PACER: Proportional Detection of Data Races [*]

Michael D. Bond Katherine E. Coons Kathryn S. McKinley
Department of Computer Sciences
The University of Texas at Austin
{mikebond,coonske,mckinley}@cs.utexas.edu

UT Austin Computer Sciences Technical Report TR-09-35. Under double-blind submission to PLDI 2010.

Abstract

Data races indicate serious concurrency bugs such as order, atomicity, and sequential consistency violations. Races are difficult to find and fix, often manifesting only in deployment. The frequency of these bugs will likely increase as software adds parallelism to exploit multicore hardware. Unfortunately, sound and precise race detectors slow programs by factors of eight or more, which is too expensive to deploy.

This paper presents a precise, low-overhead sampling-based data race detector called PACER. PACER makes a *proportionality* guarantee: it detects *any* race at a rate equal to the sampling rate, by finding races whose first access occurs during a global sampling period. During sampling, PACER tracks all accesses using the sound and precise FASTTRACK algorithm. In non-sampling periods, PACER discards sampled access information that cannot be part of a reported race, *and* PACER simplifies tracking of the happens-before relationship, yielding near-constant, instead of linear, overheads. Experimental results confirm our design claims: time and space overheads scale with the sampling rate, and sampling rates of 1-3% yield overheads low enough to consider in production software. Furthermore, our results suggest PACER reports each race at a rate equal to the sampling rate. The resulting system provides a "get what you pay for" approach to race detection that is suitable for identifying real, hard-to-reproduce races in deployed systems.

1. Introduction

Programs must become more parallel to exploit hardware trends that are producing successive processor generations with additional parallel execution contexts, instead of faster single-threaded performance. Unfortunately, correct and scalable multithreaded programming is quite challenging. In particular, it is notoriously difficult to specify the synchronization, i.e., the ways in which threads may interleave operations on shared data. Too much synchronization hurts performance and causes deadlock, while missing synchronization causes unintended interleavings. A *data race* occurs when two accesses to the same variable, one of which is a write, do not correctly synchronize. While data races are not necessarily errors in and of themselves, they indicate a variety of serious concurrency errors that are difficult to reproduce and debug such as atomicity violations [20], order violations [19], and sequential consistency violations [21]. Because some races occur only under certain inputs, environments, or thread schedules, low overhead race detection for production systems is necessary to help achieve highly robust software.

Static analysis finds races but either does not scale to large programs, reports many false positives, or is intentionally unsound to avoid reporting too many false positives. *Precision* (no false positives) is important because both false and true data race reports take lots of developer time to understand, and developers are not likely to adopt approaches that report false positives. Dynamic analyses typically use either *lockset* or *vector clock* algorithms. Lockset is imprecise, and lockset and vector clock algorithms now have about the same overhead [11], which strongly motivates using precise vector clock algorithms.

Vector clock-based race detection is precise because it tracks the *happens-before* relationship, but unfortunately the race detection analysis for most operations takes $O(n)$ time and space, where n is the number of threads, which does not scale well to many threads. Recently the FASTTRACK vector clock algorithm reduced most analysis from $O(n)$ to $O(1)$ by exploiting the observation that some reads and all writes are totally ordered. However, FASTTRACK still slows programs down by a factor of eight on average. LITERACE addresses the overhead problem by *sampling* [22]. While LITERACE finds many races handily, it uses heuristics that provide no guarantees, incurs $O(n)$ overhead at synchronization operations, and has high space overhead if performed online.

This paper presents a new approach for detecting data races based on sampling called PACER that provides *proportional detection of data races*. PACERguarantees, for each race, a detection rate equal to the sampling rate, and it adds time and space overheads proportional to the sampling rate. PACER builds on the FASTTRACK algorithm, but further reduces overhead through sampling. PACER reports *sampled, shortest races*. Two racy accesses A and B are a *shortest* race if there is no intervening racy access to the same variable. (FASTTRACK also reports shortest races.) They are *sampled* if A occurs in the sampling period. In PACER, B may occur any time later—in any subsequent sampling or non-sampling period. Please refer to the appendix for the proof of this claim. The key insights we use to reduce overhead are as follows. (1) During non-sampling periods, once PACER determines a sampled access cannot be part of a shortest race, it may discard the access metadata to save time and space. (2) During non-sampling periods, we observe that it is not necessary to increment vector clocks because we only ask if sampled accesses happen before the current time. Without increments, vector clock values converge due to redundant synchronization, which we detect and exploit to reduce almost all linear-time analysis to constant-time analysis during non-sampling periods.

PACER's scalable performance makes it suitable for all-the-time use in deployed settings. We envision its use in a distributed debugging paradigm [17; 18] where many deployed instances sample bug-finding instrumentation to increase the chances of finding rare

[*] This work is supported by NSF SHF-0910818, NSF CSR-0917191, NSF CCF-0811524, NSF CNS-0719966, NSF CCF-0429859, a Microsoft Research Ph.D. Fellowship, Intel, IBM, CISCO, Google, and Microsoft. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

bugs. Since PACER guarantees that the chance of finding *any individual race* is equal to the sampling rate, with enough deployed instances, the odds of finding every race become high. Deploying PACER widely and running it all the time is critical since the *occurrence* of some races is quite rare.

We show that for sampling rates of 1 to 3%, PACER adds overheads between 52 and 86%, which is likely low enough for many deployed settings, especially considering the alternative is buggy software. In particular, we show that it avoids nearly all $O(n)$ operations during non-sampling periods. Both sampling and the observer effect complicate race detection evaluation because they change the reported races. However, our careful evaluation suggests that PACER works as designed, finding each dynamic data race with a probability equal to the sampling rate.

In summary, PACER is a "get what you pay for" approach that provides scalable performance and scalable odds of finding *any* race. It is suitable for all-the-time use in deployed systems, where its use could help developers eliminate rare, tough-to-reproduce errors in their software.

2. Background, Motivation, and Requirements

This section describes dynamic race detection algorithms that precisely track the happens-before relationship using vector clocks. It first reviews the happens-before relationship and a GENERIC $O(n)$ (time and space) vector clock algorithm. We describe how the FASTTRACK algorithm replaces most $O(n)$ analysis, where n is the number of threads, with $O(1)$ analysis without losing accuracy. Section 2.3 motivates sampling to reduce overhead, but argues that a prior heuristic approach is unsatisfactory because it may miss races, and has unscalable time and memory overheads. (Section 6 discusses other related work.) Section 3 presents PACER, showing how it builds sampling on top of FASTTRACK, reducing analysis overhead from linear to near-constant when not sampling, while precisely and efficiently reporting races in proportion to the sampling rate.

2.1 Race Detection Using Vector Clocks

The *happens-before* relationship computes a partial order over dynamic program statements [16]. A happens before B, $A \xrightarrow{\text{HB}} B$, if any of the following is true:

- A executes before B in the same thread.
- A and B are operations on the same synchronization variable such that the semantics imply a happens-before edge (e.g., A releases a lock, and B subsequently acquires the same lock).
- $A \xrightarrow{\text{HB}} C$ and $C \xrightarrow{\text{HB}} B$. Happens before is transitive.

Two statements A and B are *concurrent* if $A \not\xrightarrow{\text{HB}} B$ and $B \not\xrightarrow{\text{HB}} A$ and thus not ordered by the happens-before relationship. A *data race* occurs when there are two concurrent accesses to a variable and at least one is a write.

Accesses to synchronization objects are always ordered and never race. Synchronization objects in Java are: *threads, locks,* and *volatile variables*. (We focus on threads and locks to simplify this presentation. Volatile variables differ slightly; see Appendix C.) All other program accesses may race, if the program synchronization does not order them. Potentially racing accesses include *object fields, static fields*, and *array element* accesses in Java. We follow the terminology in the literature: these accesses are on *variables*, and synchronization operations are on *synchronization objects*.

A *vector clock* is indexed by thread identifier: $C[1..n]$. Vector clock algorithms soundly and precisely track the happens-before relationship [16; 23]. Vector clock race detection performs *dynamic analysis* on all synchronization, read, and write operations to track

Algorithm 1 Acquire [GENERIC]: thread t acquires lock m

```
$C_t \leftarrow C_t \sqcup C_m$
```

Algorithm 2 Release [GENERIC]: thread t releases lock m

```
$C_m \leftarrow C_t$
$C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 3 Fork [GENERIC]: thread t forks thread u

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

Algorithm 4 Thread join [GENERIC]: thread t joins thread u

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

Algorithm 5 Read [GENERIC]: thread t reads variable f

```
check $W_f \sqsubseteq C_t$          {Check race with prior writes}
$R_f[t] \leftarrow C_t[t]$
```

Algorithm 6 Write [GENERIC]: thread t writes variable f

```
check $W_f \sqsubseteq C_t$          {Check race with prior writes and reads}
check $R_f \sqsubseteq C_t$
$W_f[t] \leftarrow C_t[t]$
```

the happens-before relationship. It detects concurrent variable accesses and if at least one is a write, it reports a data race.

Synchronization operations. The simplest vector clock race detection algorithm stores a vector clock for each synchronization object, each variable read, and each variable write. For each synchronization object o, the analysis maintains a vector clock C_o that maps every thread t to a clock value c.

Algorithms 1, 2, 3, and 4 show the GENERIC vector clock algorithm at lock acquires and releases, and thread forks and joins. Following Flanagan and Freund [11], gray shading indicates that operations take $O(n)$ time, where n is the number of threads. The vector clock *join* operator $\sqcup$ takes two vector clocks and returns the maximum of each element. For example, if thread t acquires lock m, GENERIC stores the join of t and m's vector clocks into t's vector clock by computing $C_t \leftarrow C_t \sqcup C_m$, which updates each element $C_t[i]$ to $\max(C_t[i], C_m[i])$. When a thread t releases a lock m, the analysis copies the contents of m's vector clock to t's vector clock. It then increments t in t's vector clock.

Variable reads and writes. GENERIC stores the vector clock value for the last read and the last write access by a thread to every variable:

$\boldsymbol{R[1..n]}$	Read vector
$\boldsymbol{W[1..n]}$	Write vector

Algorithms 5 and 6 show the analysis that GENERIC performs at reads and writes. At reads, the analysis checks that prior writes happen before the current thread's vector clock and then updates the read vector's component for the current thread. The analysis is similar at writes, except it checks for races with prior reads *and* writes and updates the write vector.

2.2 FASTTRACK

FASTTRACK is a sound and complete race detection algorithm [11]. It is nearly an order of magnitude faster than prior techniques because instead of $O(n)$ time and space analysis, it replaces all write and many read vector clocks with a scalar and performs constant-time analysis on them. FASTTRACK's insights are as follows. (1) In a race-free program, writes to a variable are totally ordered. (2) In a race-free program, upon a write, all previous reads must happen before the write. (3) The analysis must distinguish between multiple concurrent reads since they all potentially race with a subsequent write. For each write, FASTTRACK replaces the write vector clock with an *epoch* $c@t$, the clock value c at thread t of the last write. This optimization reduces nearly all analysis at reads and writes from $O(n)$ to $O(1)$ time and space. When reads are ordered by the happens-before relation, FASTTRACK uses an epoch for the last read. Otherwise, it uses a vector clock for reads. The function **epoch**(t) is shorthand for $c@t$ where $c = C_t[t]$.

For clarity of exposition, we combine the read epoch and vector clock into a single structure we call a *read map*. A read map R maps zero or more threads t to clock values c. A read map with one entry is an epoch, and we use them interchangeably. A read map with zero entries is equivalent to the initial-state epoch 0@0.

$\boldsymbol{R}$ Read map: $t \rightarrow c$

$\boldsymbol{W}$ Write epoch: $c@t$

FASTTRACK uses the same analysis at *synchronization* operations as GENERIC (Algorithms 1 and 2). Algorithms 7 and 8 show FASTTRACK's analysis at reads and writes.

At a read, if FASTTRACK discovers that the read map is a single-entry epoch equal to the current thread's time, **epoch**(t), it does nothing. Otherwise, it checks whether the prior write races with the current read. Finally, it either replaces the read map with an epoch (if the read map is an epoch already, and it happens before the current read) or updates the read map's t entry.

At a write, if FASTTRACK discovers the variable's write epoch is the same as the thread's epoch, it does nothing. Otherwise, it checks whether the current write races with the prior write. Finally, it checks for races with prior read(s) and clears the read map. The check takes $O(|R_f|)$ time and thus $O(n)$ at most, although it is amortized over the prior $|R_f|$ analysis steps that take $O(1)$ time each. In the case when R_f is an epoch, the *original* FASTTRACK algorithm does *not* clear R_f. Clearing R_f is sound since the current write will race with any future access that would have also raced with the discarded read. We modify FASTTRACK to clear R_f to make it correspond more directly with PACER, which clears read maps and write epochs to reduce space and time overheads during non-sampling periods.

Discussion. FASTTRACK performs significantly faster than prior vector clock-based race detection [11]. Notably, it performs about the same as imprecise lockset-based race detection, but it still slows programs by a factor of eight on average[1] and adds a factor of three space overhead, which is too inefficient for most deployed applications. FASTTRACK's analysis for nearly all read and write operations takes $O(1)$ time; however, its analysis for synchronization variables takes $O(n)$ time. Although synchronization operations account for only about 3% of analyzed operations, they will not scale as the number of threads increases.

2.3 Sampling

A potential strategy for reducing overhead is to *sample* race detection analysis, i.e., execute only a fraction of the analysis. On first glance, sampling seems to have two serious problems. First, if we sample synchronization operations, we will miss happens-before edges and thus report false positive races. Second, because a race involves *two* accesses, if we sample a proportion r of all reads and writes, then we expect to report only r^2 of races (e.g., 0.09% for $r = 3\%$).

LITERACE solves some of these problems [22]. To avoid missing happens-before edges, LITERACE fully instruments all synchronization operations. It then samples read and write operations with a heuristic. It applies the cold-region hypothesis: bugs occur disproportionately in cold code [7]. LITERACE samples at a rate inversely proportional to execution frequency down to a minimum. LITERACE thus cannot make claims on proportionality, since with their minimum rate of 0.1%, a race in hot code will only be reported $0.1\%^2 = 0.0001\%$, i.e., one out of a million times.

LITERACE uses *offline* race detection by recording synchronization, read, and write operations to a log file. Offline analysis performs checks for races in the log if desired, e.g., if an execution fails. We do not believe offline race detection is practical in many cases, e.g., for long-running programs and races that do not cause failures. An *online* implementation of LITERACE requires $O(n)$ analysis for synchronization operations. Furthermore, since it samples code, rather than data, the space overhead is proportional to the data, not the sample rate.

[1] The FASTTRACK implementation executes in pure Java. We estimate an efficient implementation inside a JVM would slow programs by about 3-4X.

Algorithm 7 Read [FASTTRACK]: thread t reads variable f

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

Algorithm 8 Write [FASTTRACK]: thread t writes variable f

```
if $W_f \neq$ epoch$(t)$ then                    {If same epoch, no action}
    check $W_f \sqsubseteq C_t$
    if $|R_f| \leq 1$ then
        check $R_f \sqsubseteq C_t$
        $R_f \leftarrow$ empty                     {New: clear read map}
    else
        check $R_f \sqsubseteq C_t$
        $R_f \leftarrow$ empty
    end if
    $W_f \leftarrow$ epoch$(t)$                    {Update write epoch}
end if
```

2.4 Requirements

While recent work offers significant advances in dynamic, precise race detection, several serious drawbacks limit its applicability. The most serious drawback is operations that require $O(n)$ time and space, which will not scale as the number of threads increases. We believe the following requirements are key for deployable race detection. First, like the approaches just described, race detection needs to be precise to avoid alienating developers with false positives. Second, the time and space impact must be low enough to be acceptable for production software, and must scale with the number of threads. Third, the approach must offer reasonable chances (e.g., chances linearly proportional to the sampling rate) of finding *any* race that occurs in an execution.

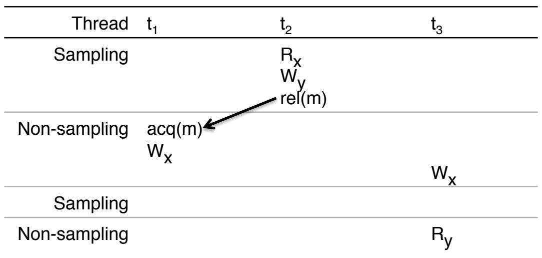

Figure 1: Pacer reports race on y. Race on x is outside sampling period.

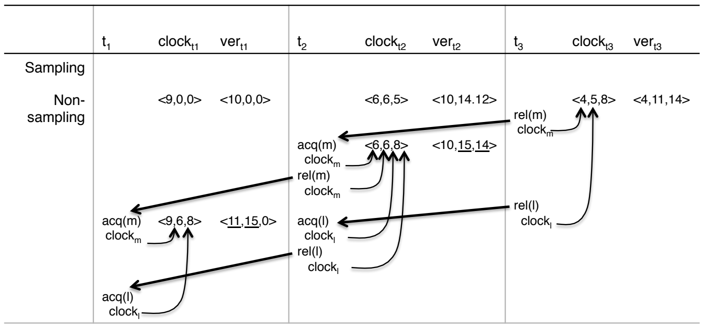

Figure 2: Pacer exploits redundant communication outside sampling period to eliminate vector clock updates and to share vector clocks.

3. PACER

This section presents the PACER sampling race detection approach. PACER is precise: it reports only true races. It guarantees that the probability of detecting *any* race is equal to the sampling rate r. PACER has time and space overheads proportional to its sampling rate r. While some of these overheads are *also* proportional to the number of threads n, decreasing the sampling rate can reduce overall overhead as much as desired.

PACER requires a fairly low sampling rate ($\leq 5\%$) to keep overhead low enough to consider deploying. The chance of finding a race in a given execution is fairly low ($\leq 5\%$) and therefore we envision developers deploying PACER on many deployed instances, as in distributed debugging frameworks [17; 18]. We now describe how PACER (1) guarantees a detection rate for each race equal to the sampling rate and (2) achieves time and space overheads proportional to the sampling rate. We present the algorithms here, and Appendices A and B formalize and prove them.

3.1 Sampling

PACER samples race detection analysis to reduce time and space overheads. PACER divides program execution into *global sampling periods* and *non-sampling periods*, periodically enabling and disabling sampling for all threads. Given randomly chosen sampling periods, PACER samples a proportion r of dynamic operations. It finds any dynamic race with a probability r by guaranteeing to report *sampled* races, defined as follows. Given two accesses A and B, PACER reports the race if A is in the sample period and A is the *last* access that *races* with B. B can occur inside or outside the sampling period. The write-read race on y in Figure 1 shows a simple example. The write at t_2 occurs in the sampling period, and the next read at t_3 is outside the sampling period, and it races with the write. PACER reports this race.

During *sampling periods*, PACER fully tracks the happens-before relationship on all synchronization operations, and variable reads and writes, using FASTTRACK. In *non-sampling periods*, PACER reduces the space and time overheads of race detection by simplifying analysis on synchronization operations and variable reads and writes. For example, PACER incurs no space overhead and performs no work for accesses to variables that were not sampled. Given a sampled access A, PACER stops tracking it and discards its read and write metadata when a subsequent access B means that A will not be the last access to race with a later access.

This guarantee is a little subtle because of the last access requirement. Figure 1 shows an example: the sampled read of x, R_x on t_2, and the non-sampled write W_x at t_1 both race with non-sampled write W_x at t_3. Since a happens-before edge orders R_x and W_x at t_1, when the write occurs, PACER detects there is no read-write race with R_x and stops tracking x. Although there is a race, the happens-before edge indicates that W_x at t_1 must also participate in any race with R_x and it is the last access before the race. Since PACER has probability r of sampling any access, it reports the race between the two writes to x with probability r. That is, if W_x at t_1 had executed in a sampling period, PACER would have reported the race. Similarly, FASTTRACK only reports the race between the two writes to x. The next two sections describe in more detail how PACER maintains accuracy while reducing work.

3.2 Synchronization Operations

The key insights we use to reduce vector clock $O(n)$ analysis in non-sampling periods are as follows.

- During non-sampling periods, we do not need to increment thread time, because we do not need to compare access times in non-sampled periods. We only need to ask if a sampled access happens before the current time.
- When we do not increment time, redundant communication will produce the same vector clock values. By detecting and exploiting this redundancy, we eliminate redundant vector clock joins and copies, reducing time and space overhead.

In a non-sampling period, the analysis stops incrementing thread vector clocks, detects redundant vector clock values, and shares them. A non-sampling period is *timeless*: it eliminates all increments. When two threads communicate with synchronization, they will only copy or join the vector clock. This information is sufficient to track happens-before during non-sampling periods, since we do not compare two non-sampled vector clocks to detect races, we only compare sampled to non-sampled vector clocks to find races. Vector clocks will converge when communication is redundant. PACER detects this redundancy and avoids linear-time vector clock operations.

Consider Figure 2. In timeless periods, only lock acquire, fork, and join operations change the vector clock values. Therefore, the lock_m release and lock_l release can share t_3's vector clock. Furthermore, we detect redundant happens-before relations. PACER must perform the join for the first lock_m acquire on t_2, but the lock_l acquire receives a redundant clock value. PACER detects this case, performs no join, and shares vector clocks. PACER thus performs all non-redundant synchronization joins and copies.

To detect redundant communication, we introduce *vector clock versions* and *version vectors*.[2] PACER assigns a version number to every unique vector clock value a thread observes. It starts the version at zero and increments the version number every time the thread's vector clock changes due to a join or increment. Every thread stores a *version vector* that records the latest version number for all threads it has "received" via a join. We also store a *version epoch* $v@t$ for each lock that stores the last thread t and version v, if any, that released this lock (i.e., copied it). When thread t releases a lock, it sets the lock's version epoch to $v@t$ where v is thread t's current version.

[2] These are not the same as version vectors used in distributed systems [25].

Algorithm 9 Vector clock copy [PACER]: $C_m \leftarrow C_t$

```
if not $sampling$ then
    setShared($clock_t$, true)                 {Share the vector clock}
    $clock_m \leftarrow_{shallow} clock_t$
else
    $clock_m \leftarrow clock_t$               {Deep element-by-element copy}
    setShared($clock_m$, false)
end if
$vepoch_m \leftarrow$ vepoch($t$)              {Update $m$'s version epoch}
```

Algorithm 10 Vector clock increment [PACER]: $C_t[t] \leftarrow C_t[t] + 1$

```
if $sampling$ then                             {If not sampling, no action}
    if isShared($clock_t$) then
        $clock_t \leftarrow$ clone($clock_t$)    {First clone $clock_t$, if shared}
        setShared($clock_t$, false)
    end if
    $clock_t[t] \leftarrow clock_t[t] + 1$
    $ver_t[t] \leftarrow ver_t[t] + 1$         {Update $t$'s version}
end if
```

Algorithm 11 Vector clock join [PACER]: $C_t \leftarrow C_t \sqcup C_m$

```
Let $v@u$ = vepoch($m$)                        {Is $m$'s vector clock newer}
if $v@u \neq$ null $\wedge$ $ver_t[u] < v$ then   {than thread $u$'s?}
    if $clock_m \not\sqsubseteq clock_t$ then    {Need to update $clock_t$?}
        if isShared($clock_t$) then
            $clock_t \leftarrow$ clone($clock_t$)
            setShared($clock_t$, false)
        end if
        $clock_t \leftarrow clock_t \sqcup clock_m$
        $ver_t[t] \leftarrow ver_t[t] + 1$     {Update version with $clock_t$}
    end if
    $ver_t[u] \leftarrow v$                    {New version $v$ of thread $u$'s vector clock}
end if
```

Algorithm 12 Read [PACER]: thread t reads variable f

```
if $sampling \vee (R_f \neq$ null $\vee W_f \neq$ null$)$ then
    check $W_f \sqsubseteq clock_t$
    if $W_f \neq$ epoch($t$) then              {If same epoch, no action}
        if $sampling$ then
            if $R_f \sqsubseteq clock_t$ then
                $R_f \leftarrow$ epoch($t$)    {Update read map}
            else
                $R_f[t] \leftarrow clock_t[t]$
            end if
        else
            $R_f[t] \leftarrow$ null           {Discard $R_f[t]$ only}
            if $isEmpty(R_f)$ then
                $R_f \leftarrow$ null
            end if
        end if
    end if
end if
```

Algorithm 13 Write [PACER]: thread t writes variable f

```
if $sampling \vee (R_f \neq$ null $\vee W_f \neq$ null$)$ then
    check $R_f \sqsubseteq clock_t$            {Check for race with prior access(es)}
    check $W_f \sqsubseteq clock_t$
    if $W_f \neq$ epoch($t$) then              {If same epoch, no action}
        if $sampling$ then
            $W_f \leftarrow$ epoch($t$)        {Update write epoch}
            $R_f \leftarrow$ null              {Discard read map}
        else
            $W_f \leftarrow$ null              {Discard write epoch and read map}
            $R_f \leftarrow$ null
        end if
    end if
end if
```

In non-sampling periods, PACER performs a *shallow* copy of the vector clock to save time and space, since vector clocks will change infrequently in non-sampling periods. If a subsequent synchronization requires an update to a shared vector clock, PACER clones the vector clock before modifying it. At a lock acquire, PACER compares the lock's version epoch and thread's version vector to decide whether it needs to perform the join.

More formally, PACER uses the following metadata for all synchronization objects (threads, locks, and volatiles):

$\boldsymbol{clock_o[1..n]}$ Vector clock.

Each thread has the following additional metadata:

$\boldsymbol{ver_t[1..n]}$ Version vector. Each element $ver_t[u]$ is the latest version received from thread *u* via joins.

Locks (and volatiles) have the following additional metadata:

$\boldsymbol{vepoch_m}$ Version epoch $v@t$. If nonnull, $clock_m$ is equal to version v of thread t's vector clock.

The function **vepoch**(o) is defined for any synchronization object o. For a thread t, **vepoch**$(t) \equiv v@t$ where $v = ver_t[t]$. For a lock m, **vepoch**$(m) \equiv vepoch_m$. The functions **isShared**(), **setShared**(), and **clone**() support sharing of one vector clock by multiple synchronization objects and cloning to eliminate sharing.

PACER performs the same analysis at synchronization operations as GENERIC and FASTTRACK (Algorithms 1, 2, 3, and 4). However, it *redefines* the low-level vector clock operations *copy, increment,* and *join*. Algorithms 9, 10, and 11 show how PACER defines these operations.

Algorithm 9 shows how we redefine vector clock copy. In a non-sampling period, PACER performs a shallow copy of the synchronization object, i.e., then m and t share vector clocks ($clock_m = clock_t$). This sharing is likely worthwhile because the thread's vector clock is likely to have the same value for a while. In a sampling period, sharing would be useless because the algorithms increment thread vector clocks immediately afterwards, so PACER performs a deep copy (i.e., element-by-element) of $clock_t$ to $clock_m$. The vector clock copy then assigns t's version epoch to m.

Algorithm 10 redefines vector clock increment. It does nothing in a non-sampling period. Otherwise, if a prior non-sampling period introduced a shared vector clock, the increment first clones $clock_t$. It then increments the vector clock and its version number.

Algorithm 11 shows PACER's redefined vector clock join. The algorithm first avoids the join altogether when t's version for u is greater than v (where $v@u$ = **vepoch**(m)); no work is needed since we know $clock_m \sqsubseteq clock_t$. Otherwise, a join *may* be required. The algorithm checks whether a join will actually change $clock_t$ (if $clock_m \not\sqsubseteq clock_t$), to avoid incrementing $ver_t[t]$ unnecessarily. If the join is not redundant, the algorithm performs the join. Since the clock changes, the algorithm clones the clock if it is shared and increments the version. Algorithm 11 is only appropriate when the target of the join is a thread vector clock. Appendix C provides the details of how PACER redefines the join into a volatile's clock so that it can often perform a shallow copy.

To correctly detect a race whose first access is in a sampling period but occurs before any synchronization operations, PACER increments each thread's vector clock at the start of a sampling period, i.e., $\forall t \; C_t[t] \leftarrow C_t[t] + 1$.

In practice, versions and shallow copies avoid nearly all $O(n)$ analysis on joins and copies during non-sampling periods. We do not currently know the expected worst case (assuming an adversarial program). We plan to explore it in future work. Section 5.4 shows how in practice versions and shallow copies avoid nearly all $O(n)$ analysis in non-sampling periods.

3.3 Reads and Writes

A race is a sampled race (i.e., a race PACER reports) if its first access is sampled and is the last access to race with its second access. In sampling periods, PACER simply performs the FASTTRACK algorithm. In non-sampling periods, PACER does not record read and write accesses, and it discards any read or write accesses that FASTTRACK would have overwritten or discarded. Thus, PACER reports a race if and only if the first access is sampled and FASTTRACK would report it.

PACER defines the read map and write epoch similarly to FASTTRACK, except that they may be **null**. A **null** read map or write epoch is equivalent to the epoch 0@0. Using **null** values to represent no read or write information helps save space and enables fast common-case checks in non-sampling periods.

Algorithms 12 and 13 show PACER's analysis for read and write operations. In both sampling and non-sampling periods, the analysis first checks if PACER is in a non-sampling period and both R_f and W_f are **null**. If so, the analysis performs no action. Otherwise, both analyses then check for races with prior accesses. The next behavior depends on whether PACER is in a sampling period. If so, it updates the read map and write epoch exactly as FASTTRACK would. If not, it discards whatever read and write accesses FASTTRACK would either replace or discard. In particular, the analysis for a read discards zero or one prior read accesses. If the read map becomes empty, it assigns **null** to it. The analysis for a write always **null**s the read map and write epoch.

4. Implementation

We have implemented PACER in Jikes RVM 3.1.0, a high-performance Java-in-Java virtual machine [2].[3] Jikes RVM's performance is competitive with commercial VMs as of November 2009.[4]

Metadata. Our implementation adds *two words* to the header of every object. The *first* word points to an efficient hash table that maps field (variable) offsets to field read/write metadata. This flexible structure uses space only for metadata that PACER has not discarded. When an object has no per-field metadata, instrumentation sets variables the header word to null. The *second* header word is a reference to the object's synchronization metadata if the object has been locked (Java programs may synchronize on any object).

We use two words per object for ease of design. An alternative implementation could use less space by using indirection. The time and space overheads reported in Section 5.4 include the cost of these extra header words.

Similarly, the implementation adds a word per static field for read/write metadata, which is null if PACER has discarded the field's metadata. It adds a word per (object or static) volatile field for synchronization metadata.

Instrumentation. Jikes RVM uses two dynamic compilers to transform Java bytecode into native code. The *baseline* compiler initially compiles each method when it first executes. When a method becomes hot, the *optimizing* compiler recompiles it at successively higher optimization levels. Our implementation modifies both compilers to add instrumentation to the application at synchronization operations and at reads and writes to potentially shared data (i.e., at most object and static field references). In the optimizing compiler, the new PACER compiler pass uses Jikes RVM's existing static escape analysis to identify accesses to provably local data, which it does not instrument.

[3] http://www.jikesrvm.org/

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

Program	$r = 1\%$	$r = 3\%$	$r = 5\%$	$r = 10\%$	$r = 25\%$
eclipse	1.0±0.2	3.0±0.4	4.8±0.6	9.5±0.7	24.1±1.0
hsqldb	0.5±0.6	2.8±1.3	5.1±1.4	10.8±1.1	26.5±1.8
xalan	1.0±0.0	3.0±0.1	5.0±0.2	10.1±0.4	24.9±0.7
pseudojbb	0.8±0.4	3.0±0.4	5.0±0.5	10.1±0.7	25.5±1.4

Table 1: Effective sampling rates (± one standard deviation) for specified PACER sampling rates.

The optimizing compiler inserts the following instrumentation at reads and writes:

```
// instrumentation
if (sampling || o.metadata != null) {
  slowPath(o, offset_of_f, siteID);
}
// original field read (similarly for write)
... = o.f;
```

The global variable sampling is true if and only if PACER is in a sampling period. The variable o.metadata is the object's first header word, which is null if all the object's field read/write metadata has been discarded. Section 5.4 shows that when the condition is false at run time, the overhead of this check is about 18%.

Our implementation uses low-level synchronization (compare-and-swap) to properly synchronize accesses to synchronization and read/write metadata.

Reporting Races. PACER records the program location (*site*) corresponding to each write epoch and read map entry. When it detects a race, this site is the *first* access. The *second* access is simply the current program location.

Sampling. The implementation turns sampling on and off at the end of garbage collections. Our experiments use the default generational mark-region collector [5]. Nursery collections occur frequently, every 32 MB of allocation. At the end of a collection, we turn on sampling with a probability of r via pseudo-random number generation. At first glance, that should sample a random fraction r of program reads and writes. However, since race detection allocates a lot of metadata while sampling, collections occur more frequently and consequently less program work occurs between two collections during sampling. We correct for this problem by measuring program work in terms of synchronization operations, which are independent of sampling. We compute the number performed during sampling and non-sampling periods, and adjust the probability of entering a sampling period accordingly.

Table 1 shows the actual, *effective sampling rates* (plus or minus one standard deviation) that this mechanism achieves for various *specified* (target) sampling rates on our benchmarks (see Section 5 for benchmark details). The implementation typically achieves an effective rate very close to the specified rate. The effective rate is sometimes *lower*, e.g., hsqldb at a 1% sampling rate, because the mechanism for eliminating sampling bias does have not enough opportunity to observe and correct for bias.

5. Results

This section evaluates the accuracy and performance of PACER. It first presents the experimental platform and characterizes the

	Threads		Races $\forall r$ 1,234 trials		Races at r = 100% 50 trials		
Program	Total	Max live	≥ 1	≥ 5	≥ 1	≥ 5	≥ 25
eclipse	16	8	77	50	55	44	**27**
hsqldb	403	102	28	28	23	23	**23**
xalan	9	9	73	38	70	34	**19**
pseudojbb	37	9	14	14	14	14	**11**

Table 2: Thread counts and race counts.

programs and races. We evaluate PACER's accuracy and overhead at various sampling rates and compare its accuracy to LITERACE [22]. We experimentally confirm our theoretical claims: PACER accurately reports races in proportion to the sampling rate and its overhead is proportional to the sampling rate. Sampling rates of 1 and 3% incur low enough overhead (52% and 86%, respectively) to consider using in deployment.

5.1 Methodology

Platform. We execute all experiments on a Core 2 Quad 2.4 GHz system with 2 GB of main memory running Linux 2.6.20.3. Each of two cores has two processors, a 64-byte L1 and L2 cache line size, and an 8-way 32-KB L1 data/instruction cache; and each pair of cores shares a 4-MB 16-way L2 on-chip cache.

Benchmarks. We use the multithreaded DaCapo benchmarks [4] (eclipse, hsqldb, and xalan; version 2006-10-MR1) and a fixed-workload version of SPECjbb2000 called pseudojbb [29]. The DaCapo benchmark lusearch is multithreaded, but Jikes RVM 3.1.0 does not run it correctly in our environment, with or without PACER.

Threads. Table 2 shows the number of threads and detected races in each benchmark. *Total* is the total number of threads started. *Max live* is the maximum number of *live* threads at any time. Compared to the LITERACE and FASTTRACK experiments, our benchmarks have many more threads and races. Our prototype implementation does not reuse thread identifiers, so vector clock sizes are proportional to *Total*. A production implementation could use *accordion clocks* to reuse thread identifiers soundly [9].

Races and trials. Dynamically detecting races is challenging because some races occur infrequently. Another challenge is that the *observer effect* may introduce *heisenbugs* [12]; changing thread timing may increase or decrease the likelihood of a race. Sampling decreases the probability of observing a race. Even when a race occurs, ideal sampling detects the race with probability r, the sampling rate. We thus need many trials to evaluate accuracy.

Table 2 characterizes races in our programs. Columns four and five ($\forall r$) report the races observed from either 50 fully accurate executions ($r = 100\%$) or in a sampled execution (more than 1,000 additional trials). The table reports statically *distinct* races, i.e., it reports each pair of program references once even if the race occurs multiple times in a single execution. Column four reports races that occurred in at least one trial, and column five reports races that occurred in at least five trials. Columns six, seven, and eight ($r = 100$) reports races from the 50 trials executed at a 100% sampling rate. These columns report races that occur in 1, 5, and 25 trials, respectively. Comparing column four with five, and six with seven and eight, shows these programs have some rare races.

While PACER can find even rare races, the probability is the product of the sampling rate times the *occurrence* rate. For example, with a sampling rate $r = 1\%$ and an occurrence rate $o = 2\%$ (1 in 50), we would need 5000 trials to expect the race to be reported in one trial—and many more trials to report the race with high probability. Even a frequent race with $o = 100\%$ and $r = 1\%$ requires 100 trials to have break-even odds of being reported. To bound our experimentation time, we evaluate the accuracy of PACER on the races that appear in at least half of our 50 fully sampled executions (last column). About a third of races from the 50 fully accurate trials appear in 25 trials of xalan (19 races); about half appear in eclipse (27), most in pseudojbb (11), and all 23 races appear in all 50 trials of hsqldb (23). These are our *evaluation races*.

To report each race with reasonably high probability, we execute between 50 and 500 trials at each sampling rate, according to the following formula.

$$numTrials_r = min(max(\lceil \frac{1000\%}{r} \rceil, 50), 500)$$

For example, we perform 500 trials at a 1% sampling rate, 334 trials at a 3% sampling rate, and 50 trials at a 100% sampling rate.

5.2 Accuracy

This section evaluates PACER's race detection accuracy and shows that PACER accurately reports a proportion r of the evaluation races at various sampling rates r. It shows results that suggest that PACER can detect *each* evaluation race at the expected rate.

Figures 3 and 4 show PACER's detection rate versus sampling rate for each benchmark. Figure 3 counts the average number of *dynamic* evaluation races per run that PACER detects. A race's *detection rate* is the ratio of (1) average dynamic races per run at sampling rate r to (2) average dynamic races with $r = 100\%$. Each point is the *unweighted* average of all evaluation races' detection rates. The plot shows that PACER reports roughly a proportion r of dynamic races. PACER slightly underreports races in eclipse. On the other three benchmarks, PACER reports races at a somewhat better rate than the sampling rate. Factors such as the observer effect, sampling approach, and statistical error may prevent PACER from meeting its guarantee exactly.

Figure 4 shows the detection rate for *distinct* races. If a static race occurs multiple times in one trial, this plot counts it only once. The detection rate is somewhat higher in this case because PACER's chances of detecting a race improve if the race happens multiple times in a run. Developers are likely to be interested in the distinct detection rate: they care about which accesses race, not necessarily about how many times they race in a single run.

Per-race detection. The fours graphs in Figure 5 plot the detection rate for each *distinct* evaluation race as a function of r for each program. The x-axis sorts the races by detection rate, and each line is a sampling rate r. We sort the races independently for each sampling rate. The figures show how well PACER meets its guarantee of detecting each race with a probability equal to the sampling rate. PACER only misses one race in eclipse at a 1% sampling rate. Because of statistical error and heisenbugs, we cannot expect perfect results. Nonetheless, the results are compelling: PACER detects all but one race at least once at every sampling rate. On average, the detection rates correspond well with the specified sampling rates.

5.3 Comparison to LITERACE

For comparison, we implemented an online version of LITERACE [22]. It lowers overhead using a sampling heuristic that hypothesizes that most races occur in cold code [7]. LITERACE adaptively samples code in order to observe code at a rate inversely proportional to its frequency. It uses per-thread sampling rates and bursty sampling [14].

Our LITERACE implementation adaptively samples, lowering the sampling rate for each method-thread pair from 100% to 0.1%. Whereas the original LITERACE is deterministic, our implementation adds randomness when resetting the sampling counter, to increase the chances of catching more races across multiple trials. We initially used a sampling burst length of 10, but for all benchmarks

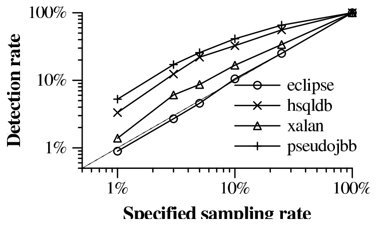

Figure 3: PACER's accuracy on *dynamic* races.

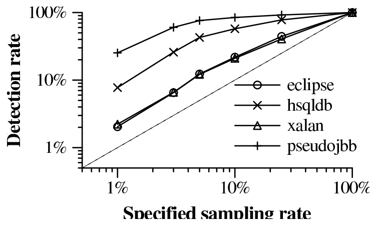

Figure 4: PACER's accuracy on *distinct* races.

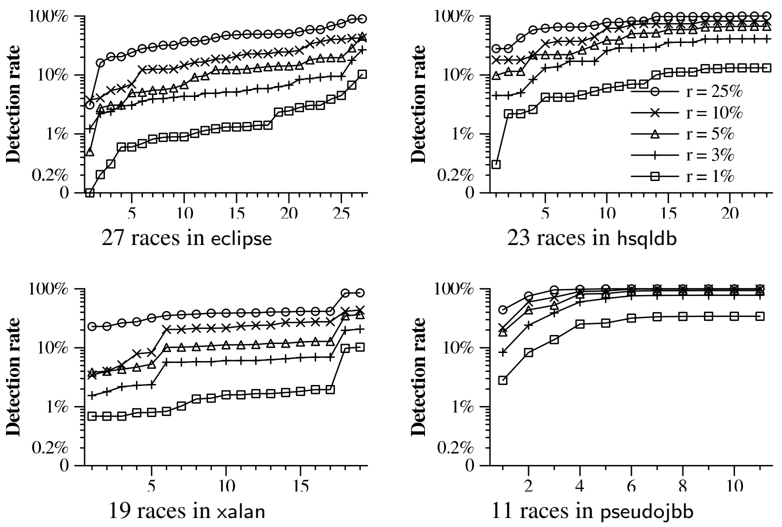

Figure 5: PACER's per-*distinct* race detection rate varying r.

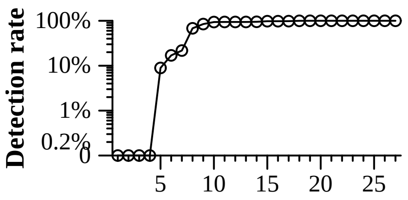

Figure 6: LITERACE's per-*distinct* race detection rate for eclipse.

except hsqldb, these configurations yielded effective sampling rates less than 1%, so we switched to burst lengths of 1,000. For hsqldb, xalan, and pseudojbb, LITERACE's heuristic is effective: it finds the evaluation races more frequently than the effective sampling rate. For eclipse (which achieves a 1.1% effective sampling rate with a burst length of 1,000), LITERACE misses some races consistently. Figure 6 shows how well LITERACE detects each evaluation race in eclipse, across 500 trials. LITERACE finds some races in many runs, but it never reports four of the evaluation races. For races between two hot accesses, we can surmise that this detection rate is approximately $0.1\%^2 = 0.0001\%$ (since 0.1% is the minimum sampling rate). These results indicate that races do not always follow the cold region hypothesis and that PACER's statistical guarantees provide accuracy improvements over the prior work.

Figure 10 shows that the space overhead of LITERACE even with an effective sampling rate of 1%, is almost as high as with 100% sampling. This result is not surprising because LITERACE samples code rather than data and does not discard metadata, so it ends up sampling most live memory. Section 5.4 presents the other data in Figure 10.

Our implementation of LITERACE's sampling mechanism is too inefficient to report fair time overheads. An online version of LITERACE will incur $O(n)$-time overheads at all synchronization operations, so it will not scale to many threads (Section 2.3).

5.4 Performance

Figure 7 presents the overheads of PACER with sampling rates of 0%, 1%, and 3%. Each sub-bar is the median of 10 trials. It breaks down the overheads into *OM + sync ops, r = 0%*, which is the cost of adding object metadata (e.g., two header words for every object) plus the cost of instrumentation at synchronization operations. Since it never samples, all vector clock operations use fast joins and shallow copies. This configuration adds about 15% overhead (we find that only about 1% comes from object metadata). *Pacer, r = 0%*, adds instrumentation at reads and writes but never executes it. Its total overhead is 33% on average. *Pacer, r = 1%*, samples with $r = 1\%$, adding 19% for a total of 52%. The last configuration, *Pacer, r = 3%*, adds 34% for a total of 86%.

Performance scalability. Larger sampling rates increase overhead roughly linearly. Figure 8 graphs slowdown vs. sampling rate for $r = 0$–100%; Figure 9 zooms in, showing $r = 0$–10%. The figures normalize to program execution time with unmodified Jikes RVM. The results for 0 and 1% sampling rates correspond to Figure 7. The graphs show that PACER achieves overheads that scale roughly linearly with the sampling rate.

At a 100% sampling rate, our implementation slows programs by 12x on average, compared with 8x in the FASTTRACK paper [11]. Our implementation performs worse because it uses hash tables instead of direct lookup (both for per-field metadata and read maps) and it inlines the non-sampling case, which decreases PACER's overhead in non-sampling periods but increases overhead in sampling periods.

Avoiding expensive operations. Table 3 shows statistics for $r = 3\%$ averaged over 10 trials on PACER's reduction of linear- to constant-time operations. The top half of the table shows the number of slow and fast joins and copies, during sampling and non-sampling periods. Note that a few deep copies occur in sampling periods because our implementation always performs deep copies for thread forks, since they are rare and it simplifies the implementation somewhat. Nearly all vector clock operations in non-sampling periods are *fast* (fast joins or shallow copies), i.e., they can be performed in $O(1)$ time.

The bottom half of Table 3 presents read and write operations that occur in sampling and non-sampling periods. Note that many more reads and writes occur in non-sampling periods, which is expected at a 3% sampling rate. In a non-sampling period, read and write instrumentation almost always takes the *fast path*: it does nothing if the field has no metadata. The number of slow path operations when PACER is not sampling corresponds well with the number of sampling slow-path operations. Because PACER checks for races with the last write and/or read to a variable in a sampling period, it performs some slow-path work in non-sampling periods.

Space overhead. PACER reduces space overhead when it discards read and write metadata or shares synchronization metadata during non-sampling periods. Figure 10 shows the amount of live (reach-

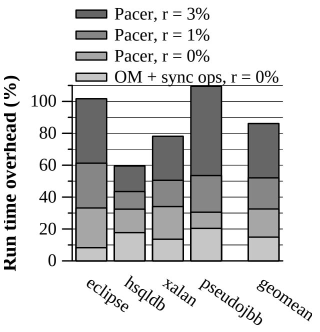

Figure 7: PACER overhead breakdown for $r = 0$–3%.

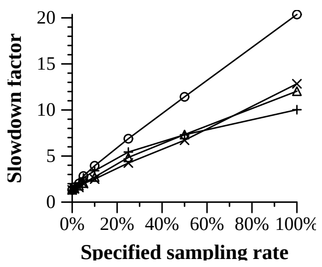

Figure 8: Performance vs. sampling rate for $r = 0$–100%.

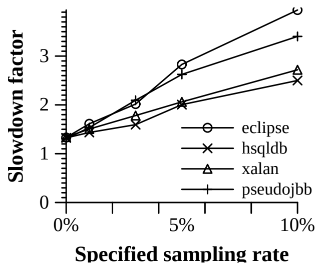

Figure 9: Performance vs. sampling rate for $r = 0$–10%.

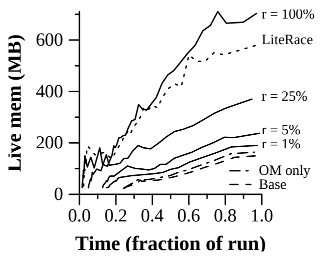

Figure 10: Total space over normalized time for eclipse.

	VC joins			
	Sampling period		Non-sampling period	
Program	Slow	Fast	Slow	Fast
eclipse	3,456K	656K	2K	149,376K
hsqldb	120K	288K	61K	14,636K
xalan	4,924K	3,255K	36K	275,724K
pseudojbb	2,932K	1,140K	3K	131,423K
	VC copies			
	Deep	Shallow	Deep	Shallow
eclipse	4,053K	–	<1K	147,458K
hsqldb	241K	–	<1K	7,938K
xalan	8,179K	–	<1K	275,760K
pseudojbb	4,072K	–	<1K	131,427K

	Reads		
	Sampling period	Non-sampling period	
	Slow path	Slow path	Fast path
eclipse	273,611K	14,170K	8,792,182K
hsqldb	13,108K	1,697K	431,167K
xalan	190,502K	118,682K	6,163,120K
pseudojbb	33,311K	51,254K	835,085K
	Writes		
eclipse	66,704K	52K	2,165,973K
hsqldb	1,696K	19K	50,217K
xalan	30,350K	442K	992,098K
pseudojbb	12,197K	1,064K	330,902K

Table 3: Counts of vector clock joins and copies, and read and write operations for PACER at a sampling rate of 3%. $O(n)$-time vector clock operations are almost entirely confined to sampling periods, and slow-path reads and writes in non-sampling periods correspond well with reads and writes in sampling periods.

able) memory for eclipse after each full-heap collection with various PACER configurations. The measurement includes application, VM, and PACER memory. We use a single trial of each configuration because averaging over multiple trials might smooth spikes caused by PACER's sampling periods. Because PACER takes longer to run with higher sampling rates, we normalize execution times over total run length. *Base* shows the memory used by eclipse running on unmodified Jikes RVM. Note that memory usage increases somewhat over time in this program. *OM only* adds two words per object and a few percent all-the-time overhead. The other configurations are PACER at various sampling rates. The graph shows that PACER's space overhead scales well with the sampling rate.

In summary, PACER is accurate and low overhead because it finds races, performs work, and uses memory, all in proportion to the sampling rate.

6. Related Work

Section 2 compared PACER to the most closely related work, FASTTRACK [11] and LITERACE [22]. This section compares PACER to other prior work on race detection.

6.1 Language Design and Static Analysis for Race Detection

Safety in types. An alternative to detecting races is to use a language that cannot have them. Boyapati et al. extend the typing of an existing programming language with ownership types, so that well-typed programs are guaranteed to be race-free [6]. Abadi et al. use type inference and type annotations to detect races soundly [1].

Static analysis. Researchers have developed advanced techniques for statically detecting data races [24; 27; 31]. These approaches scale to millions of lines of code, are typically sound except for a few exceptions, and try to limit false positives as much as possible. However, static analysis necessarily reports false positives because it abstracts control and data flow in order to scale. In contrast, model checking is precise but does not scale well to large programs [13]. Races, whether true or false, are time consuming to fix (or not fix), so even a few false positives may frustrate developers.

Choi et al. combine static and dynamic analysis to lower the overhead of dynamic race detection, which can identify read and write operations that cannot be involved in races [8]. Static approaches are typically unsound with respect to dynamic language features such as dynamic class loading and reflection. Our implementation uses simple, mostly intraprocedural escape analysis to identify some definitely thread-local objects (Section 4).

6.2 Dynamic Race Detection

Dynamic race detectors are typically based on the imprecise *lockset* algorithm or precise *vector clock* algorithm.

Lockset algorithm. The lockset algorithm checks a locking discipline based on each access to a shared variable holding some common lock [8; 28]. Because it enforces a particular locking discipline, lockset is imprecise: it reports false positives due to other synchronization idioms such as fork-join, wait-notify, and custom synchronization with volatile variables. Furthermore, recent advances in precise, *vector clock*-based race detection (notably FASTTRACK's order-of-magnitude improvement) mean that lockset and vector clocks offer about the same performance [11].

Vector clocks. Prior techniques, including FASTTRACK and LITERACE, use vector clocks to achieve precise race detection [11; 22]. They both decrease analysis overhead at reads and writes, but overheads at sychronization operations still take $O(n)$ time, so these approaches will not scale to many threads. In contrast, PACER can scale to many threads by adjusting the sampling rate since overheads are proportional to the sampling rate.

Hybrid techniques. Hybrid techniques combine lockset and vector clocks to obtain the performance of the former and the accuracy of the latter [10; 30; 32]. *Goldilocks* is sound and precise and reports overheads low enough to deploy [10]. However, as Flanagan and Freund note [11], Goldilocks is *compiled* into a JVM that only *interprets* code, so its overhead would likely be much higher in a high-performance JVM. Pozniansky and Shuster introduce improved versions of both vector clock and lockset race detection, and present *MultiRace*, which is a hybrid of these two improved detection approaches [26]. FASTTRACK further improves on MultiRace's vector clock-based detector (called Djit^{+}) [11].

6.3 Sampling

Object-centric sampling tracks only a subset of objects, chosen at allocation time [3; 15]. Modifying LITERACE to use object-centric sampling would reduce its space overhead to be proportional to n, but it would still need $O(n)$-time analysis at synchronization operations. PACER's goals are similar in spirit to those of *QVM*, which performs as much analysis as possible while staying within a user-specified overhead budget [3]. PACER is well suited to finding races in widely deployed software as part of a distributed sampling framework [17; 18].

7. Conclusion

Data races indicate serious concurrency errors that are easy to introduce but difficult to reproduce, understand, and fix. Not even thorough testing finds all races, so deployable race detection is necessary to achieve highly robust software. Prior approaches are too heavyweight, or they are only effective at finding a subset of races. This paper presents data race detection that provides a detection rate for *each* race that is equal to the sampling rate, and adds time and space overheads proportional to the sampling rate. The approach's adjustable performance and accuracy guarantees make it suitable for all-the-time use in a variety of deployed environments.

Acknowledgments

We thank Sam Guyer, Bert Maher, Satish Narayanasamy, and Ben Wiedermann for helpful discussions; Bert Maher for helpful feedback on the text; Steve Blackburn and Robin Garner for developing an improved pseudojbb; and Steve Freund for explaining some details of FastTrack.

A. Formal semantics

A program consists of a set of concurrently executing threads $t \in \mathit{Tid}$ that perform *actions* to manipulate a set of data variables $x \in \mathit{Var}$, a set of locks $m \in \mathit{Lock}$, and a set of volatile variables $vx \in \mathit{Vol}$. Threads, locks, and volatile variables are all called *synchronization objects*, and all other variables are *data variables*. An action is one of the following operations:

$rd(t, x)$: thread t reads data variable x.

$wr(t, x)$: thread t writes data variable x.

$acq(t, m)$: thread t acquires lock m.

$rel(t, m)$: thread t releases lock m.

$fork(t, u)$: thread t forks a new thread, u.

$join(t, u)$: thread t blocks until thread u terminates.

$vol\_rd(t, vx)$: thread t reads volatile variable vx.

$vol\_wr(t, vx)$: thread t writes volatile variable vx.

$sbegin()$: the analysis enters a sampling period.

$send()$: the analysis leaves a sampling period.

The acq, rel, $fork$, $join$, vol_rd, and vol_wr actions are *synchronization actions*. The $sbegin$ and $send$ actions are convenient notation for the start and end of a PACER sampling period. These actions are not initiated by any particular thread and they do not affect the happens-before relationship between threads, but they do modify the analysis state.

A *trace* α captures the sequence of actions performed by the various threads in a multi-threaded program. Given a trace α and an action b, the term $\alpha.b$ denotes the trace that results after extending α by b. An action a is related to an action b in a trace α by *program order* if a occurs before b in α, and the same thread performs both a and b. Two actions a and b in α are related by *synchronization order* if a and b are both synchronization actions, and a occurs before b. An action a is related to an action b in α by *synchronizes-with order* if they are also related by synchronization order, and any of the following hold:

- a is a release of a lock m and b is an acquire of m by any thread.
- a is a write to a volatile variable vx and b is a read of vx by any thread.
- a is a fork of a new thread u by a thread $t, t \neq u$, and u is the thread that performs b.
- a thread u performs a, and b is a join that blocks a thread t, $t \neq u$, until u terminates.

The happens-before relation for actions a and b in a trace α, $a \xrightarrow{\text{HB}}_{\alpha} b$, is the transitive closure of program order and synchronizes-with order [21]. If a is related to b by the happens-before relation then a *happens before* b and b *happens after* a, otherwise a and b are *concurrent*. Two actions a and b *conflict* if they both read or write the same data variable, and at least one of the two actions writes that variable. A trace α contains a *data race* if it contains conflicting, concurrent read/write actions.

We restrict our attention to traces that are feasible and that obey traditional synchronization operation semantics. In particular, citing many of the examples provided by Flanagan et al. [11]:

- A thread never acquires a lock that has been acquired, but not released by another thread.
- A thread never releases a lock m unless it has previously acquired m without releasing it.
- A thread u never performs any actions that precede an action $fork(t, u)$.
- A thread u never performs any actions subsequent to an action $join(t, u)$.
- Thread u performs at least one action after $fork(t, u)$.
- A volatile read action $vol\_rd(t, vx)$ always returns the value of the most recent volatile write action $vol\_wr(t, vx)$ that precedes the read in synchronization order.

A.1 Vector clocks and epochs

A vector clock $VC : \mathit{Tid} \rightarrow \mathit{Nat}$ maps thread identifiers to natural numbers that represent logical clock values. Given a vector clock $C \in VC$ the term $C(t) \in \mathit{Nat}$ refers to the clock value to which C maps thread t. Vector clocks are partially ordered in a pointwise manner; a vector clock C_1 is pointwise less-than a vector clock C_2 ($C_1 \sqsubseteq C_2$) if and only if each element in C_1 is less than or equal to the corresponding element in C_2:

$$C_1 \sqsubseteq C_2 \text{ iff } \forall t. C_1(t) \leq C_2(t)$$

The minimal element for a vector clock $\bot_c \in VC$ is the vector clock that maps every thread to 0, $\bot_c = \lambda t.0$. We define three operations on vector clocks, copy, increment, and join ($\sqcup$):

$$copy(C) = \lambda t.C(t) \tag{1}$$

$$inc_u(C_u) = \lambda t.\text{ if } t = u \text{ then } C(t) + 1 \text{ else } C(t) \tag{2}$$

$$C_1 \sqcup C_2 = \lambda t.max(C_1(t), C_2(t)) \tag{3}$$

The copy operation, $copy(C)$, returns a vector clock where each component is equal to C's value for that component. The increment operation, $inc_t(C_t)$, returns a vector that is identical to C_t except that t's component has increased by one. The increment operation is the mechanism by which logical time passes. The join operation, $C_1 \sqcup C_2$, returns the pointwise maximum of C_1 and C_2.

Like FASTTRACK, PACER uses an *epoch* to concisely represent the vector clock for a data variable x when accesses to x are totally ordered. An epoch $c@t$ is a pair consisting of a thread identifier, t, and the clock associated with that thread identifier, c. The term $\bot_e$ denotes the minimal epoch 0@0. This minimal epoch is not unique; any epoch $0@t$ is a minimal epoch. The relation $c@t \preceq C$ holds for an epoch $c@t$ and a vector clock C if and only if c is less than or equal to the t component in C:

$$c@t \preceq C \text{ iff } c \leq C(t) \tag{4}$$

Unlike vector clock comparisons, which require time proportional to the number of threads, evaluating the relation in Equation 4 requires constant time. Both FASTTRACK and PACER use epochs to store read and write metadata when accesses to data variables are totally ordered. Note that while the $\sqsubseteq$ and $\preceq$ relations imply happens-before in FASTTRACK, in PACER they imply happens-before only during sampling periods.

Given a vector clock C_t associated with a thread t, we use the abbreviation $E(t) \in Epoch$ to denote the *current epoch* of thread t's vector clock:

$$E(t) = C_t(t)@t$$

Both PACER and FASTTRACK include either a vector clock or an epoch in the metadata for each variable that they track. To enable efficient sampling, however, PACER maintains additional metadata called *version vectors* to identify redundant vector clock values.

A.2 Version vectors and version epochs

A version vector $VersionVec : Tid \rightarrow Nat$ maps each thread to a natural number that represents a *version* of that thread's vector clock. A version is a unique identifier for a snapshot of a thread's vector clock at a point in logical time. Given a version vector $V \in VersionVec$, we use $V(t)$ to denote the version to which V maps thread t. A minimal version vector $\bot_v \in VersionVec$ maps all threads to 0, $\bot_v = \lambda t.0$. We define an increment operation on a version vector V and a thread $t \in Tid$:

$$inc_t(V) = \lambda u.\text{ if } u = t \text{ then } V(u) + 1 \text{ else } V(u) \tag{5}$$

Much like the vector clock increment operation, the version vector increment returns a version vector that is identical to V except that the t component has increased by one. A thread increments its version vector whenever its vector clock changes.

PACER associates version information with each thread and each synchronization object. Rather than a version vector, PACER associates a more concise *version epoch* with locks and volatile variables. This concise representation is suitable because operations that acquire and release the same lock are always totally ordered, and most conflicting accesses to a given volatile variable are also totally ordered. A version epoch is a pair, $v@t$, where $t \in Tid$ is a thread identifier and $v \in Nat$ is a version number for thread t's vector clock. The relation $v@t \preceq V$ holds for a version epoch $v@t$ and a version vector V if and only if v is less than or equal to the t component in V:

$$v@t \preceq V \text{ iff } v \leq V(t) \tag{6}$$

The term $\bot_{ve} \in VersionEpoch$ denotes a minimal version epoch 0@0 such that $\bot_{ve} \preceq V$ is always true. This minimal version epoch is not unique; any version epoch $0@t$ is a minimal version epoch. The term $\top_{ve} \in VersionEpoch$ denotes a unique maximal version epoch such that $\top_{ve} \preceq V$ is never true. PACER uses a null version epoch to represent $\top_{ve}$.

A.3 Synchronization metadata

PACER associates both a vector clock and version information with each synchronization object. We will express the metadata for a synchronization object o as a tuple $\mathcal{S}_o \in Meta$ that consists of the following components:

- $\mathcal{S}_o.vc : VC$
- $\mathcal{S}_o.ver : (VersionVec \cup VersionEpoch)$

$\mathcal{S}_o.vc$ is the vector clock for o, and $\mathcal{S}_o.ver$ is the *version map* for o. The version map is either a version vector or a version epoch. When $o \in Lock$, $\mathcal{S}_o.ver$ is always a version epoch because all actions that acquire or release the same lock are totally ordered. A release of lock m happens-before an acquire of m, and an acquire of m by a thread t always happens-before the subsequent release of m by program order, because no other thread can acquire the lock until t releases it. Although accesses to volatile variables are not always totally ordered, we find that in practice conflicting concurrent accesses to volatile variables usually are. Thus, when $o \in Vol$, $\mathcal{S}_o.ver$ is a version epoch unless the most recent write to o was not totally ordered with respect to all prior writes to o, in which case $\mathcal{S}_o.ver$ is $\top_{ve}$. When $o \in Tid$, $\mathcal{S}_o.ver$ is a version vector.

Given synchronization metadata $\mathcal{S}_o \in Meta$, we define an abbreviation $Ver(o)$ called the *current version* of o's vector clock. The current version is a version epoch $v@t$ that indicates that o's vector clock is equal to thread t's vector clock when its current version was v. If $o \in Lock$ or $o \in Vol$, then the current version is equal to o's version metadata, which is always a version epoch:

$$Ver(o) = \mathcal{S}_o.ver \mid o \in (Lock \cup Vol)$$

If $o \in Tid$, however, then $\mathcal{S}_o.ver$ is a version vector, not a version epoch. A thread's current version is its own identifier and the corresponding slot in its version vector, which it updates every time its vector clock changes:

$$Ver(o) = \mathcal{S}_o.ver(o)@o \mid o \in Tid$$

The remaining slots in a thread t's version vector indicate, for each other thread u, what is the most recent version of u's vector clock that has been joined with t's vector clock. That version, and any prior versions of u's vector clock, are all guaranteed to be pointwise less-than t's vector clock.

A.4 Analysis state

The *analysis state* for PACER $\sigma = (\mathcal{C}, \mathcal{L}, \mathcal{V}, R, W, s)$ consists of the following components:

- $\mathcal{C} : Tid \rightarrow Meta$
- $\mathcal{L} : Lock \rightarrow Meta$
- $\mathcal{V} : Vol \rightarrow Meta$
- $R : Var \rightarrow (Epoch \cup VC)$
- $W : Var \rightarrow Epoch$
- $s : boolean$

$\mathcal{C}$, $\mathcal{L}$, and $\mathcal{V}$ map threads, locks, and volatile variables, respectively, to their metadata. R is a read map that maps each data variable x to either an epoch indicating the clock of the last totally ordered read of x and the thread that performed that read, or a vector clock that contains a join over multiple concurrent reads of x. W maps each data variable x to an epoch that indicates the clock of the last write to x and the thread that performed that write. Finally, s is a boolean that is true if PACER is currently in a sampling period.

The initial analysis state for PACER, σ_0, contains the following component values:

$$\begin{aligned}\sigma_0 \;=\; & (\lambda t.(inc_t(\bot_c), inc_t(\bot_v)), \\ & \lambda m.(\bot_c, \bot_{ve}), \\ & \lambda vx.(\bot_c, \bot_{ve}), \\ & \lambda x.\bot_e, \\ & \lambda x.\bot_e, \\ & false)\end{aligned} \tag{7}$$

PACER increments both the vector clock and the version vector for a thread in the initial state, but initializes everything else to its minimal value.

An action a transitions PACER from one analysis state to another, $\sigma \Rightarrow^a \sigma'$. Table 4 shows how PACER updates the analysis state when a data variable read or write action occurs. Note that if a condition labeled "*Race-free*" is not satisfied, then there is no rule in Table 4 that transitions PACER to the next analysis state, and the analysis becomes *stuck*. Thus, the relation $\sigma \Rightarrow^a \sigma'$ indicates that PACER successfully transitions to a new state via action a without becoming stuck and reporting a data race. If action a causes the analysis to become stuck, then we write

$$\sigma \not\Rightarrow^a \ldots$$

Table 5 shows how the $sbegin$ and $send$ actions modify the analysis state at the start and end of a sampling period. Note that although the $sbegin$ action does modify each thread's vector clock, it does not add or remove any happens-before edges between threads. This action is effectively equivalent, from the perspective of the analysis, to each thread writing its own private volatile variable that no other threads access. Thus, the analysis does not lose any information about data races as a result of entering a sampling period.

Table 6 shows how PACER updates the analysis state when a synchronization action occurs. These updates are identical to those performed by FASTTRACK except that PACER performs modified join, copy, and increment operations. Table 7 shows how these modified operations use version information to avoid O(n) operations when possible. The observable effect of these operations, however, is identical to the corresponding vector clock operations with only one exception: PACER effectively halts the flow of logical time when a sampling period ends by no longer performing the vector clock increment operation (see Rule 2 in Table 7).

In Tables 4-7 we frequently abbreviate notation for simplicity and to conserve space. For example, $R'_x = E(t)$ is an abbreviation for $R' = R[x := E(t)]$, which indicates that R' is identical to R except that x maps to $E(t)$.

We do not address shallow and deep copies of vector clocks here because we believe that their correctness will be clear to readers. PACER always checks whether metadata is shared before modifying it, and if it is shared PACER creates a deep copy prior to making any changes. Whenever PACER creates a shallow copy, it marks the object shared. Once an object is marked shared it remains that way for the rest of its lifetime.

B. PACER correctness proofs

We will refer to the components of analysis states σ and σ' as follows:

$$\sigma = (\mathcal{C}, \mathcal{L}, \mathcal{V}, R, W, s)$$
$$\sigma' = (\mathcal{C}', \mathcal{L}', \mathcal{V}', R', W', s')$$

When necessary to avoid ambiguity, we will use the following conventions to differentiate analysis states: given an action a, the term σ_a denotes the state prior to performing a, and σ'_a denotes the state after performing a: $\sigma_a \Rightarrow^a \sigma'_a$. Using conventions similar to those used to prove FASTTRACK correct [11], we will refer to the components of states σ_a and σ'_a as follows:

$$\sigma_a = (\mathcal{C}^a, \mathcal{L}^a, \mathcal{V}^a, R^a, W^a, s^a)$$
$$\sigma'_a = (\mathcal{C}'^a, \mathcal{L}'^a, \mathcal{V}'^a, R'^a, W'^a, s'^a)$$

Similar conventions hold for the function $Ver(o)$:

- $Ver(o)$ is the current version of o in state σ.
- $Ver'(o)$ is the current version of o in state σ'.
- $Ver^a(o)$ is the current version of o in state σ_a.
- $Ver'^a(o)$ is the current version of o in state σ'_a.

The abbreviation IH stands for inductive hypothesis. Note that the operator that transitions the analysis from one state to another, $\sigma \Rightarrow^a \sigma'$, and the implication operator, $\Longrightarrow$, are similar to one another, but the implication operator is longer. Recall that the term $\alpha.a$ denotes the trace that results when trace α is extended by action a. We use the Greek math symbols α, β, and γ to denote arbitrary-length sequences of actions.

DEFINITION 1. (Well-formedness). *A state* $\sigma = (\mathcal{C}, \mathcal{L}, \mathcal{V}, R, W, s)$ *is* well-formed *if* $\forall t, u \in Tid, m \in Lock, x \in Var, vx \in Vol$:

1. $\mathcal{C}_u.vc(t) \le \mathcal{C}_t.vc(t)$
2. $\mathcal{L}_m.vc(t) \le \mathcal{C}_t.vc(t)$
3. $R_x(t) \le \mathcal{C}_t.vc(t)$
4. $W_x(t) \le \mathcal{C}_t.vc(t)$
5. $\mathcal{V}_{vx}.vc(t) \le \mathcal{C}_t.vc(t)$
6. $\mathcal{C}_u.ver(t) \le \mathcal{C}_t.ver(t)$
7. $\mathcal{L}_m.ver(t) \le \mathcal{C}_t.ver(t)$
8. $\mathcal{V}_{vx}.ver(t) \le \mathcal{C}_t.ver(t)$

LEMMA 1. σ_0 *is well-formed.*

Proof. σ_0 is well-formed by Equation 7. □

LEMMA 2. (Logical time increases monotonically). *If σ is well-formed and $\sigma \Rightarrow^a \sigma'$, then $\mathcal{C}_t.vc(t) \le \mathcal{C}'_t.vc(t)$.*

Proof. $\mathcal{C}_t.vc(t) \le \mathcal{C}'_t.vc(t)$ follows directly from the update rules in Tables 5 and 6. The value of $\mathcal{C}'_t.vc(t)$ is set only by the increment operation (Rule 1 in Table 5, Rules 2, 3, 4, and 6 in Table 6), and the join operation where one of the operands is itself (Rules 1, 3, 4, and 5 in Table 6). Thus, by Equations 3 and 2 (vector clock join and increment) and the update rules in Tables 5 and 6, $\mathcal{C}_t.vc(t) \le \mathcal{C}'_t.vc(t)$ □

LEMMA 3. (Version numbers increase montonically). *If σ is well-formed and $\sigma \Rightarrow^a \sigma'$, then $\mathcal{C}_t.ver(t) \le \mathcal{C}'_t.ver(t)$.*

Proof. $\mathcal{C}_t.ver(t) \le \mathcal{C}'_t.ver(t)$ by Equation 5 (version increment definition) and Rules 6 and 3 in Table 7, which are the only rules that set $\mathcal{C}'_t.ver(t)$. □

LEMMA 4. (Preservation of well-formedness). *If σ is well-formed and $\sigma \Rightarrow^a \sigma'$ then σ' is well-formed.*

Proof. Assume σ' is not well-formed. By Definition 1 (well-formedness), Lemmas 2 and 3 (monotonicity), and the update rules in Tables 4, 5, and 6, σ must not be well-formed. Thus, we have a contradiction. □

Data reads and writes: $\sigma \Rightarrow^a \sigma'$, $\sigma = (\mathcal{C}, \mathcal{L}, \mathcal{V}, R, W, s)$			
	State updates		
Conditions	**Sampling**	**Non-sampling**	
Read data variable: $rd(t,x)$, $\sigma' = (\mathcal{C}, \mathcal{L}, \mathcal{V}, R', W, s)$			
Same epoch $R_x \in Epoch$ $R_x = E(t)$	None	None	(1)
Exclusive $R_x \in Epoch$ $R_x \preceq \mathcal{C}_t.vc$ *Race-free* $W_x \preceq \mathcal{C}_t.vc$	$R'_x = E(t)$	$R'_x = \bot_e$	(2)
Shared $R_x \in VC$ *Race-free* $W_x \preceq \mathcal{C}_t.vc$	$R'_x(t) = \mathcal{C}_t.vc(t)$	$R'_x(t) = 0$	(3)
Share $R_x \in Epoch$ $R_x \npreceq \mathcal{C}_t.vc$ *Race-free* $W_x \preceq \mathcal{C}_t.vc$	$R'_x = \bot_c$ $R'_x(t) = R_t(t)$ Let $R_x = c@u$ $R'_x(u) = c$	None	(4)
Write data variable: $wr(t,x)$, $\sigma' = (\mathcal{C}, \mathcal{L}, \mathcal{V}, R', W', s)$			
Same epoch $W_x = E(t)$	None	None	(5)
Exclusive $R_x \in Epoch$ *Race-free* $R_x \preceq \mathcal{C}_t.vc$ $W_x \preceq \mathcal{C}_t.vc$	$W'_x = E(t)$ $R'_x = \bot_e$ [1]	$R'_x = \bot_e$ $W'_x = \bot_e$	(6)
Shared $R_x \in VC$ *Race-free* $R_x \sqsubseteq \mathcal{C}_t.vc$ $W_x \preceq \mathcal{C}_t.vc$	$W'_x = E(t)$ $R'_x = \bot_e$	$R'_x = \bot_e$ $W'_x = \bot_e$	(7)

Table 4: PACER's modifications to read and write metadata. Column 1 shows the checks PACER performs to determine which updates are necessary and to check for race freedom, when needed. Column 2 shows PACER's updates within a sampling period, which match FASTTRACK's read and write metadata updates. Column 3 shows PACER's updates in non-sampling periods. Gray boxes indicate $O(n)$-time computations.

[1] The original FASTTRACK algorithm does not perform this update.

Sampling periods: $\sigma \Rightarrow^a \sigma'$, $\sigma = (\mathcal{C}, \mathcal{L}, \mathcal{V}, R, W, s)$			
Action a	**Next state** σ'	**State updates**	
$sbegin()$	$(\mathcal{C}', \mathcal{L}, \mathcal{V}, R, W, s')$	$\mathcal{C}' = \lambda t.inc_t(\mathcal{C}_t)$ $s' = true$	(1)
$send()$	$(\mathcal{C}, \mathcal{L}, \mathcal{V}, R, W, s')$	$s' = false$	(2)

Table 5: Analysis state updates that PACER performs at the start and end of a sampling period.

Synchronization actions: $\sigma \Rightarrow^a \sigma'$, $\sigma = (\mathcal{C}, \mathcal{L}, \mathcal{V}, R, W, s)$			
Action a	**Next state** σ'	**State updates**	
$acq(t,m)$	$(\mathcal{C}', \mathcal{L}, \mathcal{V}, R, W, s)$	$\mathcal{C}'_t = \mathcal{C}_t \sqcup \mathcal{L}_m$	(1)
$rel(t,m)$	$(\mathcal{C}', \mathcal{L}', \mathcal{V}, R, W, s)$	$\mathcal{L}'_m = copy(\mathcal{C}_t)$ $\mathcal{C}'_t = inc_t(\mathcal{C}_t, s)$	(2)
$fork(t,u)$	$(\mathcal{C}', \mathcal{L}, \mathcal{V}, R, W, s)$	$\mathcal{C}'_u = \mathcal{C}_u \sqcup \mathcal{C}_t$ $\mathcal{C}'_t = inc_t(\mathcal{C}_t, s)$	(3)
$join(t,u)$	$(\mathcal{C}', \mathcal{L}, \mathcal{V}, R, W, s)$	$\mathcal{C}'_t = \mathcal{C}_t \sqcup \mathcal{C}_u$ $\mathcal{C}'_u = inc_u(\mathcal{C}_u, s)$	(4)
$vol\_rd(t,vx)$	$(\mathcal{C}', \mathcal{L}, \mathcal{V}, R, W, s)$	$\mathcal{C}'_t = \mathcal{C}_t \sqcup \mathcal{V}_{vx}$	(5)
$vol\_wr(t,vx)$	$(\mathcal{C}', \mathcal{L}, \mathcal{V}', R, W, s)$	$\mathcal{V}'_{vx} = \mathcal{V}_{vx} \sqcup \mathcal{C}_t$ $\mathcal{C}'_t = inc_t(\mathcal{C}_t, s)$	(6)

Table 6: Analysis state updates that PACER performs in response to synchronization actions. See Table 7 for definitions of the copy, increment, and join operations.

Metadata join, copy, and increment operations		
Conditions	**State updates**	
Metadata copy: $\mathcal{S}'_o = copy(\mathcal{C}_t)$, $o \in (Lock \cup Vol)$		
None	$\mathcal{S}'_o.vc = \mathcal{C}_t.vc$ $\mathcal{S}'_o.ver = Ver(t)$	(1)
Metadata increment: $\mathcal{C}'_t = inc_t(\mathcal{C}_t, s)$, $t \in Tid$		
Non-sampling $s = false$	None	(2)
Sampling $s = true$	$\mathcal{C}'_t.vc = inc_t(\mathcal{C}_t.vc)$ $\mathcal{C}'_t.ver = inc_t(\mathcal{C}_t.ver)$	(3)
Metadata join for threads: $\mathcal{C}'_t = \mathcal{C}_t \sqcup \mathcal{S}_o$, $o \in (Tid \cup Lock \cup Vol)$		
Same version epoch $Ver(o) \preceq \mathcal{C}_t.ver$	None	(4)
Happens-before $Ver(o) \npreceq \mathcal{C}_t.ver$ $\mathcal{S}_o.vc \sqsubseteq \mathcal{C}_t.vc$	Let $v@u = Ver(o)$ $\mathcal{C}'_t.ver(u) = v$	(5)
Concurrent $Ver(o) \npreceq \mathcal{C}_t.ver$ $\mathcal{S}_o.vc \not\sqsubseteq \mathcal{C}_t.vc$	$\mathcal{C}'_t.vc = \mathcal{C}_t.vc \sqcup \mathcal{S}_o.vc$ $\mathcal{C}'_t.ver = inc_t(\mathcal{C}_t.ver,)$ Let $v@u = Ver(o)$ $\mathcal{C}'_t.ver(u) = v$ {if $Ver(o) \neq \top_{ve}$}	(6)
Metadata join for volatiles: $\mathcal{V}'_{vx} = \mathcal{V}_{vx} \sqcup \mathcal{C}_t$, $vx \in Vol$		
Same version epoch $Ver(vx) \preceq \mathcal{C}_t.ver$	$\mathcal{V}'_{vx} = copy(\mathcal{C}_t)$	(7)
Happens-before $Ver(vx) \npreceq \mathcal{C}_t.ver$ $\mathcal{V}_{vx}.vc \sqsubseteq \mathcal{C}_t.vc$	$\mathcal{V}'_{vx} = copy(\mathcal{C}_t)$	(8)
Concurrent $Ver(vx) \npreceq \mathcal{C}_t.ver$ $\mathcal{V}_{vx}.vc \not\sqsubseteq \mathcal{C}_t.vc$	$\mathcal{V}'_{vx}.vc = \mathcal{C}_t.vc \sqcup \mathcal{V}_{vx}.vc$ $\mathcal{V}'_{vx}.ver = \top_{ve}$	(9)

Table 7: Copy, increment, and join operations for PACER metadata.

DEFINITION 2. (Strict well-formedness). *A state $\sigma = (\mathcal{C}, \mathcal{L}, \mathcal{V}, R, W, s)$ is* strictly well-formed *if $\forall t, u \in Tid, m \in Lock, vx \in Vol$:*

1. *σ is well-formed.*
2. *if $t \neq u$ then $\mathcal{C}_u.vc(t) < \mathcal{C}_t.vc(t)$.*
3. *$\mathcal{L}_m.vc(t) < \mathcal{C}_t.vc(t)$.*
4. *$\mathcal{V}_{vx}.vc(t) < \mathcal{C}_t.vc(t)$.*

Note that "strict well-formedness" is equivalent to "well-formedness" in FASTTRACK. PACER's definition for "well-formedness" (see Definition 1) is slightly more relaxed because logical time stops during non-sampling periods.

LEMMA 5. (Strict well-formedness at sampling period entry). *If σ is well-formed, $\sigma \Rightarrow^a \sigma'$, and $a = sbegin()$, then σ' is strictly well-formed.*

Proof. σ' must be strictly well-formed by Rule 1 in Table 5 ($sbegin$) and criteria 2, 3, and 6 of Definition 1 (well-formedness). Intuitively, the $sbegin$ action establishes strict well-formedness after a non-sampling period by incrementing each thread's vector clock. □

LEMMA 6. (Preservation of strict well-formedness within a sampling period). *If σ is strictly well-formed, $\sigma \Rightarrow^a \sigma'$, and $s = true$ then σ' is strictly well-formed.*

Proof. Assume σ' is not strictly well-formed. By Lemma 4 (preservation of well-formedness), σ' is well-formed. Thus, σ' must violate criteria 2, 3, or 4 of Definition 2. By Lemma 2 (monotonicity) a must increase $\mathcal{C}'_u.vc(t)$, $\mathcal{L}'_m.vc(t)$, or $\mathcal{V}'_{vx}.vc(t)$ to be greater than $\mathcal{C}'_t.vc(t)$. By the update rules in Table 6, σ must not be strictly well-formed. Thus, we have a contradiction.

Intuitively, notice that whenever a thread's vector clock is joined into another vector clock, which may sacrifice strict well-formedness, the thread subsequently increments its vector clock (provided that the analysis is in a sampling period). This vector clock increment restores strict well-formedness. Each synchronization action that can serve as the source of a happens-before edge uses this pattern (see Rules 2, 3, 4, and 6 in Table 6). □

LEMMA 7. (Versions imply vector clock ordering). *Let $t \in Tid$ and $o \in (Tid \cup Lock \cup Vol)$. If $\sigma_0 \Rightarrow^\alpha \sigma$ then*

$$Ver(o) \preceq \mathcal{C}_t.ver \implies \mathcal{S}_o.vc \sqsubseteq \mathcal{C}_t.vc \qquad \textit{To prove} \qquad (8)$$

Proof. If $t = o$ then Equation 8 is trivially true. Thus, assume $t \neq o$, and the proof proceeds by induction on the length of α. Assume the length of α is zero, and show that Equation 8 holds in the initial state.

$$\mathcal{S}_o.vc \sqsubseteq \mathcal{C}_t.vc \qquad \text{if } o \in (Lock \cup Vol) \qquad \text{Equation 7} \qquad (9)$$
$$Ver(o) \not\preceq \mathcal{C}_t.ver \qquad \text{if } o \in Tid \qquad '' \qquad (10)$$

Equations 9 and 10 follow from the initial analysis state in Equation 7, which initializes the vector clock for a lock or volatile variable to $\bot_v$, and the vector clock for a thread t to $inc_t(\bot_v)$. In the initial state Equation 8 holds because either the consequent is guaranteed to be true, or the antecedent is guaranteed to be false by Equations 9 and 10.

Suppose that $\alpha = \beta.a$. If $\sigma_0 \Rightarrow^\beta \sigma \Rightarrow^a \sigma'$, and

$$Ver(o) \preceq \mathcal{C}_t.ver \implies \mathcal{S}_o.vc \sqsubseteq \mathcal{C}_t.vc \qquad \text{IH} \qquad (11)$$

show that

$$Ver'(o) \preceq \mathcal{C}'_t.ver \implies \mathcal{S}'_o.vc \sqsubseteq \mathcal{C}'_t.vc \qquad \text{To prove} \qquad (12)$$

The vector clock and version vector for an object can be modified by a copy, increment or join operation. The copy operation (Rule 1 in Table 7) trivially preserves Equation 12 by the inductive hypothesis.

The increment operation increments only a thread's own clock and version number (and only within a sampling period), so it trivially preserves Equation 12 as well.

The join operation for a thread's vector clock (Rules 4-6 in Table 7) sets $\mathcal{C}'_t.ver(o)$, but only after either verifying that $\mathcal{S}_o.vc \sqsubseteq \mathcal{C}_t.vc$ (Rule 5), or performing a vector clock join operation using $\mathcal{S}_o.vc$ as an argument (Rule 6). By Equation 3 (vector clock join), $\mathcal{S}'_o.vc \sqsubseteq \mathcal{C}'_t.vc$, and Equation 12 holds.

The vector clock join operation for volatiles is simply a copy operation in Rules 7 and 8, and thus satisfies Equation 12. In Rule 9, $\mathcal{V}'_{vx}.vc$ receives the result of a join operation, which means that it no longer contains a snapshot of a single thread's vector clock; it contains the join of multiple threads' vector clocks. Thus, a version epoch is no longer sufficient. In response, Rule 9 sets $\mathcal{V}'_{vx}.ver$ to $\top_{ve}$. $\top_{ve} \preceq \mathcal{C}'_t.ver$ is false by definition of $\top_{ve}$, so the implication in Equation 12 holds, as the antecedent is always false. Thus, by the inductive hypothesis, the rules in Table 6, and their definitions in Table 7, Equation 12 holds when the inductive hypothesis is true. □

LEMMA 8. (Vector clocks imply happens-before within a sampling period). *Suppose $\sigma_a \Rightarrow^{a.\alpha} \sigma_b \Rightarrow^b \sigma'_b$, where $s^a = true$, $s^b = true$, and $\forall d \in \alpha,\ s^d = true$. Let t be the thread that performs a and u be the thread that performs b. If $\mathcal{C}^a_t.vc(t) \leq \mathcal{C}^b_u.vc(t)$, then $a \xrightarrow{\text{HB}}_{a.\alpha} b$.*

Proof. By Lemmas 5 and 6 (strict well-formedness within a sampling period), state σ_a must be strictly well-formed. We refer the reader to the proof of a similar lemma for FASTTRACK that relies upon σ_a being strictly well-formed [11] ("well-formed" in FASTTRACK is equivalent to "strictly well-formed" in PACER). Within a sampling period, it should be clear that all actions that PACER performs have the same effect as those that FASTTRACK performs, except when PACER avoids joins via version numbers. Lemma 7 shows that PACER maintains an additional invariant:

$$Ver(o) \preceq \mathcal{C}_t.ver \implies \mathcal{S}_o.vc \sqsubseteq \mathcal{C}_t.vc$$

Because the vector clock join operation takes the pointwise maximum of all elements, if $\mathcal{S}_o.vc \sqsubseteq \mathcal{C}_t.vc$, then the operation $\mathcal{C}_t.vc = \mathcal{C}_t.vc \sqcup \mathcal{S}_o.vc$ is unnecessary, as it will always set $\mathcal{C}_t.vc$ equal to itself. The version numbers are used to avoid joins only in this scenario (see Rules 4, 5, 7, and 8 in Table 7). Thus, PACER effectively performs the same operations that FASTTRACK performs within a sampling period, and Lemma 8 follows from the FASTTRACK proof. □

DEFINITION 3. (Fully-sampled races). *Given two conflicting actions a and b such that $\sigma_a \Rightarrow^{a.\alpha} \sigma_b$ and $a \overset{\text{HB}}{\not\longrightarrow}_{a.\alpha} b$, a and b participate in a* fully-sampled race *if $s^a = true$, $s^b = true$, and $\forall d \in \alpha,\ s^d = true$.*

THEOREM 1. (Soundness for fully-sampled races). *If $\sigma \Rightarrow^\alpha \sigma'$ and $\forall a \in \alpha,\ s^a = true$, then α is race-free.*

Proof. By Lemmas 5 and 6 all states within a sampling period are strictly well-formed. By Lemma 8, vector clocks imply happens-before within a sampling period. Because all operations PACER performs within a sampling period are effectively equivalent to those that FASTTRACK performs, the proof proceeds similarly to the FASTTRACK soundness proof [11], substituting Lemmas 5, 6, and 8 to justify that states are strictly well-formed, and that vector clocks imply happens-before. □

DEFINITION 4. (Sampled races). *Given two conflicting actions a and b such that $\sigma_a \Rightarrow^{a.\alpha} \sigma_b$ and $a \overset{\text{HB}}{\not\to}_{a.\alpha} b$, a and b participate in a* sampled race *if $s^a = true$.*

DEFINITION 5. (Shortest races). *Given two conflicting actions a and b such that $\sigma_a \Rightarrow^{a.\alpha} \sigma_b$ and $a \overset{\text{HB}}{\not\to}_{a.\alpha} b$, a and b participate in a* shortest race *if there does not exist any intervening read or write action d such that d conflicts with b, $\alpha = \beta.d.\gamma$ and $d \overset{\text{HB}}{\not\to}_{a.\alpha} b$.*

THEOREM 2. (Statistical Soundness). *If $\sigma_0 \Rightarrow^{\alpha} \sigma'$ then α contains no sampled shortest races.*

Proof. Suppose that α contains a sampled shortest race, and thus includes actions a and b such that

$$a \overset{\text{HB}}{\not\to}_{\alpha} b \qquad \text{Assume} \qquad (13)$$

$$s^a = true \qquad '' \qquad (14)$$

$$a \text{ and } b \text{ participate in a shortest race} \qquad '' \qquad (15)$$

In a manner similar to FASTTRACK's soundness proof, the proof proceeds by lexicographic induction on the length of α and the length of γ [11].

If α and γ have length zero then α trivially does not contain sampled shortest races. Thus, without loss of generality, assume that $\alpha = \beta.a.\gamma.b$, that $\beta.a.\gamma$ does not contain any sampled shortest races, and

$$\sigma_0 \Rightarrow^{\beta} \sigma_a \Rightarrow^{a} \sigma'_a \Rightarrow^{\gamma} \sigma_b \qquad \text{IH} \qquad (16)$$

$$\sigma_0 \Rightarrow^{\beta} \sigma_a \Rightarrow^{a} \sigma'_a \Rightarrow^{\gamma} \sigma_b \Rightarrow^{b} \sigma'_b \qquad \text{To prove} \qquad (17)$$

Let t be the thread that performs a, and let u be the thread that performs b. If $t = u$ then t and u do not race by program order and we have a contradiction with Equation 13. Thus, assume $t \neq u$, and proceed with the inductive step, which we prove by contradiction.

If $a = wr(t, x)$, $b = rd(u, x)$, and the rule for b is "Read, *Same epoch*" (Rule 1 in Table 4), then there must exist a prior access $d = rd(u, x)$ that set the epoch value. If d is after a, then $a \xrightarrow{\text{HB}}_{\alpha} d \xrightarrow{\text{HB}}_{\alpha} b$ by induction on the length of γ.

If d is before a, then because a is in a sampling period, d must either occur within the same sampling period as a, or there must exist a $sbegin$ action that occurs after d but before a. If d occurred prior to the $sbegin$ action, then the $sbegin$ action would have incremented $\mathcal{C}_u.vc(u)$, and the epoch would have changed, violating the conditions for Rule 1 (thus a different rule would have handled this case). If d occurred in the same sampling period as a, then there could be no intervening fork, release, or volatile write actions by thread u, as those actions would have incremented $\mathcal{C}_u.vc(u)$ and changed the epoch. Thus, $d \overset{\text{HB}}{\not\to}_{\alpha} a$, and because both d and a occur in the same sampling period, by Theorem 1 PACER would have gotten stuck by the prior race, contradicting the assumption in Equation 16.

Otherwise, if the rule for b is not Rule 1, it must be the case that

$$W^b_x \preceq \mathcal{C}^b_u.vc \qquad \text{Table 4, Rules 2-7} \qquad (18)$$

because otherwise the analysis would become stuck. The value of W^b_x must either have been set by a, or have been set by some other access, d, that also wrote x. The inductive hypothesis ensures that $a \xrightarrow{\text{HB}}_{\alpha} d \xrightarrow{\text{HB}}_{\alpha} b$ in the case where d occurs after a, and if d occurs before a, then either $W'^d_x = \mathcal{C}^a_t.vc(t)@t$ already, or a replaces the write epoch for d with that value. In either case,

$$W^b_x = W'^a_x = \mathcal{C}^a_t.vc(t)@t \qquad \text{Table 4, Rules 5-7} \qquad (19)$$

By Equation 14 and Lemmas 5 and 6, we know that state σ_a must be strictly well-formed, and thus

$$\mathcal{C}^a_u.vc(t) < \mathcal{C}^a_t.vc(t) \qquad \text{Definition 2} \qquad (20)$$

$$W^b_x \not\preceq \mathcal{C}^a_u.vc \qquad \text{Equations 19 and 20} \qquad (21)$$

Thus, some action must increase the value of $\mathcal{C}'^a_u.vc(t)$ to satisfy Equation 18. The only actions that could satisfy that criteria ensure that $a \xrightarrow{\text{HB}}_{\alpha} b$, and thus we have a contradiction with Equation 13. □

The completeness proof for PACER proceeds in a similar manner to the completeness proof for FASTTRACK [11]. We use their abbreviation

$$K^a = \begin{cases} \mathcal{S}'^a & \text{if } a \text{ is a } join \text{ or } acq \text{ operation} \\ \mathcal{S}^a & \text{otherwise} \end{cases}$$

LEMMA 9. (Happens-before implies vector-clock ordering). *Suppose α is well-formed, $\sigma \Rightarrow^{\alpha} \sigma'$, and $a, b \in \alpha$. Let t be the thread that performs a, and let u be the thread that performs b.*

$$a \xrightarrow{\text{HB}}_{\alpha} b \implies K^a_t.vc \sqsubseteq K^b_u.vc \qquad \textit{To prove} \qquad (22)$$

Proof. By induction on the derivation of $a \xrightarrow{\text{HB}}_{\alpha} b$. Note that although PACER does not increment vector clocks during non-sampling periods, it does perform join operations during non-sampling periods. These join operations ensure that Equation 22 holds even when vector clocks do not increment. While vector clocks imply happens-before only during sampling periods in PACER, happens-before always implies vector clock ordering. □

THEOREM 3. (Completeness). *If α is race-free then $\sigma_0 \Rightarrow^{\alpha} \sigma$.*

Note that we define completeness in terms of a race-free program, rather than a program that is free of sampled, shortest races. We do this because sometimes, PACER will report a race that is not the shortest. We are indifferent to whether PACER reports such races (in fact, it is preferable if it does) and do not consider these races to be false positives. Thus, the weaker completeness criterion is sufficient.

Proof. The proof proceeds by contradiction in a manner similar to the FASTTRACK proof [11]. Suppose $\alpha = \beta.a.\gamma$ such that operation a is stuck,

$$\sigma_0 \Rightarrow^{\beta} \sigma' \not\Rightarrow^{a}$$

If a is stuck and t is the thread that performs a, then $a = rd(t, x)$ or $a = wr(t, x)$, and

$$W^a_x \not\preceq \mathcal{C}^a_t.vc \qquad \text{Table 4} \qquad (23)$$

Here we show the argument for a preceding write. Let $b = wr(u, x)$ be the most recent write to x that precedes a.

$$W^a_x = \begin{cases} W'^b_x & \text{if } s^b = \text{true} \\ \bot_e & \text{otherwise} \end{cases} \qquad \text{Table 4, Rules 5-7} \qquad (24)$$

If b did not occur within a sampling period, then

$$W^a_x = \bot_e \qquad \text{Equation 24} \qquad (25)$$

$$\bot_e \not\preceq \mathcal{C}^a_t.vc \qquad \text{Equations 23 and 25} \qquad (26)$$

and we have a contradiction, because by definition of $\bot_e$ the relation $\bot_e \preceq \mathcal{C}^a_t.vc$ always holds.

Otherwise, if b did occur within a sampling period, then

$$W'^b_x = \mathcal{C}^b_u.vc(u)@u \qquad \text{Table 7, Rules 5-7} \qquad (27)$$

$$= W^a_x \not\preceq \mathcal{C}^a_t.vc \qquad \text{Equations 24 and 23} \qquad (28)$$

Hence,

$$K^b_u.vc(u) = \mathcal{C}^b_u.vc(u) \not\sqsubseteq \mathcal{C}^a_t.vc(u) = K^a_t.vc(u)$$

By Lemma 9 $b \overset{\text{HB}}{\not\rightarrow}_\alpha a$ so the program is not race-free, and we have a contradiction. □

C. Handling Volatile Variables

This section details how prior GENERIC race detection and PACER handle synchronization operations involving volatile variables. The Java Memory Model states that each write to a volatile variable *happens before* subsequent reads of the same variable [21]. Volatiles are quite similar to locks—a volatile read is like a lock acquire, and a volatile write is like a lock release—except that a volatile read need not be followed by a volatile write on the same thread.

How* GENERIC *handles volatile variables. GENERIC uses the same synchronization metadata for each volatile field x that it uses for other synchronization objects: a vector clock C_x. Algorithms 14 and 15 show how GENERIC handles reads and writes to volatile variables. The analysis for a volatile read is *identical* to the analysis for a lock acquire. The analysis for a volatile write is similar to the analysis for a lock release, except the volatile write analysis performs a vector clock *join* instead of *copy*.

FASTTRACK does not introduce new analysis for synchronization operations; it uses the same algorithms as GENERIC for volatile variables.

How* PACER *handles volatile variables. PACER redefines low-level vector clock operations (join, increment, and copy) used in GENERIC's analysis for synchronization operations (Section 3.2). PACER uses the same synchronization operations for each volatile variable x as for each lock: $clock_x$ and $vepoch_x$. While PACER redefines vector clock join in Algorithm 11, it is only suitable when the target of the join is a thread, not a volatile variable, because it relies on the target having a versioned vector clock and a version vector. Thus PACER uses a special vector clock join for this case, shown in Algorithm 16. In non-sampling periods, the algorithm uses versions to detect if $clock_x \sqsubseteq clock_t$; if so, the join simply becomes a *shallow copy* from $clock_t$ to $clock_x$. Then the behavior is the same as at a lock release, i.e., the analysis copies the thread's clock to the volatile's clock.

Algorithm 14 Read volatile [GENERIC]: thread t reads volatile x

```
$C_t \leftarrow C_t \sqcup C_x$
```

Algorithm 15 Write volatile [GENERIC]: thread t writes volatile x

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

Algorithm 16 Vector clock join for assigning to a volatile's vector clock [PACER]: $C_x \leftarrow C_x \sqcup C_t$

```
Let $v@u = \textbf{vepoch}(x)$
$subsumes \leftarrow \textbf{false}$
if $sampling$ then
    if ($v@u \neq \textbf{null} \wedge ver_t[u] < v$) then          {Check version}
        $subsumes \leftarrow \textbf{true}$
    else if $clock_x \sqsubseteq clock_t$ then
        $subsumes \leftarrow \textbf{true}$
    end if
end if
if $subsumes$ then
    setShared($clock_t$, true)
    $clock_x \leftarrow_{shallow} clock_t$
    $vepoch_x \leftarrow \textbf{vepoch}(t)$
else
    if isShared($clock_x$) then
        $clock_x \leftarrow \textbf{clone}(clock_x)$
        setShared($clock_x$, false)
    end if
    $vepoch_x \leftarrow \textbf{null}$          {Cannot assign vepoch($t$)}
end if
```

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[2] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 314–324, Denver, CO, 1999.

[3] M. Arnold, M. Vechev, and E. Yahav. QVM: An Efficient Runtime for Detecting Defects in Deployed Systems. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 143–162, 2008.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[5] S. M. Blackburn and K. S. McKinley. Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *ACM Conference on Programming Language Design and Implementation*, pages 22–32, 2008.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, 2002.

[7] T. M. Chilimbi and M. Hauswirth. Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, 2004.

[8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *ACM Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

[9] M. Christiaens and K. D. Bosschere. Accordion Clocks: Logical Clocks for Data Race Detection. In *International European Conference on Parallel Processing*, pages 494–503, 2001.

[10] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-Aware Java Runtime. In *ACM Conference on Programming Language Design and Implementation*, pages 245–255, 2007.

[11] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *ACM Conference on Programming Language Design and Implementation*, pages 121–133, 2009.

[12] J. Gray. Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[13] T. A. Henzinger, R. Jhala, and R. Majumdar. Race Checking by Context Inference. In *ACM Conference on Programming Language Design and Implementation*, pages 1–13, 2004.

[14] M. Hirzel and T. Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–126, 2001.

[15] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic Object Sampling for Pretenuring. In *ACM International Symposium on Memory Management*, pages 152–162, 2004.

[16] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *ACM Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[18] B. R. Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California at Berkeley, 2004.

[19] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[20] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[21] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *ACM Symposium on Principles of Programming Languages*, pages 378–391, 2005.

[22] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. In *ACM Conference on Programming Language Design and Implementation*, pages 134–143, 2009.

[23] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1988.

[24] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[25] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, 1983.

[26] E. Pozniansky and A. Schuster. MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice & Experience*, 19(3):327–340, 2007.

[27] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *ACM Conference on Programming Language Design and Implementation*, pages 320–331, 2006.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.

[29] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[30] C. von Praun and T. R. Gross. Object Race Detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

[31] J. W. Voung, R. Jhala, and S. Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–214, 2007.

[32] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *ACM Symposium on Operating Systems Principles*, pages 221–234, 2005.