PACER: Proportional Detection of Data Races *

Michael D. Bond Katherine E. Coons Kathryn S. McKinley

Department of Computer Science, The University of Texas at Austin

{mikebond,coonske,mckinley}@cs.utexas.edu

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 after deployment. The frequency and unpredictability of these bugs will only increase as software adds parallelism to exploit multicore hardware. Unfortunately, sound and precise race detectors slow programs by factors of eight or more and do not scale to large numbers of threads.

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 dynamically 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 theoretical guarantees. PACER reports races in proportion to the sampling rate. Its time and space overheads scale with the sampling rate, and sampling rates of 1-3% yield overheads low enough to consider in production software. The resulting system provides a "get what you pay for" approach that is suitable for identifying real, hard-to-reproduce races in deployed systems.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Debuggers, Run-time environments; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Testing tools

General Terms Reliability, Performance, Experimentation

Keywords Concurrency, Data Races, Bugs, Sampling

1. Introduction

Software must become more parallel to exploit hardware trends, which are increasing the number of processors on each chip. Unfortunately, correct and scalable multithreaded programming is quite challenging. It is notoriously difficult to specify program synchronization, i.e., the ways in which threads may interleave operations on shared data. Too much synchronization degrades 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 [25], order violations [24], and sequential consistency violations [26]. Because some races occur only under certain inputs, environments, or thread schedules, deployed low-overhead race detection is necessary to achieve highly robust deployed software.

Static techniques for detecting races scale to large programs and try to limit false positives, typically by being unsound in a few limited ways [29; 33; 38]. *Precision* (no false positives) is important because both false and true data race reports take lots of developer time to understand, and thus developers are not embracing approaches that report false positives. Dynamic analysis typically uses either *lockset* or *vector clock* algorithms. Lockset algorithms reveal errors in locking disciplines, but are imprecise. Vector clock algorithms are precise and now achieve about the same performance as lockset algorithms [14].

Vector clock-based race detection is precise because it tracks the *happens-before* relationship. Recently FASTTRACK reduced most vector clock-based analysis time from $O(n)$ to $O(1)$, where n is the number of threads. FASTTRACK exploits the observation that some reads and all writes are totally ordered in race-free programs, but it still slows programs down by a factor of eight on average. LITERACE reduces overhead by *sampling* [27]. While LITERACE finds many races handily, it uses heuristics that provide no guarantees, incurs $O(n)$ overhead at synchronization operations, and has high online space overhead.

This paper presents a new approach for detecting data races based on sampling called PACER. PACER makes a proportionality guarantee: it detects each race with a probability equal to the sampling rate, and in practice it adds time and space proportional to the sampling rate. PACER builds on the FASTTRACK algorithm but reduces overhead through sampling. FASTTRACK finds *shortest races*. Two racy accesses A and B are a *shortest* race if there is no intervening access that races with B. PACER reports *sampled, shortest races*. Two racy accesses A and B are *sampled* if A occurs in a sampling period. B may occur any time later—in any subsequent sampling or non-sampling period. 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 discards the metadata to save time and space. (2) During non-sampling periods, we observe that because PACER must only determine if the *sampled* accesses happen before the current time, it is not necessary to increment vector clocks. Without increments, vector clock values converge due to redundant synchronization, which PACER exploits to reduce almost analysis from $O(n)$

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

to $O(1)$ time during non-sampling periods. The appendices prove PACER's completeness and statistical soundness.

PACER provides a *qualitative* improvement over most prior work. Its scalable performance makes it suitable for all-the-time use in production. For sampling rates of 1 to 3%, PACER adds overheads between 52 and 86%, which we believe could be lower with additional implementation effort and may already be low enough for many deployed settings. We show that PACER 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 evaluation (which currently evaluates only frequently occurring races due to experimental resource limitations) suggests that PACER achieves its theoretical guarantees, finding each dynamic data race with a probability equal to the sampling rate.

A given data race may occur extremely infrequently or never in testing and in some deployed environments, but occur periodically in other deployed environments. Widely deploying PACER gives coverage across many environments, with reasonable odds of finding any given race that occurs periodically. We note that potentially harmful data races *occur* quite frequently without causing errors *in the same execution*. Thus, sampling may be attractive even in safety-critical software, to identify data races before they lead to errors. In less critical software such as web browsers, PACER is attractive for identifying data races that frequently lead to errors; its reports can help developers detect, diagnose, and fix the root cause of previously undiagnosed nondeterministic crashes.

PACER is a "get what you pay for" approach that provides scalable performance and scalable odds of finding *any* race. PACER provides a *qualitative* improvement over prior approaches because it is suitable for all-the-time use in deployed systems, where it can help developers eliminate rare, tough-to-reproduce errors.

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 prior heuristics are unsatisfactory because they may miss races, and have unscalable time and memory overheads.

2.1 Race Detection Using Vector Clocks

The *happens-before* relationship computes a partial order over dynamic program statements [21]. Statement 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 $B \not\xrightarrow{\text{HB}} A$, i.e., they are 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. We follow prior work by considering *happens-before* data races because their absence guarantees sequential consistency.

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

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 $R_f \sqsubseteq C_t$
$W_f[t] \leftarrow C_t[t]$
```

this presentation. Appendix C explains the differences for volatile variables.) 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 literature: all these accesses are on *variables*, and synchronization operations are on *synchronization objects*.

Vector clock race detection algorithms soundly and precisely track the happens-before relationship [21; 28]. These algorithms perform *dynamic analysis* on all synchronization, read, and write operations. They detect concurrent variable accesses, and if one is a write, they report 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. A *vector clock* is indexed by thread identifier: $C[1..n]$. 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 GENERIC vector clock algorithms at lock acquires and releases, and thread forks and joins. Following Flanagan and Freund [14], 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 t's vector clock to m's vector clock. It then increments the t entry in t's vector clock.

Variable reads and writes. GENERIC tracks, for each variable, the logical time at which every thread last read and wrote it:

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

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

Algorithms 5 and 6 show GENERIC analysis for 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. At writes, the analysis checks for races with prior reads *and* writes, and updates the write vector.

2.2 FASTTRACK

FASTTRACK is a dynamically sound and complete race detection algorithm [14]. 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 almost always performs $O(1)$-time analysis on them. FASTTRACK exploits the following insights. (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 variable, FASTTRACK replaces the write vector clock with an *epoch* $c@t$, which records the thread t and its clock value c that last wrote the variable. 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 generalize the read epoch and vector clock into 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. A read map with zero entries is 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, 2, 3, and 4). 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 reads 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. 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 will also race with the discarded read. We modify FASTTRACK to clear R_f to 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 [14]. Notably, it performs about the same as imprecise lockset-based race detection, but it still slows programs by 8X on average and adds 3X space overhead, which is too inefficient for most deployed applications. (While the original FASTTRACK implementation executes in pure Java, we estimate that an efficient implementation inside a JVM would still slow programs by 3-4X. Our PACER implementation at a 100% sampling rate, while implemented in a JVM and functionally equivalent to FASTTRACK, incurs a 12X slowdown primarily because it is optimized for the non-sampling case at the expense of the sampling case; Section 5.4.) FASTTRACK's analysis for nearly all read and write operations takes $O(1)$ time; however, its analysis for synchronization operations takes $O(n)$ time. Although synchronization operations account for only about 3% of analyzed operations, as the number of threads increases, they present a scalability bottleneck.

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
        $R_f[t] \leftarrow C_t[t]$                {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 here}
    else
        check $R_f \sqsubseteq C_t$               {$O(1)$ amortized time}
        $R_f \leftarrow$ empty
    end if
    $W_f \leftarrow$ epoch($t$)                   {Update write epoch}
end if
```

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 has two serious problems. First, sampling synchronization operations will miss happens-before edges and thus will report false positive races. Second, because a race involves *two* accesses, sampling a proportion r of all reads and writes will report only r^2 of races (e.g., 0.09% for $r = 3\%$).

LITERACE solves some of these problems [27]. 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 [10]. LITERACE samples at a rate inversely proportional to execution frequency, down to a minimum. LITERACE thus cannot make claims on proportionality, since with its 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.

The LITERACE paper uses *offline* race detection by recording synchronization, read, and write operations to a log file [27]. Offline analysis performs checks for races in the log when, for example, an execution fails. Offline race detection is impractical in many cases, such as long-running programs. An *online* implementation of LITERACE requires $O(n)$ analysis for synchronization operations. Furthermore, since it samples code, rather than data, space overhead is proportional to the data, not the sampling rate.

2.4 Requirements

While recent work offers significant advances in dynamic, precise race detection, serious drawbacks limit its applicability: analysis that requires $O(n)$ time and space, and sampling heuristics that consistently miss some races. 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

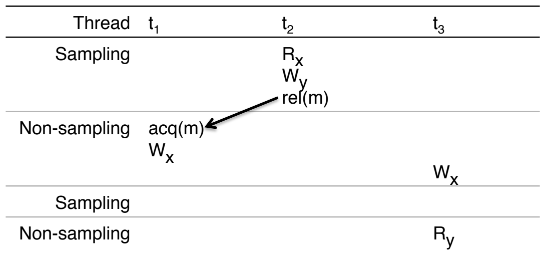

Figure 1: PACER reports the race on y, but *not* the race on x because the shortest race's first access is not sampled.

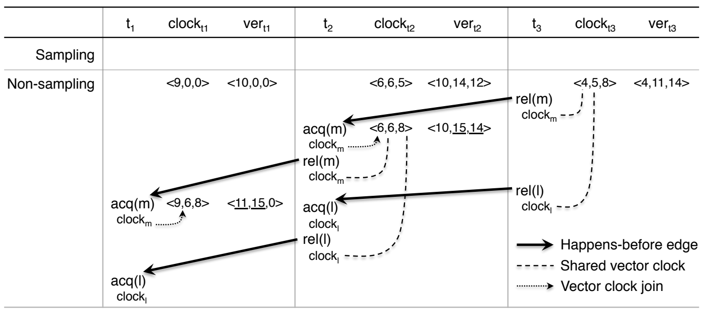

Figure 2: PACER exploits redundant communication in non-sampling periods to eliminate vector clock updates and to share vector clocks.

approach should find *any* race in proportion to how frequently the race occurs in an execution.

3. PACER

This section presents the PACER sampling-based race detection algorithm. 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. While some of these overheads are *also* proportional to the number of threads n, decreasing the sampling rate reduces overall overhead. PACER requires a fairly low sampling rate ($\leq 3\%$) to keep overhead low enough to consider deploying. Since the chance of finding a race in a given execution is fairly low ($\leq 3\%$), we envision developers using PACER on many deployed instances, as in distributed debugging frameworks [22; 23]. This section presents the PACER algorithms, which Appendices A and B formalize and prove correct.

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, shortest* races, defined as follows. Given two accesses A and B, PACER reports the race if A is in a sampling period and A is the *last* access that *races* with B. B can occur inside or outside a sampling period. The write-read race on y in Figure 1 shows a simple example. The write at t_2 occurs inside the sampling period and races with the read at t_3, which is outside the sampling period. 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 future accesses.

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 subsections describe in more detail how PACER maintains accuracy while reducing work.

3.2 Synchronization Operations

We use the following key insights to reduce vector clock $O(n)$ analysis in non-sampling periods.

- During non-sampling periods, PACER does not increment threads' clocks because it does not need to compare two access times from non-sampled periods. The analysis only needs to test if a sampled access happens before the current time.
- When PACER does not increment time, redundant communication produces the same vector clock values. By detecting and exploiting this redundancy, PACER eliminates redundant vector clock joins and copies, reducing time and space overhead.

Non-sampling periods are "timeless": the analysis stops incrementing thread vector clocks. As a result, it can avoid redundant vector clock joins and often share vector clock objects. Skipping clock increments is sufficient to track happens-before because during non-sampling periods, we only compare sampled to non-sampled vector clocks to find races. Thus, vector clocks will not change when communication is redundant. PACER detects this redundancy and avoids $O(n)$-time vector clock operations.

Consider Figure 2. In timeless periods, only lock acquire, fork, and join operations change the vector clock values. Therefore, clock$_m$ and clock$_l$ can share t_3's vector clock after the release operations. Furthermore, PACER omits unnecessary join operations. PACER must perform the join when t_2 acquires lock m, but the acquire of lock l produces a redundant clock value. PACER detects this case, performs no join, and shares vector clocks upon release.

To detect redundant communication, we introduce *vector clock versions* and *version vectors*.[1] PACER assigns a version number to every unique vector clock value a thread observes. It starts the version at zero and increments it 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. It also stores a *version epoch* $v@t$ for each lock that stores the last thread t and version v, if any, that released this lock. When thread t releases a lock, it sets the lock's version epoch to $v@t$ where v is thread t's current version.

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.

[1] These are not the same as version vectors used in distributed systems [31].

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
```

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 redefines these operations.

Algorithm 9 shows how PACER redefines vector clock copy. In a non-sampling period, the algorithm performs a shallow copy of the synchronization object, i.e., m and t share vector clocks ($clock_m = clock_t$). This sharing is worthwhile because the thread's vector clock is likely to have the same value for a while. In a sampling period, sharing is useless because the algorithms increment thread vector clocks immediately afterward, 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 right-hand side C_m may be a lock, thread, or volatile vector clock. 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 any 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., $\forall t\ C_t[t] \leftarrow C_t[t] + 1$.

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

```
if $sampling \vee (R_f \neq$ null $\vee W_f \neq$ null$)$ then
    if $R_f \neq$ epoch($t$) then            {If same epoch, no action}
        check $W_f \sqsubseteq clock_t$
        if $|R_f| \leq 1 \wedge R_f \sqsubseteq clock_t$ then
            if $sampling$ then
                $R_f \leftarrow$ epoch($t$)                {Update read map}
            else
                $R_f \leftarrow$ null                      {Discard read map}
            end if
        else
            if $sampling$ then
                $R_f[t] \leftarrow clock_t[t]$
            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
end if
```

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

```
if $sampling \vee (R_f \neq$ null $\vee W_f \neq$ null$)$ then
    if $W_f \neq$ epoch($t$) then            {If same epoch, no action}
        check $W_f \sqsubseteq clock_t$
        if $|R_f| \leq 1$ then
            check $R_f \sqsubseteq clock_t$
        else
            check $R_f \sqsubseteq clock_t$             {$O(1)$ amortized time}
        end if
        if $sampling$ then
            $W_f \leftarrow$ epoch($t$)              {Update write epoch}
        else
            $W_f \leftarrow$ null                    {Discard write epoch}
        end if
        $R_f \leftarrow$ null                        {Discard read map}
    end if
end if
```

Section 5.4 shows that in practice PACER avoids nearly all $O(n)$ analysis in non-sampling periods by using shallow copies and versions on joins and copies. We do not currently know the worst case for an adversarial program.

3.3 Reads and Writes

PACER reports sampled, shortest races: the first access must be sampled, and the first access must be the most recent that races with the second access. In sampling periods, PACER mimics the FASTTRACK algorithm. In non-sampling periods, PACER does not record read and write accesses, and discards the same read or write information that FASTTRACK overwrites or discards.

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 check if the current access is in the same epoch. If not, both analyses check for races with prior accesses. The next behavior depends on whether PACER is in a sampling period. If sampling, PACER updates the read map and write epoch exactly the same way as FASTTRACK. If not, it discards the read and write accesses that FASTTRACK replaces or discards. In particular, the analysis for a read discards zero or one prior read accesses. If the read map becomes empty, PACER assigns **null** to it. The analysis for a write always **null**s the read map and write epoch.

4. Implementation

We implemented PACER in Jikes RVM 3.1.0, a high-performance Java-in-Java virtual machine [3]. Jikes RVM's performance is competitive with commercial VMs as of November 2009.[2] PACER is publicly available on the Jikes RVM Research Archive.[3]

Metadata. Our implementation adds *two words* to the header of every object. The first word points to a hash table that maps field (variable) offsets to field read/write metadata. This hash table only uses space for metadata that PACER has sampled and has not discarded. When an object has no per-field metadata, instrumentation sets the header word to null. The second header word points to synchronization metadata. Although Java programs may synchronize on any object, many objects are never synchronized [5]. We thus only instantiate this pointer if the program locks the object. We use two words per object to simplify our implementation task. A one-word implementation is possible and would use less space [5].

Similarly, the implementation adds a word per static field for read/write metadata, which is reset to null if PACER discards 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 adds instrumentation to both compilers. In the optimizing compiler, the new PACER compiler pass uses Jikes RVM's static escape analysis to identify accesses to provably local data, which it does not instrument. The PACER optimizing compiler inserts the following instrumentation at reads and writes:

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

[3] http://www.jikesrvm.org/Research+Archive

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.

```
// 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 only during 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, or has never been sampled. Our implementation uses low-level synchronization (compare-and-swap) to properly synchronize accesses to synchronization and read/write metadata. Section 5.4 shows the overhead of this check alone is about 18%.

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*, which is convenient because GC stops all threads, but other points are possible. Turning sampling on and off without stopping all threads would likely be correct if the implementation propagates the `sampling` flag along happens-before edges, but we have not proved it.

We use the default generational mark-region collector [7]. Nursery collections occur frequently, every 32 MB of allocation. At the end of a collection, we turn on sampling with probability of r via pseudo-random number generation. At first glance, this mechanism should sample a random fraction r of program reads and writes. However, since race detection allocates lots of metadata, collections occur more frequently and less program work occurs between two collections during sampling. We instead measure 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 that this mechanism achieves actual, *effective sampling rates* (plus or minus one standard deviation) very close to various *specified* target sampling rates. The effective rate is sometimes *lower*, e.g., hsqldb for a 1% target sampling rate, because the correction mechanism does not have 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, benchmarks, and races. We evaluate accuracy and overhead at various sampling rates and compare accuracy to LITERACE [27]. We experimentally confirm our theoretical results: PACER accurately reports races in proportion to the sampling rate, with overhead proportional to the sampling rate.

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.

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

Table 2: Thread counts and race counts.

Benchmarks. We use the multithreaded DaCapo benchmarks [6] (eclipse, hsqldb, and xalan; version 2006-10-MR1) and pseudojbb, a fixed-workload version of SPECjbb2000 [35]. The DaCapo benchmark lusearch is multithreaded, but Jikes RVM 3.1.0 does not run it correctly with or without PACER.

Threads. Columns 2 and 3 of Table 2 show the number of threads 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 [12].

Races and trials. Dynamically detecting races is challenging because some races occur infrequently. Another challenge is that the *observer effect* may introduce *heisenbugs* [16]; 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.

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.

Columns 4-8 of Table 2 count *distinct* races reported by PACER, i.e., it counts each reported static pair of program references, even if the race occurs multiple times in an execution or in multiple executions. Columns 4-6 (r = 100%) report races from the 50 trials executed at a 100% sampling rate. These columns report races that occur in at least 1, 5, and 25 trials, respectively. Columns 7 and 8 ($\forall r$) report the races observed in either the 50 fully accurate executions (r = 100%) or in any of over 1,000 executions at r = 1-25%. Column 7 reports races that occurred in at least 1 trial, and column 8 reports races that occurred in at least 5 trials. Comparing columns 4 and 5 with 6, and column 7 with 8, shows these programs have some rare races.

While PACER can find 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 5,000 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 for the expected number of times it will be reported to be 1. To bound experimentation time, we evaluate the accuracy of PACER on the races that appear in at least half of our 50 fully sampled executions (column 6, in bold). 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*.

While we evaluate only frequent races due to time and resource limits, future work should evaluate PACER on *infrequent* races. One open question is whether the observer effect is more significant for infrequent races than for frequent ones, which would make the evaluation more challenging.

Theoretical accuracy and performance. The following table summarizes the effect FASTTRACK and PACER have on (1) the detection rate for any race and (2) program performance for sampling rate r and data race occurrence rate o.

	Det. rate	**Slowdown**
FASTTRACK	o	$c_{rw} + c_{sync}n$
PACER	$o \times r$	$c_{sampling}(c_{rw} + c_{sync}n)r + c_{nonsampling}$

Constant c_{rw} is the slowdown due to analysis at reads and writes, and $c_{sync}n$ is the linear slowdown in the number of threads n due to analysis at synchronization operations. PACER essentially scales FASTTRACK's overhead by r, as well as a small constant factor $c_{sampling}$ due to PACER's additional complexity (e.g., indirect metadata lookups). PACER adds a slowdown $c_{nonsampling}$ during non-sampling periods, which is small and near-constant in practice.

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. We note that not all detected races are necessarily bugs, and PACER does not show how races might lead to errors or how to reproduce races.

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 per run 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 higher because PACER's chances of detecting a race improve if the race occurs multiple times in a run.

Per-race detection. The four graphs in Figure 5 plot the detection rate for each *distinct* evaluation race as a function of r. The x-axis sorts races by detection rate. We sort races *independently* for each sampling rate. Due to statistical error and heisenbugs, we cannot expect perfect results, i.e., each race's detection rate equal to the sampling rate. Nonetheless, the results are compelling. The only race PACER misses completely occurs in eclipse with 1% sampling. On average, detection rates correspond well with specified sampling rates.

5.3 Comparison to LITERACE

For comparison, we implemented an online version of LITERACE [27]. LITERACE lowers overhead using a sampling heuristic that hypothesizes that most races occur in cold code [10]. It 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 [19].

Our LITERACE implementation adaptively samples, lowering the sampling rate for each method-thread pair from 100% to 0.1%.

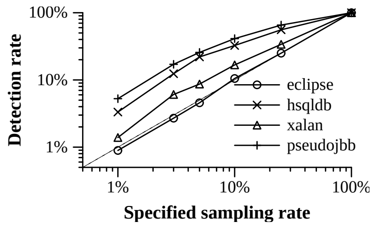

Figure 3: PACER's accuracy on *dynamic* races.

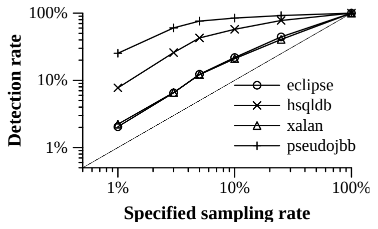

Figure 4: PACER's accuracy on *distinct* races.

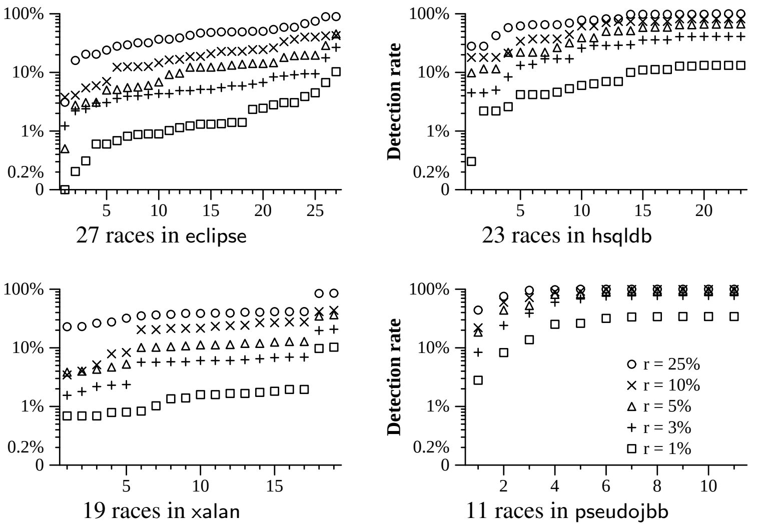

Figure 5: PACER's per-race accuracy (pseudojbb legend applies to all graphs).

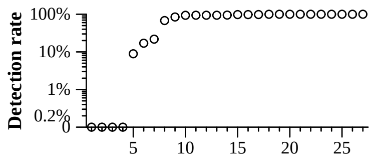

Figure 6: LITERACE's per-race accuracy for eclipse.

Whereas the original LITERACE is deterministic, our implementation adds randomness when resetting the sampling counter, to increase the chances of catching additional distinct races across multiple trials. However, we also experimented with deterministic sampling, and the results were not noticeably different. We initially used a sampling burst length of 10, but for all benchmarks 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 involving 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 LITERACE's space overhead, 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 samples most live memory. Section 5.4 explains the other data in Figure 10.

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

5.4 Performance

Figure 7 presents the time 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 the following configurations with monotonically increasing functionality. *OM + sync ops, r = 0%* 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 header words. *Pacer, r = 0%*, adds instrumentation at reads and writes but never executes the slow path. 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%.

These overheads are low enough for some but not all deployed settings. Lower overheads could be achieved in non-sampling periods. For example, aggressive compiler optimizations could hoist and statically simplify much of the instrumentation added to reads and writes (Section 4). Modest hardware support [2; 17] could make PACER's checks virtually free in non-sampling periods.

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 results for 0, 1, and 3% 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, PACER is functionally equivalent to FASTTRACK, but PACER slows programs by 12X on average, compared with 8X in the FASTTRACK paper [14]. There are two reasons for this difference. First, our implementation is optimized for low sampling rates: it uses hash tables instead of direct lookup—both for per-field metadata and for read maps—and it inlines the non-sampling case, which decreases PACER's overhead in non-sampling periods but increases overhead in sampling periods. Second, we evaluate PACER on different benchmarks with more threads than Flanagan and Freund used to evaluate FASTTRACK.

Avoiding expensive operations. Table 3 shows the number of $O(n)$- vs. $O(1)$-time vector clock operations, and slow vs. fast path reads and writes for PACER at $r = 3\%$, averaged over 10 trials. 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.

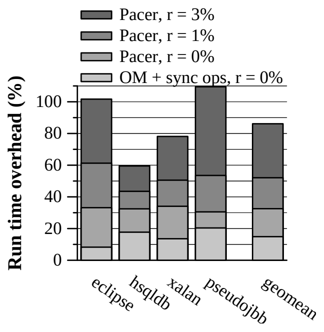

Figure 7: PACER overhead breakdown for r = 0–3%.

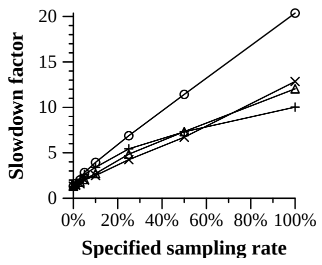

Figure 8: Performance vs. sampling rate for r = 0–100%.

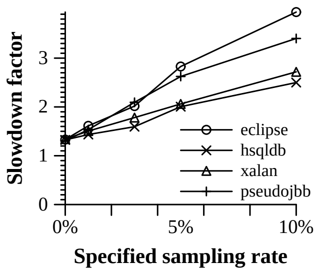

Figure 9: Performance vs. sampling rate for r = 0–10%.

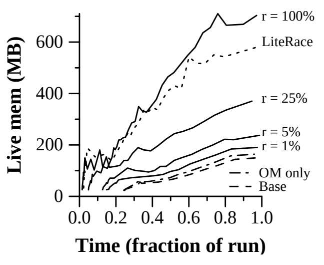

Figure 10: Total space over normalized time for eclipse.

	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	–	15	147,458K
hsqldb	241K	–	363	7,938K
xalan	8,179K	–	8	275,760K
pseudojbb	4,072K	–	30	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%.

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. Slow path operations in non-sampling periods correspond well with slow path operations in sampling periods. Because PACER checks for races with the last write and/or read to a variable in a sampling period, it inevitably 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 (reachable) 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 (except LITERACE) are PACER at various sampling rates. The graph shows that PACER's space overhead scales well with the sampling rate.

In practice, PACER 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 [14] and LITERACE [27]. 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 [9]. 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 [29; 33; 38]. These approaches scale to millions of lines of code, are typically sound except for a few exceptions, and 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 [18].

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 [11]. 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 [11; 34]. 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. On the other hand, lockset can identify true races that do not violate the happens-before race in the *current* execution. 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 [14].

Vector clocks. Prior techniques, including FASTTRACK and LITERACE, use vector clocks to achieve precise race detection [14; 27]. They both decrease analysis overhead at reads and writes, but analysis at synchronization operations still takes $O(n)$ time, so these approaches will not scale to many threads. In contrast, PACER can scale to many threads by adjusting the sampling rate.

Hybrid techniques. Hybrid techniques combine lockset and vector clocks to obtain the performance of the former and the accuracy of the latter [13; 37; 39]. *Goldilocks* is sound and precise and reports overheads low enough to deploy [13]. However, as Flanagan and Freund note [14], 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 [32]. FASTTRACK further improves on MultiRace's vector clock-based detector (called Djit$^+$) [14].

Classifying races. Recent work identifies *harmful* races by replaying racy accesses and evaluating the range of possible outcomes under a weak memory model [15; 30].

6.3 Sampling

Thakur et al. use global sampling to capture cross-thread memory interleavings [36] as predicates for statistical bug isolation [22; 23]. Their approach finds a variety of concurrency bug types without requiring complex analysis such as tracking synchronization operations. On the other hand, PACER does not require observable failures in order to find races, detects races without combining multiple runs' results, and uses a global sampling mechanism that does not require both accesses to be sampled.

Object-centric sampling tracks only a subset of objects, chosen at allocation time [4; 20]. 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 [4].

7. Conclusions and Future Work

Data races often 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. PACER achieves a *qualitative* improvement over prior work: its adjustable performance and accuracy make it suitable for all-the-time use in a variety of deployed environments.

Future work should evaluate infrequent races (*Races and trials* in Section 5.1); consider the worst case for version vectors (Section 3.2); and optimize the instrumentation that is always executed in non-sampling periods (Section 5.4). At a high level, PACER's cross-thread sampling approach may be applicable to other concurrency bugs.

Acknowledgments

We thank Bert Maher, Ben Wiedermann, Sam Guyer, and Satish Narayanasamy for helpful discussions; Bert Maher, Jingling Xue, Cormac Flanagan, Steve Freund, Dimitris Prountzos, and the anonymous reviewers for helpful feedback on the text; Steve Freund for explaining FASTTRACK details; and Steve Blackburn and Robin Garner for improving pseudojbb.

A. Formal Semantics

This section formalizes PACER's algorithm, and the next section proves statistical soundness and completeness. We introduce notation better suited to formalism that closely matches FASTTRACK [14]. An extended technical report version of this paper includes details that we omit in the next two sections [8].

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

$\mathit{rd}(t, f)$: thread t reads data variable f.
$\mathit{wr}(t, f)$: thread t writes data variable f.
$\mathit{acq}(t, m)$: thread t acquires lock m.
$\mathit{rel}(t, m)$: thread t releases lock m.
$\mathit{tfork}(t, u)$: thread t forks a new thread, u.
$\mathit{tjoin}(t, u)$: thread t blocks until thread u terminates.
$\mathit{vol\_rd}(t, x)$: thread t reads volatile variable x.
$\mathit{vol\_wr}(t, x)$: thread t writes volatile variable x.
$\mathit{sbegin}()$: the analysis enters a sampling period.
$\mathit{send}()$: the analysis leaves a sampling period.

The *acq*, *rel*, *tfork*, *tjoin*, *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. The helper function $\mathit{tid}(a)$ returns the thread that performs action a.

A *trace* α captures a sequence of actions. The trace $\alpha.b$ results when trace α is extended with action b. An action a relates to an action b in trace α by *program order* if a is before b in α, and $\mathit{tid}(a) = \mathit{tid}(b)$. Action a relates to action b in α by *synchronization order* if a and b are both synchronization actions, and a is before b in α. Action a relates to an action b in α by *synchronizes-with order* if they are also related by synchronization order, and:

- a releases a lock m and b is an acquire of m by any thread,
- a writes a volatile variable x and b is a read of x by any thread,
- a is a fork by t of a new thread, u, $t \neq u$, and $\mathit{tid}(b) = u$, or
- $\mathit{tid}(a) = u$, and b is a join that blocks a thread t, $t \neq u$, until u terminates.

The happens-before relation is the transitive closure of program order and synchronizes-with order [26]. If action a relates to action b in α by the happens-before relation then a *happens before* b, $a \xrightarrow{\text{HB}}_\alpha b$. If $a \not\xrightarrow{\text{HB}}_\alpha b$ and $b \not\xrightarrow{\text{HB}}_\alpha a$, then a and b are *concurrent* in α. Actions a and b *conflict* if they read or write the same variable, and at least one of them is a write. A trace α contains a *data race* if it contains conflicting, concurrent actions on data variables.

We restrict our attention to traces that are feasible and that obey Java's synchronization operation semantics.

A.1 Vector Clocks, Epochs, and Read Maps

A vector clock $\mathit{Clock} : \mathit{Tid} \to \mathit{Nat}$ maps threads to natural numbers representing logical clock values. The term $C[t] \in \mathit{Nat}$ denotes the clock value to which vector clock C maps thread t.

Vector clocks are pointwise partially ordered; vector clock C_1 is pointwise less-than 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 vector clock $\bot_c \in \mathit{Clock}$ is the vector clock that maps every thread to 0, $\bot_c = \lambda t.0$. We define two operations on vector clocks, increment and join ($\sqcup$):

$$\begin{aligned} inc_t(C) &= \lambda u. \text{ if } u = t \text{ then } C[u] + 1 \text{ else } C[u] \\ C_1 \sqcup C_2 &= \lambda t. max(C_1[t], C_2[t]) \end{aligned}$$

An epoch $\mathit{Epoch} : \mathit{Nat} \times \mathit{Tid}$ is a pair consisting of a natural number representing a logical clock value, c, and a thread identifier, t, written $c@t$. The term $\bot_e$ denotes the minimal epoch 0@0. 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:

$$c@t \preceq C \text{ iff } c \leq C[t]$$

A read map $\mathit{ReadMap} : \mathit{Tid} \rightarrow \mathit{Nat}$ maps any number of threads to logical clock values. Any thread not explicitly mapped implicitly maps to the minimal clock value, zero. The *magnitude* $|R|$ of read map R is a natural number indicating the number of threads explicitly mapped by R. An epoch is a special case of a read map with magnitude one. The relation $R \sqsubseteq C$ holds for read map R and vector clock C if and only if R is pointwise less than or equal to C for all values explicitly mapped by R.

A.2 Version Vectors and Version Epochs

PACER associates a version vector with each thread and increments that version vector whenever the thread's vector clock changes. A version vector $\mathit{Version} : \mathit{Tid} \rightarrow \mathit{Nat}$ maps threads to natural numbers representing vector clock *versions*. The term $V[t]$ denotes the version to which version vector V maps thread t. A minimal version vector $\bot_v \in \mathit{Version}$ maps all threads to 0, $\bot_v = \lambda t.0$. We define an increment helper function for a version vector V:

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

PACER associates a concise *version epoch* with each lock and each volatile variable. A version epoch $\mathit{Vepoch} : \mathit{Nat} \times \mathit{Tid}$ is a pair, written $v@t$, where t is a thread and v is a version number for t's vector clock. The relation $v@t \preceq V$ holds for version epoch $v@t$ and version vector V if and only if v is less than or equal to the t component in V:

$$v@t \preceq V \text{ iff } v \leq V[t]$$

The minimal version epoch $\bot_{ve}$ is equivalent to 0@0 such that $\bot_{ve} \preceq V$ is always true. The term $\top_{ve} \in \mathit{Vepoch}$ denotes a unique maximal version epoch such that $\top_{ve} \preceq V$ is never true.

A.3 Analysis State

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

$$\begin{aligned} &C : (\mathit{Tid} \cup \mathit{Lock} \cup \mathit{Volatile}) \rightarrow \mathit{Clock} && R : \mathit{Var} \rightarrow \mathit{ReadMap} \\ &V : \mathit{Tid} \rightarrow \mathit{Version} && W : \mathit{Var} \rightarrow \mathit{Epoch} \\ &E : (\mathit{Lock} \cup \mathit{Volatile}) \rightarrow \mathit{Vepoch} && s : \mathit{boolean} \end{aligned}$$

C maps synchronization variables to vector clocks. V maps threads to their version vectors, and E maps locks and volatile variables to their version epochs. R maps each data variable to its read map, and W maps each data variable to its write epoch. Finally, s is a boolean indicating whether or not PACER is in a sampling period.

Subscripts reference mapped values; for example, C_t refers to the vector clock to which C maps thread t. The abbreviation $\mathbf{epoch}(t) \in \mathit{Epoch}$ denotes the *current epoch* for thread t:

$$\mathbf{epoch}(t) = C_t[t]@t$$

The abbreviation $\mathbf{vepoch}(o) \in \mathit{Vepoch}$ denotes the *current version* for $o \in (\mathit{Tid} \cup \mathit{Lock} \cup \mathit{Volatile})$:

$$\mathbf{vepoch}(o) = \begin{cases} V_o[o]@o & \text{if } o \in \mathit{Tid} \\ E_o & \text{otherwise} \end{cases}$$

The initial analysis state, σ_0, contains the following components:

$$\sigma_0 = (C_0, V_0, E_0, R_0, W_0, s_0) \text{ where} \tag{1}$$

$$C_0 = \begin{cases} \lambda t. inc_t(\bot_c) \\ \lambda m. \bot_c \\ \lambda x. \bot_c \end{cases} \quad \begin{aligned} &V_0 = \lambda t. inc_t(\bot_v) \\ &E_0 = \begin{cases} \lambda m. \bot_{ve} \\ \lambda x. \bot_{ve} \end{cases} \end{aligned} \quad \begin{aligned} R_0 &= \lambda f. \bot_e \\ W_0 &= \lambda f. \bot_e \\ s_0 &= \mathit{false} \end{aligned}$$

PACER increments the initial vector clock and version vector for each thread and initializes everything else to its minimal value.

An action a transitions PACER from one analysis state to another, $\sigma \Rightarrow^a \sigma'$. Tables 4-6 show the analysis state updates for each action. When multiple rules exist for a single action (e.g. Rules 1-3 in Table 4), the rules appear in order of precedence. Table 4 shows how PACER updates the analysis state on reads and writes. When a condition labeled "*Race-free*" is not satisfied, no rule in Table 4 transitions PACER to a new state, and the analysis becomes *stuck*:

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

When stuck, PACER reports a data race. (The PACER *implementation* continues executing and reporting races.) The term $\sigma \Rightarrow^a \sigma'$ therefore indicates no race reported when action a occurs from σ.

Table 5 shows how *sbegin* and *send* modify the analysis state at the start and end of a sampling period. Table 6 shows how PACER updates the analysis state when synchronization actions occur, and Table 7 defines the copy, increment, and vector clock join operations used in Table 6. For example, consider the action $acq(t, m)$ from state σ. Rule 1 in Table 6 requires $\mathbf{vcjoin}(t, m)$, which corresponds to Rules 4-6 in Table 7. If $E_m \preceq V_t$, then Rule 4 applies, no updates occur, and the analysis transitions to σ'.

We do not address shallow and deep copies of vector clocks because we believe that their correctness will be clear to readers. PACER always creates a deep copy of a shared vector clock prior to modifying it. Whenever PACER creates a shallow copy, it marks the vector clock shared. Once a vector clock is shared it remains shared for the rest of its lifetime.

B. PACER Correctness Proofs

We prove PACER's completeness and statistical soundness. Because PACER builds on FASTTRACK, the proofs are similar and sometimes identical [14]. We highlight key differences here and expand proofs in the technical report [8]. The key intuition for PACER's statistical soundness is that while FASTTRACK's analysis state is always strictly well-formed, PACER's is strictly well-formed only during sampling periods.

The components of analysis states σ and σ' are as follows:

$$\begin{aligned} \sigma &= (C, V, E, R, W, s) \\ \sigma' &= (C', V', E', R', W', s') \end{aligned}$$

Similarly, $\mathbf{vepoch}'(o)$ is the current version of o in state σ'.

When necessary to avoid ambiguity, σ_a denotes the state prior to action a, and σ'_a denotes the state after action a: $\sigma_a \Rightarrow^a \sigma'_a$. The components of states σ_a and σ'_a are as follows:

$$\begin{aligned} \sigma_a &= (C^a, V^a, E^a, R^a, W^a, s^a) \\ \sigma'_a &= (C^{a\prime}, V^{a\prime}, E^{a\prime}, R^{a\prime}, W^{a\prime}, s^{a\prime}) \end{aligned}$$

Symbols α, β, and γ denote arbitrary-length sequences of actions.

Conditions	State updates		
	$s = true$	$s = false$	
Read data variable: $a = rd(t, f)$, $\sigma' = (C, V, E, R', W, s)$			
Same epoch $R_f = \mathbf{epoch}(t)$	None	None	(1)
Exclusive $\lvert R_f \rvert \leq 1 \wedge R_f \sqsubseteq C_t$ *Race-free* $W_f \preceq C_t$	$R_f{}' = \mathbf{epoch}(t)$	$R_f{}' = \bot_e$	(2)
Shared $\lvert R_f \rvert > 1 \vee R_f \not\sqsubseteq C_t$ *Race-free* $W_f \preceq C_t$	$R_f{}'[t] = C_t[t]$	$R_f{}'[t] = 0$	(3)
Write data variable: $a = wr(t, f)$, $\sigma' = (C, V, E, R', W', s)$			
Same epoch $W_f = \mathbf{epoch}(t)$	None	None	(4)
Exclusive or *Shared* $W_f \neq \mathbf{epoch}(t)$ *Race-free* $R_f \sqsubseteq C_t \wedge W_f \preceq C_t$	$R_f{}' = \bot_e$ $W_f{}' = \mathbf{epoch}(t)$	$R_f{}' = \bot_e$ $W_f{}' = \bot_e$	(5)

Table 4: Analysis state updates for data reads and writes. Column 1 shows conditions for each rule, and Columns 2 and 3 show state updates for sampling and non-sampling periods, respectively.

Action a	Next state σ'	State updates	
sbegin$()$	(C', V', E, R, W, s')	$s' = true$ $\lambda t.\mathbf{inc}(t)$	(1)
send$()$	(C, V, E, R, W, s')	$s' = false$	(2)

Table 5: Analysis state updates at sampling period start and end.

Action a	Next state σ'	State updates	
acq(t, m)	(C', V', E, R, W, s)	$\mathbf{vcjoin}(t, m)$	(1)
rel(t, m)	(C', V', E', R, W, s)	$\mathbf{copy}(m, t)$ $\mathbf{inc}(t)$	(2)
tfork(t, u)	(C', V', E, R, W, s)	$\mathbf{vcjoin}(u, t)$ $\mathbf{inc}(t)$	(3)
tjoin(t, u)	(C', V', E, R, W, s)	$\mathbf{vcjoin}(t, u)$ $\mathbf{inc}(u)$	(4)
vol_rd(t, x)	(C', V', E, R, W, s)	$\mathbf{vcjoin}(t, x)$	(5)
vol_wr(t, x)	(C', V', E', R, W, s)	$\mathbf{vcjoin}(x, t)$ $\mathbf{inc}(t)$	(6)

Table 6: Analysis state updates for synchronization actions. See Table 7 for copy, increment, and vector clock join operations.

Definition 1. (Well-formedness). *State* $\sigma = (C, V, E, R, W, s)$ *is* well-formed *if* $\forall t, u \in \mathit{Tid}, m \in \mathit{Lock}, f \in \mathit{Var}, x \in \mathit{Volatile}$:

$$\begin{array}{ll} C_u[t] \leq C_t[t] & R_f[t] \leq C_t[t] \\ C_m[t] \leq C_t[t] & W_f[t] \leq C_t[t] \\ C_x[t] \leq C_t[t] & \end{array}$$

Lemma 1. σ_0 *is well-formed.*

Conditions	State updates	
$\mathbf{copy}(o, t)$, $o \in (\mathit{Lock} \cup \mathit{Volatile}), t \in \mathit{Tid}$		
None	$C_o{}' = C_t$ $E_o{}' = \mathbf{vepoch}(t)$	(1)
$\mathbf{inc}(t)$, $t \in \mathit{Tid}$		
Non-sampling $s = false$	None	(2)
Sampling $s = true$	$C_t{}' = inc_t(C_t)$ $V_t{}' = inc_t(V_t)$	(3)
$\mathbf{vcjoin}(t, o)$, $t \in \mathit{Tid}, o \in (\mathit{Tid} \cup \mathit{Lock} \cup \mathit{Volatile})$ Let $v@u = \mathbf{vepoch}(o)$		
Same version epoch $v \leq V_t[u]$	None	(4)
Happens-before $C_o \sqsubseteq C_t$	$V_t{}'[u] = v$ { if $v@u \neq \top_{ve}$ }	(5)
Concurrent $C_o \not\sqsubseteq C_t$	$C_t{}' = C_t \sqcup C_o$ $V_t{}' = inc_t(V_t)$ $V_t{}'[u] = v$ { if $v@u \neq \top_{ve}$ }	(6)
$\mathbf{vcjoin}(x, t)$, $x \in \mathit{Volatile}, t \in \mathit{Tid}$ Let $v@u = \mathbf{vepoch}(x)$		
Same version epoch $s = true \wedge v \leq V_t[u]$	$\mathbf{copy}(x, t)$	(7)
Happens-before $s = true \wedge C_x \sqsubseteq C_t$	$\mathbf{copy}(x, t)$	(8)
Concurrent $s = false \vee C_x \not\sqsubseteq C_t$	$C_x{}' = C_x \sqcup C_t$ $E_x{}' = \top_{ve}$	(9)

Table 7: Copy, increment, and vector clock join operations.

Lemma 2. (Preservation of well-formedness). *If* σ *is well-formed and* $\sigma \Rightarrow^a \sigma'$ *then* σ' *is well-formed.*

Definition 2. (Strict well-formedness). *State* σ *is* strictly well-formed *if* σ *is well-formed and* $\forall t, u \in \mathit{Tid}, m \in \mathit{Lock}, x \in \mathit{Volatile}$:

$$\begin{array}{ll} \text{if } t \neq u \text{ then } C_u[t] < C_t[t] & C_m[t] < C_t[t] \\ & C_x[t] < C_t[t] \end{array}$$

Lemma 3. (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.*

Lemma 4. (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.*

Lemma 5. (Sampled vector clocks imply happens-before). *Suppose* σ_a *is strictly well-formed,* $s^a = true$*, and* $\sigma_a \Rightarrow^{a.\alpha} \sigma_b \Rightarrow^b \sigma'_b$*. Let* $t = tid(a)$ *and* $u = tid(b)$*. If* $C^a_t[t] \leq C^b_u[t]$ *then* $a \xrightarrow{\text{HB}}_{a.\alpha.b} b$*.*

Proof. Because $s^a = true$, σ_a is strictly well-formed and the proof proceeds similarly to the FastTrack proof that vector clocks imply happens-before [14], by induction on the length of α. ☐

Definition 3. (Sampled races). *Two conflicting actions* a *and* b *in a trace* α *participate in a* sampled race *if* a *occurs before* b *in* α*,* $a \not\xrightarrow{\text{HB}}_\alpha b$*, and* $s^a = true$*.*

Definition 4. (Shortest races). *Two conflicting actions* a *and* b *in a trace* α *participate in a* shortest race *if* a *occurs before* b *in*

α, $a \overset{\text{HB}}{\not\to}_\alpha b$, and there does not exist any read or write action d between a and b such that d conflicts with b and $d \overset{\text{HB}}{\not\to}_\alpha b$.

THEOREM 1. (Statistical Soundness). *If $\sigma_0 \Rightarrow^\alpha \sigma$ then α contains no sampled, shortest races.*

Proof. Suppose that α does contain a sampled, shortest race, and thus contains conflicting read/write actions a and b such that

$$a \overset{\text{HB}}{\not\to}_\alpha b \qquad \text{Assume} \qquad (2)$$
$$s^a = true \qquad '' \qquad (3)$$
$$a \text{ and } b \text{ participate in a shortest race} \qquad '' \qquad (4)$$

Proceed by lexicographic induction on the length of α and the number of intervening actions between a and b. Without loss of generality, assume that the segment of α prior to b does not contain any sampled, shortest races, and that b does not race with any actions prior to a. Let $t = tid(a)$, and let $u = tid(b)$. If $t = u$ then a and b do not race by program order so assume $t \neq u$.

If $a = wr(t, f)$, $b = rd(u, f)$, and Rule 1 in Table 4 applies for b, then a read $d = rd(u, f)$ must have set $R_f = \mathbf{epoch}(u)$. If d is after a, then $a \xrightarrow{\text{HB}}_\alpha d \xrightarrow{\text{HB}}_\alpha b$ by induction and program order, contradicting Equation 2. Action d must be within the same sampling period as a, because otherwise a *sbegin* action would have incremented $C_u[u]$ and changed the epoch, thus $s^d = true$. No intervening fork, release, or volatile write actions by thread u exist because they would increment $C_u[u]$ and change the epoch. Thus d and a race and α contains a prior sampled, shortest race.

Similarly, if $b = wr(u, f)$ and Rule 4 in Table 4 applies for b, then a write $d = wr(u, f)$ must have set $W_f = \mathbf{epoch}(u)$, and a similar argument applies.

Otherwise,

$$W_f^b \preceq C_u^b \qquad \text{Table 4, Rules 2, 3, and 5} \qquad (5)$$

If an intervening write d occurs between a and b then $a \xrightarrow{\text{HB}}_\alpha d \xrightarrow{\text{HB}}_\alpha b$ by induction and Equation 4, thus there are no intervening writes. If a is a write then

$$W_f^{a\prime} = W_f^b = C_t^a[t]@t \qquad \text{Equation 3; Table 4, Rules 4-5} \qquad (6)$$
$$C_t^a[t] \leq C_u^b[t] \qquad \text{Equations 5 and 6} \qquad (7)$$

thus by Lemma 5 $a \xrightarrow{\text{HB}}_\alpha b$, contradicting Equation 2. A similar argument applies for R_f when a is a read and b is a write. □

PACER's completeness proof uses the following abbreviation [14]:

$$K^a = \begin{cases} C^{a\prime} & \text{if } a \text{ is a } \textit{tjoin} \text{ or } \textit{acq} \text{ operation} \\ C^a & \text{otherwise} \end{cases}$$

LEMMA 6. (Happens-before implies vector-clock ordering). *Suppose α is well-formed, $\sigma \Rightarrow^\alpha \sigma'$, and $a, b \in \alpha$. Let $t = tid(a)$, and let $u = tid(b)$. If $a \xrightarrow{\text{HB}}_\alpha b$, then $K_t^a \sqsubseteq K_u^b$*

Proof. By induction on the derivation of $a \xrightarrow{\text{HB}}_\alpha b$. □

THEOREM 2. (Completeness). *If α is race-free then $\sigma_0 \Rightarrow^\alpha \sigma$.*

Proof. Suppose α is race-free yet contains an action b that is stuck. Consider the case where $b = rd(u, f)$ or $b = wr(u, f)$ and

$$W_f^b \npreceq C_u^b \qquad \text{Table 4, Rules 2, 3, and 5} \qquad (8)$$

Equation 8 requires that a prior write $a = wr(t, f)$ set W_f, and that $s^a = true$ or the rule for a was Rule 4 in Table 4. Thus,

$$W_f^b = C_t^a[t]@t \qquad \text{Table 4, Rules 4-5}$$
$$\npreceq C_u^b \qquad \text{Equation 8} \qquad (9)$$

$$K_t^a[t] = C_t^a[t] \nleq C_u^b[t] = K_u^b[t] \qquad \text{Equation 9}$$

By Lemma 6 $a \overset{\text{HB}}{\not\to}_\alpha b$, and we have a contradiction. A similar argument holds if $b = wr(u, f)$ and $R_f^b \not\sqsubseteq C_u^b$. □

C. Handling Volatile Variables

This section details how PACER handles 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 [26]. 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 *and* FASTTRACK *handle volatile variables. GENERIC and FASTTRACK use the same synchronization metadata for each volatile field x that they use for other synchronization objects: a vector clock C_x. Algorithms 14 and 15 show how GENERIC and FASTTRACK handle 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 uses the same synchronization operations for each volatile variable x as for each lock: $clock_x$ and $vepoch_x$. PACER's vector clock join in Algorithm 11 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 volatile variables, shown in Algorithm 16. In non-sampling periods, versions indicate if $clock_x \sqsubseteq clock_t$; if so, the join simply becomes a *shallow copy* from $clock_t$ to $clock_x$. The behavior is the same as at a lock release, i.e., the analysis copies the thread's clock to the volatile's clock.

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] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196, 2009.

[3] 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.

[4] 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.

[5] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[6] 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.

[7] 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.

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, volatiles [PACER]: $C_x \leftarrow C_x \sqcup C_t$

```
Let $v@u = \textbf{vepoch}(x)$
$\textit{fastpath} \leftarrow \textbf{false}$
if not $\textit{sampling}$ then
    if $(v@u \neq \textbf{null} \wedge v \leq \textit{ver}_t[u])$ then        {Check version}
        $\textit{fastpath} \leftarrow \textbf{true}$
    else if $\textit{clock}_x \sqsubseteq \textit{clock}_t$ then
        $\textit{fastpath} \leftarrow \textbf{true}$
    end if
end if
if $\textit{fastpath}$ then
    $\textbf{setShared}(\textit{clock}_t, \textbf{true})$
    $\textit{clock}_x \leftarrow_{\textit{shallow}} \textit{clock}_t$
    $\textit{vepoch}_x \leftarrow \textbf{vepoch}(t)$
else
    if $\textbf{isShared}(\textit{clock}_x)$ then
        $\textit{clock}_x \leftarrow \textbf{clone}(\textit{clock}_x)$
        $\textbf{setShared}(\textit{clock}_x, \textbf{false})$
    end if
    $\textit{clock}_x \leftarrow \textit{clock}_x \sqcup \textit{clock}_t$
    $\textit{vepoch}_x \leftarrow \textbf{null}$        {Cannot assign $\textbf{vepoch}(t)$}
end if
```

[8] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional Detection of Data Races. Extended technical report.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] C. Flanagan and S. N. Freund. Adversarial Memory For Detecting Destructive Races. In *ACM Conference on Programming Language Design and Implementation*, 2010.

[16] 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.

[17] T. Harris, S. Tomic, A. Cristal, and O. Unsal. Dynamic Filtering: Multi-Purpose Architecture Support for Language Runtime Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 39–52, 2010.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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.

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

[29] 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.

[30] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *ACM Conference on Programming Language Design and Implementation*, pages 22–31, 2007.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] A. Thakur, R. Sen, B. Liblit, and S. Lu. Cooperative Crug Isolation. In *International Workshop on Dynamic Analysis*, pages 35–41, 2009.

[37] 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.

[38] 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.

[39] 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.