, as depicted in Figure 3, detects violations of the 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 write to x will cause *readers_lock* to be removed from CL_x.

Consider the execution trace illustrated in Figure 4, depicting the relevant portion of the instrumentation state of Lockset. Initially, CL_x contains the set of all possible locks. At the

Thread T1	**Thread T2**	$L_{\texttt{T1}}$	$L_{\texttt{T2}}$	CL_x
= *x* // rd_1		$\{readers\_lock\}$		$\{readers\_lock\}$
lock l_1		$\{l_1\}$		$\{readers\_lock\}$
x = // wr_1		$\{l_1\}$		$\{\emptyset\}$ **(false warning)**
unlock l_1		$\{\emptyset\}$		$\{\emptyset\}$
	lock l_2		$\{l_2\}$	$\{\emptyset\}$
	= *x* // rd_2		$\{l_2\}$	$\{\emptyset\}$ **(race!)**
	unlock l_2		$\{\emptyset\}$	

Figure 4. An execution trace of LOCKSET.

first read rd_1 performed by `T1`, $L_{\texttt{T1}}$, which represents the set of locks held by `T1`, is updated to $\{readers\_lock\}$ and CL_x is updated by intersecting itself and $L_{\texttt{T1}}$ to find x's lockset CL_x. At the subsequent write wr_1 performed also by `T1`, $CL_x = \emptyset$. So LOCKSET issues a warning to indicate that variable x is not consistently protected by a common lock. However, this is a false positive as LOCKSET does not consider $\xrightarrow{hb}$. At the read rd_2 performed by `T2`, $CL_x = \emptyset$. So LOCKSET issues another warning, which turns out to be a real data race.

In this example, the three accesses to x are executed in the order of rd_1, wr_1 and rd_2 as shown. If the two reads are executed before the write instead, then using the fake lock $\{readers\_lock\}$ will avoid the false positive that would otherwise be reported between rd_1 and rd_2. In addition, the real race between rd_2 and wr_1 will still be detected.

By ignoring $\xrightarrow{hb}$, LOCKSET may result in excessive false positives. To alleviate this, ERASER [9] uses a state machine to handle thread-local and read-shared data. However, this handling is unsound. Consider Figure 4 again. ERASER will classify the two operations rd_1 and wr_1 performed by `T1` as "thread-local" because variable x has been exclusively accessed by `T1` so far. As a result, the ERASER instrumentation state is flagged as "thread-local". At the second read rd_2 by `T2`, ERASER realizes that x has now escaped into a different thread and may be accessed concurrently thereafter. So the instrumentation state transits to "shared". From now on, ERASER behaves identically as LOCKSET. The price paid by ERASER for unsoundly reducing false positives this way is that the racy pair (wr_1, rd_2) is missed.

In practice, we find that ERASER fails to find a number of real races in the benchmarks tested due to the reasons discussed above. Take, for example, `hedc` listed in Table I. ERASER cannot find one real race that can be found by both FASTTRACK and ACCULOCK.

3. ACCULOCK

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, respectively. 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 identically 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 3.1 discusses how to track $\xrightarrow{accu-hb}$. Section 3.2 describes how ACCULOCK approximates the lock-subset condition [18, 19] to both eliminate some redundant race checks and catch more data races than FASTTRACK. Section 3.3 contains the key contribution of the work. It describes how ACCULOCK detects data races by combining our new lockset algorithm and $\xrightarrow{accu-hb}$ tracked with lightweight epochs. Section 3.4 characterizes the data races reported by ACCULOCK with respect to FastTrack. Section 3.5 evolves AccuLock into a more powerful but more expensive detector by using MultiLock-HB, which detects data races using sets of locksets instead of just locksets.

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

```
$L_u \leftarrow \mathit{empty}$
$C_u \leftarrow C_u \sqcup C_t$
$C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 7 Join: thread t joins thread u

```
$C_t \leftarrow C_t \sqcup C_u$
```

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 $\mathcal{W}_x.epoch \not\preceq C_t$ then
        assert $\mathcal{R}_x[t].lockset \cap \mathcal{W}_x.lockset \neq \emptyset$   {Check with prior write}
    end if
end if
```

3.1. *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 $\xrightarrow{hb}$.

Algorithm 11 Write: thread t writes variable x

```
if epoch($t$) $\neq \mathcal{W}_x.epoch$ then                 {If same epoch, no action}
  if $\mathcal{W}_x.epoch \npreceq C_t$ then
    $\mathcal{W}_x.lockset \leftarrow \mathcal{W}_x.lockset \cap L_t$
    assert $\mathcal{W}_x.lockset \neq \emptyset$                 {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 $\mathcal{R}_x[t'].epoch \npreceq C_t$ then
      assert $\mathcal{R}_x[t'].lockset \cap L_t \neq \emptyset$    {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

```
$C_x \leftarrow C_x \sqcup C_t$
$C_t[t] \leftarrow C_t[t] + 1$
```

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 updating C_u. 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 to start a new epoch merely to facilitate the lock-subset optimization described in Section 3.2.

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, whose release-acquire edges are tracked as part of $\xrightarrow{hb}$ in $O(n)$ time and space in FASTTRACK but ignored in $\xrightarrow{accu-hb}$ used 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.

3.2. *Approximating the Lock-Subset Condition*

Traditionally, the lock-subset optimization [18, 19] serves to reduce the number of accesses participating for race checks. Recall that a lockset for an access is the set of locks protecting it.

Definition 2 (Redundant Accesses)
Let there be two accesses a and b to a shared location x made by a thread T_1. Let c be any future access to the same location x made in a different thread T_2. When looking for racy accesses to x by T_1 and T_2, b is said to be *redundant* with respect to a if c races with a whenever c races with b.

As a result, according to this definition, redundant accesses such as b can be ignored when detecting data races between two threads. Based on the lock-subset optimization [18, 19], the following theorem provides a theoretical basis by which redundant accesses can be eliminated for a program with any number of concurrently running threads.

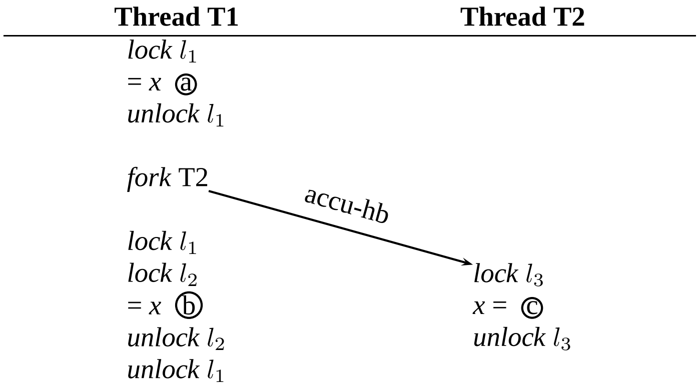


Figure 5. An illustration of Theorem 1 as to why a and b are required not to be separated by a synchronization operation that induces an $\xrightarrow{accu-hb}$ edge.

Theorem 1 (Lock-Subset-based Elimination of Redundant Accesses)
Let there be two accesses a and b to the same shared location x made by a thread such that the two accesses are not separated by any synchronization operation inducing an $\xrightarrow{accu-hb}$ edge with another thread. Then 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.

Proof
Let c be a future access to x made in T_2 that happens after the accesses a and b made in T_1, where $T_1 \neq T_2$. As there does not exist a synchronization operation in T_1 that induces a $\xrightarrow{accu-hb}$ edge during the two accesses a and b between T_1 and T_2, c must be concurrent with a and b, and thus may potentially race with a and b. Given Conditions (1) – (3), we conclude that c races with a whenever c races with b. By Definition 2, b is redundant with respect to a. □

In Theorem 1, the two accesses a and b in a thread are required not to be separated by a synchronization operation in the thread if the synchronization operation induces an $\xrightarrow{accu-hb}$ edge with another thread. Otherwise, as illustrated in Figure 5, b may not necessarily be redundant. In this case, a and b satisfy Conditions (1) – (3). Since $a \xrightarrow{accu-hb} c$, a and c do not race with each other. Since b and c are concurrent, the two accesses race as they are not protected by a common lock.

However, applying Theorem 1 directly to eliminate all redundant checks would be counterproductive as subset operations are costly as validated in our experiments in Section 5. 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 Corollaries 1 and 2 are immediate from Theorem 1.

Corollary 1 (Redundant Reads in ACCULOCK)
Let P_x be the epoch of a prior access (read or write) to x. If $P_x = \textbf{epoch}(t)$, then a future read at $\textbf{epoch}(t)$ is redundant (with respect to the prior access).

Corollary 2 (Redundant Writes in ACCULOCK)
Let P_x be the epoch of a prior write to x. If $P_x = \textbf{epoch}(t)$, then a future write at $\textbf{epoch}(t)$ is redundant (with respect to the prior write)

Consider thread `T1` in Figure 2(a). If P_x represents the epoch of the first read to `x` in the code, then the second read to `x` is redundant by Theorem 1 (with respect to the first) and also by Corollary 1 as both are in the same epoch. Consider thread `T1` in Figure 2(c). The second read to `x` is not redundant

Code in a Thread	**Code in a Thread**
lock l_1	*lock* l_1
= x // rd_1	*x =* // wr_1
lock l_2	*lock* l_2
= x // rd_2	*x =* // wr_2
unlock l_2	*unlock* l_2
lock l_3	*lock* l_3
= x // rd_3	*x =* // wr_3
unlock l_3	*unlock* l_3
unlock l_1	*unlock* l_1
(a) Corollary 1	(b) Corollary 2

Figure 6. An illustration of the nonnecessity of the lock-subset conditions stated in Corollaries 1 and 2 to enable for the lock-subset optimization.

(with respect to the first read to x) and also concluded so by Corollary 1 as both reads have different epochs.

However, the lock-subset condition in either corollary is sufficient but not necessary, as illustrated in Figure 6. To see why the condition in Corollary 1 is not necessary, consider the code sequence executed in a thread given in Figure 6(a). The last two reads, rd_2 and rd_3, are redundant with respect to the first read rd_1 by Theorem 1. However, in ACCULOCK, rd_1 shares the same epoch as rd_2 but that epoch is different from the epoch of rd_3. So ACCULOCK can ignore rd_2 but must analyze rd_3. The nonnecessity of the stronger lock-subset condition in Corollary 2 is illustrated similarly using Figure 6(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 as descried in the following section.

3.3. 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 (with respect to $\xrightarrow{accu-hb}$): *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 (with respect to $\xrightarrow{accu-hb}$) 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 Corollary 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 (with respect to $\xrightarrow{accu-hb}$) 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 Corollary 2).

3.3.1. Write-Read Races At a read in a thread t, as shown in Algorithm 10, ACCULOCK does nothing if the current read is redundant (Corollary 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 the assert statement is evaluated. In this case, 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.

3.3.2. 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 Corollary 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 (with respect to $\xrightarrow{accu-hb}$). 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. Resetting $\mathcal{R}_x$ this way helps ACCULOCK to achieve comparable performance as FASTTRACK. On the other hand, some real races caused by the multiple protecting lock idiom may go undetected but this should happen rarely according to [22] and our empirical validation described in Section 3.5. Indeed, no such races are found in a large collection of real-world benchmarks used in our experiments, which represents a variety of applications in practice.

3.3.3. Examples Let us revisit the four examples given in Figure 2 to compare and contrast ACCULOCK and FASTTRACK in terms of how they detect the data races in these programs.

Figure 2(a). If T1 acquires lock l_1 before T2, then A $\xrightarrow{hb}$ B holds. So (A, B) is not racy according to FASTTRACK. However, the two accesses will be flagged by ACCULOCK as being potentially racy, which turns out to be true if the lock acqusition order is reversed. At the first read to x in T1, ACCULOCK stores the current epoch and the empty lockset for the read into $\mathcal{R}_x[\mathtt{T1}]$ so that $\mathcal{R}_x[\mathtt{T1}].lockset = \emptyset$. The second read is redundant and thus ignored. When the write in T2 is analyzed, its protecting lockset is $L_{\mathtt{T2}} = \{l_1\}$. So $\mathcal{W}_x.lockset$ is updated to be $\{l_1\}$. ACCULOCK detects the race (A, B) because $\mathcal{R}_x[\mathtt{T1}].lockset \cap L_{\mathtt{T2}} = \emptyset$, implying that x is not consistently locked by a common lock. If the lock acquisition order is reversed, however, (A, B) is racy and will be detected by both ACCULOCK and FASTTRACK.

Figure 2(b). The race between A and B does not occur if and only if the thread interleaving is T1 → T2 → T3 or T3 → T2 → T1. So FASTTRACK will report it when it is actually seen in a thread interleaving. In contrast, ACCULOCK reports it each time as a potential race. Consider T1 → T2 → 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 $\mathcal{R}_x[\mathtt{T1}].lockset = \{l_2\}$ and $\mathcal{R}_x[\mathtt{T2}].lockset = \{l_1, l_2\}$. At the later write in T3, $L_{\mathtt{T3}} = \{l_1\}$ holds. ACCULOCK detects the race as the accesses A and B are not protected by a common lock.

Figure 2(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 2(d). ACCULOCK detects this $\emptyset$-race similarly as in Figure 2(a), except that it is a false warning, confirmed later only by the programmer or other means. However, FASTTRACK does not. We argued earlier that such warning should be issued for further analysis.

3.3.4. ACCULOCK's Lockset Mechanism To satisfy its design objectives stated in Section 1.1, 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.

3.4. Characterizing Data Races

We give a few properties about ACCULOCK to show its fulfillment of our design requirements. We supplement this analysis by providing experimental evidence in Section 4.

Theorem 2 (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, as discussed in our experiments.

ACCULOCK misses no real races detected by FASTTRACK when looking for potential races that may occur in alternate thread interleavings.

Theorem 3 (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 Corollaries 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 4 ($\emptyset$-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

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

Figure 7. An illustration of a false warning (A, B) that is not an $\emptyset$-race reported by AccuLock.

accesses must not be protected by a common lock using a simple case analysis. The rest of the proof follows from Definition 1. □

3.5. *Multiple Protecting Locks*

In the presence of multiple protecting locks, AccuLock may miss some real races when they are not identified as potential races just like FastTrack (as illustrated in Figure 2(c)) and report false warnings that are not $\emptyset$-races just like Lockset and Eraser (as illustrated in Figure 7). Given that the idiom is rarely used and costly to handle, we are now in a better position to understand how and why AccuLock behaves this way in order to meet its design objectives. To overcome both problems, we will see clearly the need to keep track of sets of locksets rather than just locksets for a shared location. Unfortunately, doing so is expensive in both time and space. We introduce for the first time an epoch-based lockset solution, MultiLock-HB, to achieve this. In Section 5, the cost-effectiveness of the AccuLock design is justified empirically and the practical benefits of its adaptation with MultiLock-HB are evaluated.

Let us look at the two problems mentioned above with AccuLock with examples:

Figure 2(c): Missing Data Races. Consider this example with the underlying thread interleaving being T1 $\rightarrow$ T2, in which case, A and B do not race. AccuLock, just like FastTrack, does not flag (A, B) as a potential race, even though this actually occurs, for example, when the thread interleaving is T2 $\rightarrow$ T1. Consider how AccuLock works given T1 $\rightarrow$ T2. There are two reads made to x by T1. After the second read is processed, $\mathcal{R}_x[\texttt{T1}]$ keeps only the information for the second read. As a result, $\mathcal{R}_x[\texttt{T1}].lockset = \{l_1\}$, which is the lockset for the second read, rather than $\{l_2\}$. On encountering the write in T2 later, AccuLock has lost the information for the first read in T1 and thus cannot detect the race (A, B). To avoid producing this false negative, the locksets for the two reads in T1 must be recorded.

Figure 7: Reporting false Warnings. Consider now this program that is adapted from Figure 2(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, $\mathcal{W}_x.lockset \leftarrow \mathcal{W}_x.lockset \cap L_t$, performed in Algorithm 11 is the culprit for such false warnings. To avoid producing the false positive in this example, the locksets for the two writes in T1 and T2 must be recorded explicitly.

Our MultiLock-HB design, given in Algorithms 14 – 25, avoids the two aforesaid problems by providing a fully-fledged implementation of the lock-subset optimization stated in Theorem 1 (at significantly higher analysis overheads). Unlike AccuLock, MultiLock-HB relies on a different notion of epoch. In MultiLock-HB, a trace of memory accesses in a thread t are divided into sub-traces by $\xrightarrow{accu-hb}$-inducing synchronization operations in t. Precisely, two accesses in a thread t are in the same epoch if and only if they are not separated by any $\xrightarrow{accu-hb}$-inducing synchronization operation in t. Thus, the epochs in a thread are formed simply by letting the thread tick its clock value at each $\xrightarrow{accu-hb}$-inducing synchronization operation as in Algorithms 16 – 19, 24 and 25. Note that in Algorithms 24 and 25, we have included the effects of volatile variables on $\xrightarrow{accu-hb}$. Otherwise, MultiLock-HB proceeds by treating volatile variables just as lock objects.

Unlike AccuLock, MultiLock-HB now keeps sets of locksets instead of just locksets in the two metadata structures for each shared location x as follows:

- $\mathcal{R}_x[t]$ records a set of $(epoch, lockset)$ pairs for each thread t, where each pair $(epoch, lockset)$ is associated with a prior read made in t when t's clock value is given by $epoch$ and the set of protecting locks for the read is given by $lockset$.

Algorithm 14 Acquire [MultiLock-HB]: thread t acquires lock m

```
$L_t \leftarrow L_t \cup \{m\}$
```

Algorithm 15 Release [MultiLock-HB]: thread t releases lock m

```
$L_t \leftarrow L_t - \{m\}$
```

Algorithm 16 Fork [MultiLock-HB]: thread t forks thread u

```
$C_u \leftarrow C_u \sqcup C_t$
$C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 17 Join [MultiLock-HB]: thread t joins thread u

```
$C_t \leftarrow C_t \sqcup C_u$
$C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 18 Notify [MultiLock-HB]: thread t notifies thread u

```
$C_u \leftarrow C_u \sqcup C_t$
$C_t[t] \leftarrow C_t[t] + 1$
$C_u[u] \leftarrow C_u[u] + 1$
```

Algorithm 19 NotifyAll [MultiLock-HB]: thread t wakes up all waiting threads

```
for all threads $u$ waiting for thread $t$ do
    $C_u \leftarrow C_u \sqcup C_t$
    $C_u[u] \leftarrow C_u[u] + 1$
end for
$C_t[t] \leftarrow C_t[t] + 1$
```

- $\mathcal{W}_x[t]$ is similarly maintained for the writes to x.

Furthermore, MultiLock-HB always maintains the invariant that $\mathcal{R}_x$ and $\mathcal{W}_x$ keeps track of all prior reads and writes to x except the redundant ones removable by Theorem 1. For convenience, this invariant is referred to below as the *RAF-invariant* (short for redundant-access-free invariant).

Let us first consider how a read, R, to a shared location x made in thread t is analyzed in Algorithm 20. Recall that C_t and L_t are the VC and (currently held) lockset of t, respectively. This

Algorithm 20 Read [MultiLock-HB]: thread t reads variable x

```
Update_on_read($\mathcal{R}_x[t], \mathcal{W}_x[t]$)        {Update and remove redundant reads}
for all threads $t'$ in write map $\mathcal{W}_x$ do
  for all $(epoch_{t'}, lockset_{t'}) \in \mathcal{W}_x[t']$ do
    if $epoch_{t'} \not\preceq C_t$ then
      assert $L_t \cap lockset_{t'} \neq \emptyset$        {Check with prior writes}
    end if
  end for
end for
```

Algorithm 21 Write [MultiLock-HB]: thread t writes variable x

```
Update_on_write($\mathcal{R}_x[t], \mathcal{W}_x[t]$)        {Update and remove redundant reads and writes}
for all threads $t'$ in write map $\mathcal{W}_x$ do
  for all $(epoch_{t'}, lockset_{t'}) \in \mathcal{W}_x[t']$ do
    if $epoch_{t'} \not\preceq C_t$ then
      assert $L_t \cap lockset_{t'} \neq \emptyset$        {Check with prior writes}
    end if
  end for
end for
for all threads $t'$ in read map $\mathcal{R}_x$ do
  for all $(epoch_{t'}, lockset_{t'}) \in \mathcal{R}_x[t']$ do
    if $epoch_{t'} \not\preceq C_t$ then
      assert $L_t \cap lockset_{t'} \neq \emptyset$        {Check with prior reads}
    end if
  end for
end for
```

Algorithm 22 Update_on_read [MultiLock-HB]: Update and remove redundant reads

```
for all $(epoch, lockset) \in (\mathcal{R}_x[t] \cup \mathcal{W}_x[t])$ do
  if $epoch = C_t[t] \wedge lockset \subseteq L_t$ then
    return        {Ignore current read}
  else
    $\mathcal{R}_x[t] \leftarrow \mathcal{R}_x[t] \cup \{(C_t[t], L_t)\}$
  end if
end for
for all $(epoch, lockset) \in \mathcal{R}_x[t]$ do
  if $epoch = C_t[t] \wedge L_t \subseteq lockset$ then
    $\mathcal{R}_x[t] \leftarrow \mathcal{R}_x[t] - \{(epoch, lockset)\}$        {Remove prior read}
  end if
end for
```

means that $C_t[t]$ and L_t are the epoch and lockset of the current read R being analyzed. There are two steps. In the first step, Update_on_read given in Algorithm 22 is called to update $\mathcal{R}_x[t]$ so that the RAF-invariant is maintained. If there exists $(epoch, lockset) \in (\mathcal{R}_x[t] \cup \mathcal{W}_x[t])$ for a prior read or write access, where $epoch = C_t[t]$, such that $lockset \subseteq L_t$, then R is redundant with respect to the prior access by Theorem 1. In this case, $\mathcal{R}_x[t]$ and $\mathcal{W}_x[t]$ remain unchanged. Otherwise, we insert $(C_t[t], L_t)$ associated with the current read R into $\mathcal{R}_x[t]$ and remove every $(epoch, lockset) \in \mathcal{R}_x[t]$, where $epoch = C_t[t]$, such that $L_t \subseteq lockset$ since the corresponding prior read is redundant with respect to the current read R by Theorem 1. In the second step, we check for data races between the current read R and every concurrent write recorded previously. We report a race whenever the two accesses are not protected by a common lock, i.e., when the intersection of their locksets is empty.

Algorithm 23 Update_on_write [MULTILOCK-HB]: Update and remove redundant reads/writes

```
for all $(epoch, lockset) \in \mathcal{W}_x[t]$ do
  if $epoch = C_t[t] \wedge lockset \subseteq L_t$ then
    return                                                  {Ignore current write}
  else
    $\mathcal{W}_x[t] \leftarrow \mathcal{W}_x[t] \cup \{(C_t[t], L_t)\}$
  end if
end for
for all $(epoch, lockset) \in \mathcal{R}_x[t]$ do
  if $epoch = C_t[t] \wedge L_t \subseteq lockset$ then
    $\mathcal{R}_x[t] \leftarrow \mathcal{R}_x[t] - \{(epoch, lockset)\}$          {Remove prior read}
  end if
end for
for all $(epoch, lockset) \in \mathcal{W}_x[t]$ do
  if $epoch = C_t[t] \wedge L_t \subseteq lockset$ then
    $\mathcal{W}_x[t] \leftarrow \mathcal{W}_x[t] - \{(epoch, lockset)\}$          {Remove prior write}
  end if
end for
```

Algorithm 24 Volatile Read [MULTILOCK-HB]: thread t reads volatile variable x

```
$C_t \leftarrow C_t \sqcup C_x$
$C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 25 Volatile Write [MULTILOCK-HB]: thread t writes volatile variable x

```
$C_x \leftarrow C_x \sqcup C_t$
$C_t[t] \leftarrow C_t[t] + 1$
```

Similarly, in Algorithm 21, a write W to a shared location x made in thread t is analyzed. There are also two steps. In the first step, Update_on_write given in Algorithm 23 is called to update both $\mathcal{R}_x[t]$ and $\mathcal{W}_x[t]$ so that the RAF-invariant is maintained. If there exists $(epoch, lockset) \in W_x[t]$ for a prior write, where $epoch = C_t[t]$, such that $lockset \subseteq L_t$, then W is redundant with respect to the prior write by Theorem 1. In this case, $\mathcal{R}_x[t]$ and $\mathcal{W}_x[t]$ remain unchanged. Otherwise, we insert $(C_t[t], L_t)$ associated with the current write W into $\mathcal{W}_x[t]$ and, in addition, remove every $(epoch, lockset) \in (\mathcal{R}_x[t] \cup \mathcal{W}_x[t])$, where $epoch = C_t[t]$, such that $L_t \subseteq lockset$ since the corresponding read or write made earlier is redundant with respect to W by Theorem 1. In the second step, we check for data races between the current write W and every concurrent write or read made earlier.

Let us examine how MULTILOCK-HB has avoided the two aforesaid problems:

Figure 2(c). Consider this example again with the thread interleaving `T1` → `T2`. After the two reads are executed in `T1`, we have $\mathcal{R}_x[\texttt{T1}] = \{(1@1, \{l_2\}), (1@1, \{l_1\})\}$. When the write in `T2` is analyzed, $C_{\texttt{T2}} = (0, 1)$ and $L_{\texttt{T2}} = \{l_1\}$. So the write is concurrent with the two reads. The race (`A`, `B`) is detected as the two accesses `A` and `B` are not protected by a common lock.

Figure 7. Consider this example assuming as before that the write in `T2` is made between the two accesses in `T1`. After the two writes are executed, we have $\mathcal{W}_x[\texttt{T1}] = \{(1@1, \{l_2\})\}$ and $\mathcal{W}_x[\texttt{T2}] = \{(1@1, \{l_1, l_2\})\}$. When the second read in `T1` is analyzed, $C_{\texttt{T1}} = (1, 0)$ and $L_{\texttt{T1}} = \{l_1\}$. The second read happens after the first read as both are made in `T1`. However, the second read is concurrent with the write in `T2`. The false positive (`A`, `B`) that would otherwise be reported by ACCULOCK is avoided as `A` and `B` are protected by a common lock, l_1.

Theorem 5 ($\emptyset$-Races Only Warnings)
All race warnings reported by MULTILOCK-HB are $\emptyset$-races.

Proof
In MULTILOCK-HB, all concurrent reads and writes to a shared location x are recorded in $\mathcal{R}_x$ and $\mathcal{W}_x$ except the redundant ones by Theorem 1. Whether two concurrent accesses race or not are checked to see if they share a common lock. By Definition 1, all warnings reported must be $\emptyset$-races. □

4. EXPERIMENTAL EVALUATION IN JIKES RVM

We validate the fulfillment of its design objectives by ACCULOCK by comparing it against six other dynamic race detectors in the Jikes RVM using 11 Java benchmarks, the largest programs ever used as a collection in the literature. (We did not include `weblech` illustrated in Figure 1 since it cannot compile successfully under Jikes RVM.) The six other detectors are: ERASER [9] (a well-known imprecise detector based on LOCKSET), RACETRACK [10] (an imprecise hybrid lockset/VC detector), "HYBRID" [19] (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 known to use for Java programs).

Our experimental 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, none of the other detectors meet all our design objectives.

4.1. *Implementation*

Our implementation is based on the publicly available source code for PACER [21]. In order to ensure reliable comparisons, all race detection algorithms were implemented on top of EMPTY as similarly as possible so as to reuse the same data structures such as vector clocks, readmaps 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. The performance of Jikes RVM is competitive with commercial VMs when compared in November 2009 based on the `Dacapo` benchmark suite.[†]

4.1.1. Metadata There are three kinds of metadata for ACCULOCK concerning reads/writes, VCs and locksets. We handle reads/writes and VCs as in PACER [21] except for some differences as described below. We handle locksets as in ERASER [9] and RACETRACK [10].

- *Two words* are added to the header of each object. The first word points to an array of per-field read/write metadata. For each instrumented field x, $\mathcal{R}_x$ and $\mathcal{W}_x$ are recorded. We therefore trade off memory for speed so that the read/write metadata for an instrumented field are accessed directly, without having to go through a hashing process (as in PACER). The second points to the synchronization data, i.e., its VC for a synchronization object. As PACER is sampling-based, its sampling-related code is suppressed and thus not used in any detector examined in our experiments. 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.

[†] `http://dacapo.anu.edu.au/regression/perf/2006-10-MR2.html`

The other six detectors are implemented similarly.

4.1.2. Instrumentation There are two dynamic compilers to translate Java bytecode into native code in the Jikes RVM. Initially, the baseline compiler compiles each method it first encounters into non-optimized code. When a method becomes hot based on the profiling information gathered by the Jikes RVM, the optimizing compiler re-compiles it into more optimized code to accelerate program execution. Our implementation modifies both compilers to add instrumentation 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.

4.1.3. 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.

4.2. Methodology

4.2.1. 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.

4.2.2. 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 RVM: `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 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.

4.2.3. 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.

4.2.4. 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.

4.2.5. 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.

In this paper, we 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`.

4.3. *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× (in `sunflow`) slower than ERASER and inherits the same imprecise state machine from ERASER. The aggressive lock-subset optimization [18, 19] 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 3 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.

4.3.1. 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.

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 the fastest of all detectors evaluated.

Program	Size (LOC)	#Classes Loaded	#Methods Compiled	#Threads	Base Time (secs)	#Instrumented Times (Slowdowns)								#Race Warnings						
						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							
Total														239	100	139	92[42]	88[42]	119[49]	257[130]

Table I. Benchmark results under Jikes RVM. 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 extra warnings inside the brackets are generated if class initializers and object constructors are also instrumented.

Program	Base 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 (shown in Column 2).

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: (1) Same-Epoch, (2) Exclusive when $|R_x| = 1$ in FastTrack or $|\mathcal{R}_x| = 1$ in AccuLock, and (3) Read-Shared when $|R_x| > 1$ or $|\mathcal{R}_x| > 1$. There are two states for writes: (1) Same-Epoch and (2) 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.

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

Program	#INSTRUMENTATION STATES ENTERED						#VC OPS ON SYNC OBJECTS $[O(n)]$
	READS			WRITES			
	SAME EPOCH	EXCLUSIVE	READ SHARED	SAME EPOCH	EXCLUSIVE		
					$\lvert R_x \rvert = 1$	$\lvert R_x \rvert > 1$	
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			LOOK -UPS	INTER -SECTS	INSERTS
					$\lvert \mathcal{R}_x \rvert = 1$	$\lvert \mathcal{R}_x \rvert > 1$				
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.

benchmarks used, and (3) it finds more real races among the extra race warnings reported (relative to FASTTRACK).

By Theorem 3, ACCULOCK always finds a superset of races found by FASTTRACK under the condition that both detectors analyze a program execution with the same thread interleaving. This condition may or may not hold if each detector is run once on a given program. 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 MULTILOCK-HB for 10 runs. Only three for `eclipse` are found to be false warnings that are removable using sets of locksets as in MULTILOCK-HB (rather than just locksets as in ACCULOCK). 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 1.1.

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

Program	-E	+E		-F	+F		+FA
		FP	$\emptyset$-races		FP	$\emptyset$-races	
xalan	0	0	19	0	0	30	2
lusearch	0	0	1	0	0	0	0
hsqldb	3	0	0	0	0	0	3
eclipse	108	3	50	0	3	41	14
avrora	34	0	3	0	0	1	3
sunflow	0	0	22	0	0	0	4
mtrt	5	0	1	0	0	0	7
cache4j	0	0	1	0	0	0	3
jcs	0	0	2	0	0	2	2
hedc	1	0	4	0	0	0	0
jspider	3	0	2	0	0	0	5

-E/+E: fewer/more than Eraser -F/+F: fewer/more than FastTrack
FP: false positives (warnings) removable using sets of locksetsinstead of just locksets
+FA: real races missed by Eraser but found by both FastTrack and Acculock

Table V. Comparing Acculock with Eraser and FastTrack in terms of data races reported.

analyzed all $\emptyset$-races reported in three of the remaining four benchmarks, `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 2(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 [21], 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 by FastTrack, despite that the race is counted for FastTrack in Table I. (Thus, this race is not included in the 30 $\emptyset$-races shown in the last column for this benchmark.) Figure 8 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.

4.3.2. 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

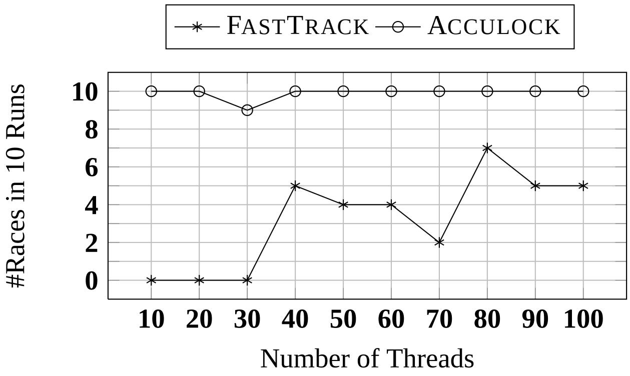


Figure 8. Sensitivity of ACCULOCK and FASTTRACK to thread interleaving on the racy accesses to the field `m_attrs` of an object `ElemDesc` in `xalan`.

suppressing many false warnings that would have otherwise been produced by LOCKSET. However, this is done unsoundly as many real races are also suppressed as discussed below.

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 $\emptyset$-races (for `eclipse`). In addition, ERASER did not report the two real races found by ACCULOCK discussed above in `avrora` and `xalan`. Finally, the "+FA" column gives the number of real races missed by ERASER but found by both FASTTRACK and ACCULOCK.

5. EXPERIMENTAL EVALUATION IN ROADRUNNER

Dynamic race detectors are known to be sensitive to not only the thread interleavings occurring at run time but also the instrumentation frameworks used at compile time. In this section, we show that porting ACCULOCK and FASTTRACK to a different dynamic analysis framework, RoadRunner [20], and repeating our experiments done previously for Jikes RVM yields similar observations about the two detectors. In addition, putting MULTILOCK-HB in action in RoadRunner reveals the cost-effectiveness of ACCULOCK and validates again its design objectives. Unlike Jikes RVM, RoadRunner is a framework designed for developing dynamic analyses for multithreaded Java programs at the bytecode level and has been extended for implementing several dynamic analysis tools [27, 12, 28]. As RoadRunner is a well designed tool for reuse, it is relatively straightforward to port our three analysis algorithms from Jikes RVM.

We have carried our experiments using ACCULOCK, FASTTRACK and MULTILOCK-HB in RoadRunner exactly as in Jikes RVM. The benchmark `eclipse` is not used since it failed to run. Instead, we have replaced it with `weblech`, which is another web crawler often used in the literature.

Performance Slowdowns Table VI gives the "Instrumented Times" (as the ratios to the uninstrumented running times) and race warnings reported. Compared with FASTTRACK, ACCULOCK's exhibits similar analysis overheads across these benchmarks as in Jikes RVM. In particular, ACCULOCK is still able to analyze `lusearch` slightly more efficiently than FASTTRACK. On average, ACCULOCK is about 6.5% slower than FASTTRACK. These results demonstrate further that ACCULOCK achieves comparable performance as FASTTRACK. Meanwhile, MULTILOCK-HB is nearly three times as slow as ACCULOCK (on average) as the lock subset operations can be costly.

Memory Overhead The memory overheads for the three analysis algorithms are compared in Table VII. As in Jikes RVM, ACCULOCK and FASTTRACK exhibit similar memory requirements

Program	Base Time (secs)	#Instrumented Times (Slowdowns)			#Race Warnings		
		FastTrack	AccuLock	MultiLock-HB	FastTrack	AccuLock	MultiLock-HB
xalan	4.65	11.07	12.58	27.27	6	36	36
lusearch	6.89	9.17	9.03	19.14	1	1	1
hsqldb	2.74	20.13	23.49	51.7	3	3	3
weblech*	1.4	1.08	1.09	1.09	4	4	4
avrora	13.6	3.75	4.46	65.14	3	4	4
sunflow	5.56	39.93	40.02	98.56	5	18	18
mtrt	1.53	18.31	18.92	38.12	6	6	6
cache4j	49.1	3.04	4.50	9.49	2	2	2
jcs	32.4	5.86	6.02	16.72	3	5	5
hedc*	1.24	1.08	1.09	1.09	3	3	3
jspider*	33.4	1.07	1.07	1.09	7	7	7
Average		13.9	14.8	40.7			
Total					43	89	89

Table VI. Benchmark results under RoadRunner. The three benchmarks marked with * are not compute-bound and are thus excluded when computing average slowdowns. The warnings are generated with class initializers and object constructors not being instrumented.

in RoadRunner. However, the amount of memory consumed by MultiLock-HB has more than doubled on average as MultiLock-HB has to keep track of a lot more information about concurrent accesses made in the past.

Program	Base Memory (MB)	Memory Overhead		
		FastTrack	AccuLock	MultiLock-HB
xalan	814	4.36	4.77	7.76
lusearch	2076	4.75	4.39	5.47
hsqldb	393	3.85	4.15	12.69
weblech	109	2.81	2.84	2.84
avrora	143	2.51	2.67	21.83
sunflow	1003	1.90	2.42	6.36
mtrt	299	5.89	5.50	15.89
cache4j	549	1.99	2.16	4.13
jcs	551	1.20	1.24	4.13
hedc	132	1.51	1.5	1.52
jspider	125	2.69	2.76	2.79
Average	563.1	3.00	3.13	7.06

Table VII. Comparing memory overhead under RoadRunner, which is the ratio of the maximum heap space used during analysis to the maximum heap space used under uninstrumented execution (in Column 2).

Race Warnings The race warnings reported by the three detectors are listed in the last three columns of Table VI (with "initialization warnings" excluded). Some observations are in order:

- MULTILOCK-HB reports exactly the same set of warnings as ACCULOCK for each benchmark tested. According to Theorem 5, all warnings reported must be $\emptyset$-races, the type of races that ACCULOCK is designed to catch. For these benchmarks, using sets of locksets as in MULTILOCK-HB is not cost-effective as the multiple protecting lock idiom occurs rarely in read-world programs. So ACCULOCK appears to make a good tradeoff between efficiency and precision in detecting data races in practice.
- ACCULOCK reports a superset of race warnings compared to FASTTRACK (Theorem 3). Comparing Tables I and VI, we find that ACCULOCK and FASTTRACK report somewhat different race warnings for `sunflow` due to the differences in thread interleavings encountered and compiler framework used. For all the other common benchmarks used, each detector reports exactly the same race warnings in RoadRunner as in Jikes RVM.
- ACCULOCK is less sensitive to thread interleaving than FASTTRACK. For the data race illustrated earlier for `xalan` in Figure 8, similar results are observed. In particular, when the program is executed with the number of threads being 16, 32 and 64, respectively, ACCULOCK can find the race in each of the 10 runs in each of the three configurations tested. In contrast, FASTTRACK only finds the race in 2, 3, 6 of the 10 runs in each of the three cases. For the data race of the `weblech` benchmark discussed in Figure 1, ACCULOCK succeeds in finding the race in each of the 10 runs. However, FASTTRACK only catches the race in 6 of the 10 runs.

6. RELATED WORK

We supplement our review of related work in Section 1 by focusing only on dynamic race detection. All dynamic analysis algorithms proposed in the literature are based on lockset or happens-before or a combination of both. They can be classified into the four categories below.

Lockset Race Detection The basic idea is to verify the locking discipline [8] that a common lock should be consistently held on each access to a particular shared memory location and report a warning otherwise. This underpins the lockset algorithm introduced for the first time in ERASER [9, 10]. False positives are unavoidable when some synchronization mechanisms or idioms do not follow the locking discipline. Praun et al. proposed *object race detection* [11] that can improve the lockset algorithm's performance by applying escape analysis to filter out thread-local data and detecting data races at the object level instead of the field level. Unfortunately, a reduction in analysis time can lead to even more false positives reported due to its coarser analysis granularity.

Happens-Before Race Detection The basic idea here is to verify the *happens-before* relation, a causal relationship induced by program order and synchronization order during a program execution, represented using vector clocks (VCs) [12, 13, 14, 15, 16]. Unlike lockset, happens-before can therefore be precise by reporting no false positives but requires repeated test runs to increase its coverage. Two earlier analysis algorithms reported are TRADE [14] and DJIT$^+$ [18]. VCs are expensive to implement, exhibiting $O(n)$ complexity in both time and space, where n is the number of threads. FASTTRACK [12] reduces it from $O(n)$ to $O(1)$ for the majority of the VC operations in a program execution. As a result, FASTTRACK and ERASER are comparable in performance overheads for the benchmarks tested in this work. However, each VC update at a synchronization operation is still (n) and can thus be costly for programs with many synchronization operations. In contrast, ACCULOCK does not track the release-acquire edges at synchronization operations. Instead of making VC updates, ACCULOCK relies on a new lockset algorithm to detect more data races.

Goldilocks [29] represents a somewhat different solution. In this detector, the happens-before relation is captured 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].

Hybrid Techniques In the pre-FASTTRACK era, there were two kinds of attempts on combining lockset and happens-before race detection to detect data races [19, 10, 18, 29]. 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 information to reduce false positives in a lockset detector like ERASER. "HYBRID" [19] 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 much 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 discussed in this paper.

This paper has improved our earlier work [30] in two principal directions. First, we have re-implemented ACCULOCK and FASTTRACK in RoadRunner and confirmed again ACCULOCK's fulfillment of its design objectives. This second evaluation is important as dynamic race detectors are often sensitive to thread interleavings exercised and compiler frameworks used. Second, we have introduced for the first time, MULTILOCK-HB, a new epoch-based lockset detector that uses sets of locksets rather than just locksets to detect data races caused by the use of the multiple protecting lock idiom. Replacing this new lockset algorithm used in ACCULOCK with MULTILOCK-HB results in only three false warnings in `eclipse` to be suppressed. However, the price paid for this is nearly a factor-of-three performance slowdown on average for the 10+ benchmarks tested. Two conclusions can be drawn immediately. First, the ACCULOCK design is justified as the races caused by the multiple protecting lock idiom are rare and thus unnecessarily expensive to detect with MULTILOCK-HB. Second, MULTILOCK-HB can be selectively deployed for certain applications (e.g., `eclipse`) that contain potentially such races.

THREADSANITIZER [22] has also recently been developed to combine lockset and happens-before to dynamically detect data races for x86 binaries. 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 (one per thread just like for reads) to increase its chances in detecting races caused by the multiple protecting lock idiom. In contrast, ACCULOCK maintains only the lockset for the last write. However, according to the authors of THREADSANITIZER [22] and our experimental validation and analysis assisted by MULTILOCK-HB, 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 $O(n)$. Given that the multiple protecting lock idiom is rarely used, ACCULOCK is not only efficient but also can accurately pinpoint where a race occurs even if reads and writes are handled asymmetrically (implied in the proof of Theorem 3).

Sampling By sampling only a subset of memory accesses, as in LITERACE [31] and RACEZ [32], certain data races can still be found at significantly reduced instrumentation overheads. PACER [21] improves the prior work by mathematically guaranteeing that the possibility of finding a data race is proportional to the sampling rate. However, as data races are often quite difficult to catch, sampling-based detectors must be run repeatedly to increase their success rates in finding data races.

7. CONCLUSION

This paper presents a new dynamic race detector that can detect more data races than FASTTRACK, the fastest 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 in Jikes RVM and RoadRunner using a collection of Java benchmarks.

The basic idea behind ACCULOCK is not tied to the epoch-based happens-before 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.

ACKNOWLEDGEMENT

Thanks to the anonymous reviewers for their feedback on this work. This research is supported by the Australian Research Council (ARC) grants (DP0665581 and DP0987236) a China Scholarship Council (CSC) grant, and an International Science and Technology Cooperation Program of China (2011DFG13000).

REFERENCES

1. Netzer RH, Miller BP. What are Race Conditions? - Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems* 1992; **1**:74–88.
2. Pratikakis P, Foster JS, Hicks M. Locksmith: Context-Sensitive Correlation Analysis for Race Detection. *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2006; 320–331, doi:http://doi.acm.org/10.1145/1133981.1134019.
3. Flanagan C, Freund SN. Type-based Race Detection for Java. *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2000; 219–232, doi:http://doi.acm.org/10.1145/349299.349328.
4. Sasturkar A, Agarwal R, Wang L, Stoller SD. Automated Type-based Analysis of Data Races and Atomicity. *PPoPP '05: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM: New York, NY, USA, 2005; 83–94, doi:http://doi.acm.org/10.1145/1065944.1065956.
5. Engler D, Ashcraft K. Racerx: Effective, Static Detection of Race Conditions and Deadlocks. *SIGOPS Oper. Syst. Rev.* 2003; **37**(5):237–252.
6. Naik M, Aiken A, Whaley J. Effective Static Race Detection for Java. *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2006; 308–319, doi:http://doi.acm.org/10.1145/1133981.1134018.
7. Voung JW, Jhala R, Lerner S. Relay: Static Race Detection on Millions of Lines of Code. *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ACM: New York, NY, USA, 2007; 205–214, doi: http://doi.acm.org/10.1145/1287624.1287654.
8. Dinning A, Schonberg E. Detecting Access Anomalies in Programs with Critical Sections. *SIGPLAN Not.* 1991; **26**(12):85–96, doi:http://doi.acm.org/10.1145/127695.122767.
9. Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer System* 1997; **15**(4):391–411.
10. Yu Y, Rodeheffer T, Chen W. Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *SOSP '05: Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, ACM: New York, NY, USA, 2005; 221–234, doi:http://doi.acm.org/10.1145/1095810.1095832.
11. von Praun C, Gross TR. Object Race Detection. *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2001; 70–82.
12. Flanagan C, Freund S. FastTrack: Efficient and Precise Dynamic Race Detection. *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* Jun 2009; .
13. Pozniansky E, Schuster A. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. *PPoPP '03: Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003; 179–190, doi:http://doi.acm.org/10.1145/781498.781529.
14. Christiaens M, De Bosschere K. TRaDe, a Topological Approach to On-the-fly Race Detection in Java Programs. *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, USENIX Association: Berkeley, CA, USA, 2001; 15–24.
15. Schonberg E. On-the-fly detection of access anomalies. *PLDI'89: In Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, 1998; 285–297.
16. Min SL, Choi JD. An Efficient Cache-based Access Anomaly Detection Scheme. *Proceedings of the fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IV, ACM: New York, NY, USA, 1991; 235–244, doi:http://doi.acm.org/10.1145/106972.106996.
17. Lamport L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 1978; **21**(7):558–565, doi:http://doi.acm.org/10.1145/359545.359563.

18. Pozniansky E, Schuster A. MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. *Concurrency and Computation: Practice and Experience* 2007; **19**(3):327–340, doi:http://dx.doi.org/10.1002/cpe.v19:3.
19. O'Callahan R, Choi JD. Hybrid Dynamic Data Race Detection. *PPoPP '03: Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM: New York, NY, USA, 2003; 167–178, doi:http://doi.acm.org/10.1145/781498.781528.
20. Flanagan C, Freund SN. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. *PASTE '10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ACM: New York, NY, USA, 2010; 1–8, doi:http://doi.acm.org/10.1145/1806672.1806674.
21. Bond MD, Coons KE, McKinley KS. Pacer: Proportional Detection of Data Races. *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2010; 255–268, doi:http://doi.acm.org/10.1145/1806596.1806626.
22. Serebryany K, Iskhodzhanov T. ThreadSanitizer: Data Race Detection in Practice. *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, ACM: New York, NY, USA, 2009; 62–71, doi:http://doi.acm.org/10.1145/1791194.1791203.
23. Blackburn S, Garner R, Hoffmann C. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *OOPSLA '06: Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* Jan 2006; .
24. JavaCodingnet. JSpider, A Highly Configurable and Customizable Web Spider Engine. `http://j-spider.sourceforge.net/` 2003.
25. Stepovoy Y. Cache4j, Cache for Java Objects. `http://cache4j.sourceforge.net/` 2006.
26. Foundation AS. Jcs, a Distributed Caching System Written in Java. `http://jakarta.apache.org/jcs/` 2009.
27. Flanagan C, Freund SN, Yi J. Velodrome: a Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. 2008; 293–303, doi:http://doi.acm.org/10.1145/1375581.1375618.
28. Flanagan C, Freund SN. Adversarial Memory for Detecting Destructive Races. *SIGPLAN Not.* 2010; **45**(6):244–254, doi:http://doi.acm.org/10.1145/1809028.1806625.
29. Elmas T, Qadeer S, Tasiran S. Goldilocks: a Race and Transaction-aware Java Runtime. *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM: New York, NY, USA, 2007; 245–255, doi:http://doi.acm.org/10.1145/1250734.1250762.
30. Xie X, Xue J. Acculock: Accurate and Efficient Detection of Data Races. *CGO'11: Proceedings of the sixth annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2011; 201–212.
31. Marino D, Musuvathi M, Narayanasamy S. LiteRace: Effective Sampling for Lightweight Data-race Detection. *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009; 134–143, doi:http://doi.acm.org/10.1145/1542476.1542491.
32. Sheng T, Vachharajani N, Eranian S, Hundt R, Chen W, Zheng W. RaceZ: a Lightweight and Non-invasive Race Detection Tool for Production Applications. *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, ACM: New York, NY, USA, 2011; 401–410.