MaTSa: Race Detection in Managed Languages
FORTH-TR-492 November 2024

ALEXANDROS EMMANOUIL ANTONAKAKIS, ICS-FORTH & University of Crete, Greece
IACOVOS G. KOLOKASIS, ICS-FORTH & University of Crete, Greece
FOIVOS ZAKKAK, Red Hat, Ireland
ANGELOS BILAS, ICS-FORTH & University of Crete, Greece
POLYVIOS PRATIKAKIS, ICS-FORTH & University of Crete, Greece

Parallel programs are prone to data races, which are concurrency bugs that are difficult to track and reproduce. Various attempts have been made to create or incorporate tools that aim to dynamically detect data races in Java, but most rely on external race detectors that: a) miss some of the nuances in the Java Memory Model (JMM) [11], b) are too slow and complicated to be used in complex real-world applications, or c) produce a lot of false positive reports. In this paper, we present MaTSa, a tool built within OpenJDK, that aims to dynamically detect data races and offer informative pointers to the origin of the race. We evaluate MaTSa and detect several races in the DaCapo benchmark suite and the Quarkus framework, many of which have been reported and resulted in upstream fixes. We compare MaTSa to Java TSan, the only current state-of-the-art dynamic race detector that works on recent OpenJDK versions. We analyze issues with false positives and false negatives for both tools and explain the design decisions causing them. We found MaTSa to be 22% faster on average, and use 52% less memory while detecting several races not found by Java TSan.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: data race, happens-before, dynamic race detection, instrumentation, MaTSa, multi-threaded programming, FastTrack, OpenJDK

ACM Reference Format:

Alexandros Emmanouil Antonakakis, Iacovos G. Kolokasis, Foivos Zakkak, Angelos Bilas, and Polyvios Pratikakis. 2024. MaTSa: Race Detection in Managed Languages FORTH-TR-492 November 2024. 1, 1 (December 2024), 20 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A **data race** occurs when two or more threads access the **same memory location** and at least **one of the accesses is a write**. Data race bugs are notoriously difficult to track and debug due to their intermittent manifestation, non-deterministic behavior, lack of reproducibility, complexity of concurrent execution, and subtle symptoms. The timing-dependent nature of these issues makes them hard to reproduce consistently, and the unpredictable final state of shared memory further complicates the process of identifying and resolving the underlying problems.

There are two major approaches when trying to detect data races, Lockset [18] and Happens-Before [9] relation checkers. Each technique has been used in both static and dynamic analysis

Authors' addresses: Alexandros Emmanouil Antonakakis, ICS-FORTH & University of Crete, Heraklion, Greece, aantonak@ics.forth.gr; Iacovos G. Kolokasis, ICS-FORTH & University of Crete, Heraklion, Greece, kolokasis@ics.forth.gr; Foivos Zakkak, Red Hat, Cork, Ireland, fzakkak@redhat.com; Angelos Bilas, ICS-FORTH & University of Crete, Heraklion, Greece, bilas@ics.forth.gr; Polyvios Pratikakis, ICS-FORTH & University of Crete, Heraklion, Greece, polyvios@ics.forth.gr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2024/12-ART
https://doi.org/XXXXXXX.XXXXXXX

tools. Lockset associates each shared memory location with a set of locks that must be held by each thread when accessing it. This technique infers the guarded-by relationship and is best for parallel programs using mutual exclusion locks as the sole synchronization mechanism, and can be modeled both dynamically, as e.g., in Eraser[18], or statically as e.g., in Locksmith [17]. Happens-Before algorithms, on the other hand, work by inferring the happens-before relation between memory accesses, that is, accesses that may not happen in parallel, due to some mechanism that enforces ordering. This approach has the advantage of being able to model multiple synchronization primitives and implementations, as long as there is a way to associate a happens-before relation with the synchronization mechanism.

The main state-of-the-art tool that uses the Happens-Before algorithm and is widely used in production, is ThreadSanitizer or TSan [3, 19, 20]. TSan is part of the LLVM project and is a dynamic race detection tool for C and C++. Currently, there are active attempts to incorporate TSan in languages like Java [16] and Go [4, 10]. These implementations combine compiling with LLVM to instrument the VM itself, and manual insertion of callbacks to the TSan runtime library for interpreted bytecode.

Mapping a Happens-Before dynamic analysis implementation to the Java language poses the following challenges. First, the Java Memory Model (JMM) [11] requires certain restrictions in parallelism, which in effect introduce happens-before relations without explicit synchronization. Most happens-before analyses link the inferred relation to synchronization primitives, and thus would not be able to model the JMM. Second, the JMM allows racy accesses to occur, and attributes semantics to such programs, which in turn allows for custom implementations of synchronization primitives and lock-free data structures. Modeling this in Happens-Before analysis poses an additional challenge: a precise race detector would then either face the impossible task to detect (and verify) custom synchronization implementations, or at least provide an easy way for application and library developers to annotate their code for use with the detector. Third, the JMM specifies constraints that only allow racy accesses to occur in restricted cases. Taking advantage of these restrictions can greatly improve performance. Finally, managed languages such as Java use Garbage Collection to manage the heap, resulting in objects changing their location in memory throughout the application execution. Thus, a dynamic analysis must handle accesses *per object* instead of *per address*.

In order to address these challenges, Java TSan, the current state-of-the-art dynamic race detector for Java, makes certain design choices that affect both performance and precision. For instance, in order to leverage the LLVM/TSan implementation, Java TSan is forced to increase the cost of GC, which now must track object migration and adjust all information about concurrent accesses accordingly. Also, Java-specific parallelism primitives such as parallel `volatile` accesses, custom library locks, or `CompareAndSet`, can be used to implement custom synchronization. To handle these without producing false positive warnings, Java TSan introduces happens-before edges in all such cases, even when e.g., a volatile access is not used to synchronize the rest of the program. This approximation will hide many false positives, but will also hide true races from being reported. These, and more, design decisions result in trade-offs between performance, memory consumption, and precision with respect to false positives and false negatives.

This paper aims to study and address these trade-offs, by proposing MaTSa: MAnaged Thread SAnitizer. MaTSa uses **FastTrack** [6], a happens-before data race detection algorithm that uses vector clocks as a way of establishing happens-before relations. FastTrack is similar to Java TSan. We designed and implemented MaTSa to explore the possible design choices involved when applying FastTrack to the JMM. We evaluated it on a wide set of benchmarks and compare the impact of each design difference with Java TSan, in terms of precision and performance. We found that our tool incurs up to 91% less runtime overhead while consuming up to 90% less memory, compared to

Java TSan. This performance difference enables us to apply MaTSa on Quarkus, a large production system, and find real races that were reported and fixed. We also demonstrate cases where each tool may produce false positives and false negatives, and discuss occurrences of these issues in real software.

Overall, this paper makes the following novel contributions:

- We explore the design space and discuss the trade-offs introduced when mapping the FastTrack algorithm on the Java Memory Model; we implement our design in MaTSa, a dynamic race detector implemented in OpenJDK.
- We compare MaTSa to Java TSan, quantitatively and qualitatively. We evaluate both tools and compare them in terms of performance, memory consumption, and precision, on a wide set of benchmarks. We analyze the results and evaluate the trade-offs between performance, false-positives and false-negatives, for each design.

The rest of this paper is organized as follows. Section 2, presents the race detection algorithm. Section 3 presents the design principles of the tool as well as some parts of the Java language specification we have to pay attention to. Finally, section 5 presents the evaluation of our work.

2 ALGORITHM

MaTSa, similarly to FastTrack, uses a vector clock per thread and per synchronization object, to keep track of ordering constraints during execution, and thus establish happens-before relations between memory accesses.

2.1 Happens-Before

Happens-before relations describe the ordering constraints between certain operations during program execution. The key idea is that if event A happens-before event B, then the effects of A are guaranteed to be visible to B, and B cannot affect the execution of A. This ordering is crucial for ensuring that concurrent operations do not interfere with each other in unexpected ways, leading to data races. In our case, a happens-before relation is established if event A happens before B and there is a mechanism that ensures that the effects of event A are visible to event B. Such mechanisms might be locks, synchronized blocks or methods, thread join, or program ordering.

2.2 Vector Clocks

In the context of MaTSa's algorithm, a vector clock is an array of timestamps (epochs) where each element represents the logical time of a thread as observed by the current thread. For a system with n threads, each vector clock is an array of n integers. Each thread maintains its vector clock. We call $\mathrm{VC}[t_i]$ the vector clock of thread t_i.

$$\mathrm{VC}[t_i] = [e_1, e_2, \ldots, e_n]$$

Where e_j represents the latest known timestamp of thread t_j as observed by thread t_i. For convenience, each thread starts with a clock value of 1. In essence, a vector clock is a mechanism for the current thread, to keep track of what has been executed in other threads.

2.3 Forming Happens-Before Relations

MaTSa uses the vector clocks in threads and synchronization objects to keep track of ordering constraints. In specific events that create ordering constraints between threads, MaTSa updates the vector clocks involved, to keep track of what each thread "knows" about the execution of other threads:

- On **lock events**, the current thread's vector clock is updated by taking the maximum of the current thread's vector clock and the lock's vector clock, and then storing the result in the current thread's vector clock. This ensures that the thread's vector clock is synchronized with the lock's vector clock, reflecting any happens-before relationships established by the lock.
- On **unlock events**, the lock's vector clock is updated by taking the maximum of the current thread's vector clock and the lock's vector clock. Finally, the current thread clock is incremented. This way, the lock has stored information about what the current thread knows has happened-before this event in every thread.
- The **Wait-Notify** pattern requires special handling as `wait` acts both like an unlock and a lock. On `wait` invocation we perform a vector clock release operation similar to an `unlock` event, as the thread invoking the method releases the lock to enter the wait state. On `wait` return we perform a vector clock acquire operation similar to a `lock` event, as the thread invoking the method needs to acquire the corresponding lock before continuing.
- On **thread start events** we copy the vector clock of the parent thread (the thread that is starting it), to the starting thread's vector clock. Since it is impossible for a thread to participate in a data race prior to its starting.
- On **join events**, as after a thread has finished (exited), and has joined with a thread, accesses after the join are impossible to cause a race condition. When a thread is exiting, we store its vector clock and we later transfer the vector clock from the thread object to the current thread's vector clock.

2.4 Access Metadata & Race Check

On each memory access event i, MaTSa stores metadata $c_i = (t_i, e_i, k_i)$ for that access. At the minimum, it has to store the **thread id** t_i of the accessing thread, the **epoch** e_i of the accessing thread at the time of access, and if the access kind k_i was a **read or a write**. On each access, this information is compared to information from previous accesses to that location. If a previous access j does not have a happens-before relation to this access, they could have happened in parallel, and the pair forms a possible race. Specifically, for a race to be possible, the following criteria must be met:

- Different threads accessing the same memory location: $t_i \neq t_j$.
- At least one of the accesses is a write: $k_i = W \vee k_j = W$.
- The current thread t_i has not yet observed in its vector clock $\mathrm{VC}[t_i]$ the epoch e_j of thread t_j that executed the previous access, *i.e.*, there is no happens-before edge between the two accesses: $\mathrm{VC}[t_i][j] < e_j$.

2.5 Happens-Before Example

Consider the example in Figure 1, in which two threads lock and unlock a shared lock. In the example, we assume Thread 0 gets the lock first, while Thread 1 blocks and waits for the lock. On every **lock**, the lock's vector clock is acquired or *merged* with the thread vector clock, while on every **unlock**, the thread's vector clock is released or *merged* with the lock's vector clock. By merge, we mean:

$$merge(v, v') = [x_i | x_i = max(v_i, v'_i)]$$

That is, the current thread will update its own vector clock to the maximum epoch for each thread, that either the current thread or the lock has witnessed. Intuitively, the current thread will know that all other threads have executed *at least* until the epoch in the lock object's vector clock. Thus,

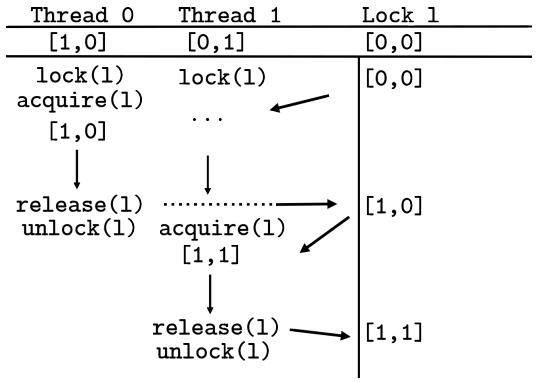


Fig. 1. Vector clock transfer between 2 threads via a shared lock.

accesses by the other threads will have *happened-before* the accesses about to be executed in this thread.

In the example, we see that Thread 0 acquires the lock first, resulting in the lock object's vector clock $[0, 0]$ being merged into the vector clock of Thread 0 $[1, 0]$. Upon release, the vector clock of Thread 0 $[1, 0]$ gets merged into the vector clock of Lock l. Then, the vector clock of Thread 0 will increment to $[2, 0]$ on unlock; this is not shown in the Figure, for simplicity.

Thread 1 is blocked during this process, as it waits to acquire Lock l. When Thread 0 releases Lock l, Thread 1 can then wake up and progress, by acquiring the lock. On acquire, Thread 1 then merges the vector clock of Lock l $[1, 0]$ into its own, which advances to $[1, 1]$. This way, Thread 1 knows that Thread 0 has executed at least until its epoch 1, and every access until this point has happened before Thread 1 execution from now on. Upon release, Thread 1 merges its vector clock into the vector clock of Lock l, as above. This way, subsequent acquires of Lock l, will "inform" the acquiring thread that actions up until epoch 1 for Thread 0 and epoch 1 for Thread 1, have happened before that acquire.

3 DESIGN & IMPLEMENTATION

3.1 Shadow Memory

As mentioned above, MaTSa needs to store various metadata for each memory access to accurately detect data races. Similarly to TSan, we use **Shadow Memory**, which reserves a part of the application's virtual memory.

In shadow memory, we store shadow cells. Each shadow cell is 8 bytes of useful information about the access. We dedicate 4 shadow cells (32 bytes) for every 8 bytes of application memory. Generally, the shadow memory is 4x larger than the application memory. Note that this is different and simpler than TSan's shadow memory, as we are able to take advantage of the Java Memory Model. Namely, Java does not allow concurrent accesses outside the heap, which is of known maximum size[1]. Moreover, concurrent updates may only happen on object fields and arrays. These constraints allow MaTSa to optimize memory consumption.

The mapping from application memory at an address x to shadow memory is very cheap and is performed as:

$$\text{heap}_{\text{offset}} = x - \text{heap}_{\text{base}}$$

[1]Unsafe accesses are outside the scope of this work as these are outside the scope of the JMM

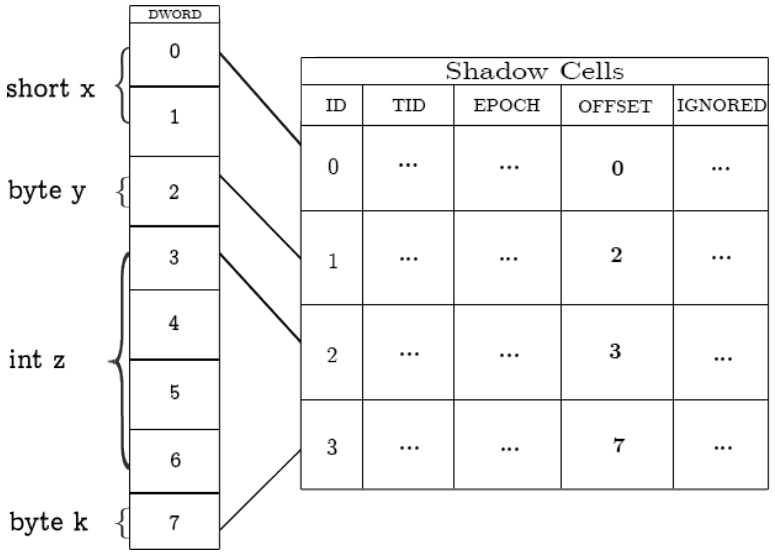


Fig. 2. Offset of fields of different sizes.

$$\text{shadow}_{\text{index}} = \text{heap}_{\text{offset}} \times 4$$
$$\text{shadow}_{\text{address}} = \text{shadow}_{\text{base}} + \text{shadow}_{\text{index}}$$

Where the Java heap starts at $\text{heap}_{\text{base}}$, and shadow memory starts at $\text{shadow}_{\text{base}}$

3.2 Shadow cells

Shadow cells contain useful information about each memory access and have a size of 64 bits. We dedicate 4 shadow cells for each 8-byte word in the heap. Each shadow cell has the fields mentioned in section 2.4 and one extra:

- $\text{Epoch}_{\text{thread}}$ (42 bits): The clock value of the accessing thread at the time of the access.
- TID (17 bits): The thread ID of the accessing thread.
- Offset (3 bits): Because of the division of the application memory by 8, we observe that addresses that may refer to smaller accesses (1,2, or 4 bytes), map to the same shadow address. To resolve that, we dedicate 3 bits for the offset within the 8 bytes of the application. This field is used as shown in Fig. 2, where fields of different sizes are mapped to the same DWord address. This means MaTSa will not produce a warning for concurrent field accesses to different fields, even if these are located in the same DWord. However, this may reduce the probability a race will be detected, as the given DWord still maps to only four cells, now used to track accesses to multiple fields, not just a single word. Thus, accesses to the other fields have a higher probability to overwrite an offending access to a racy field before the race is observed.
- IsWrite (1 bit): Used to mark if the access was a read or write to memory.
- IsIgnored (1 bit): In case of a reported or ignored race, this bit would be set to indicate that there is no need to test further or report races for this shadow block. In order of this mechanism to ignore an access, it has to refer to the exact same offset within the DWord.

Due to Java being a garbage-collected language, objects may be moved, or freed, and the address they previously lived on, can be reused by another object. It is possible then, for different objects to map to the same shadow memory at different times. To handle that we should check every object

that gets freed/moved and clear/update the corresponding shadow region. This, however, will add significant overhead. To avoid expensive shadow memory moves after a GC, we compare two alternative solutions. One involves using an additional field in each shadow cell that will act as GC epoch counter, so that we ignore accesses with different GC counters. The other solution is to reset the entire shadow memory after each GC cycle. This is faster in practice, as we are not zeroing out the entire shadow memory but instead unmap and remap it, essentially letting the OS do the rest lazily. Out of the two, we measured up to 5% improvement in runtime with the second approach and used that instead. As these solutions may hide race warnings involving accesses separated by a GC, we measure the incurred precision loss by comparing to executions with zero garbage collections. Both approaches can reduce accuracy up to 7% with average loss of 3%, as seen on table 3 but are significantly faster and enable support for every garbage collector with contiguous heaps without any modification in the collector itself.

We assign bits to fields under the following intuition: We want to keep each cell as small as possible without sacrificing precision. We opted to go for 64 bits which can accommodate a great number of threads and a wide range of epochs. The $\text{Epoch}_{\text{thread}}$ field is essentially a counter for each thread that increments on vector clock release events, including unlock, thread join, and class initialization. As such, it is possible to increment quite fast and thus requires to be big enough to avoid frequent overflows. Additionally, the *TID* field by default can support up to 131k threads, we opted to go for that number to also support virtual threads in future builds. Virtual threads are threads that are not tied to a specific os thread. It is also worth mentioning that these fields are configurable at compile time of the JVM, and can be tailored to the needs of a specific application.

3.3 Synchronization Objects

As stated above, each synchronization object has its vector clock. Due to Java being a garbage-collected language, we cannot apply the same shadow mapping principle used in memory accesses to synchronization objects. A moved lock object might map to an address that already has an active vector clock, leading to false reports or even crashes.

An efficient solution, as seen in previous work [8], is to include a pointer inside each java object header (*ordinary object pointer*) which is used to allocate memory for a vector clock. This way, even if an object is moved, it would not affect the corresponding vector clock. On each lock or unlock operation, if not already initialized, we initialize this pointer with a new vector clock. Figure 3 depicts the resulting object layout.

The exchange of vector clocks happens right after the lock operation has succeeded and also just before the unlock operation. We do that to avoid synchronization in the vector clock array. This way, all accesses to the vector clock will be guarded by the actual application lock.

3.4 Runtime

We extended the JVM template interpreter, by instrumenting every load and store instruction for fields and arrays. We altered the CPU-specific templates for the corresponding bytecodes to generate a shadow cell and perform the race check. Currently, we only support the bytecode templates for x86. Requiring the user to run dynamic race detection in interpretation mode increases the overhead for both MaTSa and Java TSan, the current state-of-the-art dynamic race detector.

We altered the bytecode implementation for object locks, to allocate vector clocks, if needed, and perform the corresponding merges. We similarly altered the library implementation for `java.util.concurrent` locks, so that it calls into MaTSa at every lock or unlock operation. We similarly added MaTSa calls for most synchronization primitives in `java.util.concurent`, such as re-entrant lock, semaphore, thread pool, stamped lock, and countdownlatch. Additionally, we enabled support for JNI locks, similarly by adding MaTSa calls on each lock an unlock.

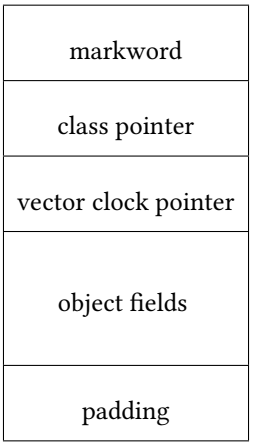


Fig. 3. Object layout.

3.5 Previous Stack Traces

As mentioned above, a race condition occurs when two or more threads access the same memory location, and at least one of the accesses is a write. We keep information about accesses to different memory locations, in the shadow memory. When a race is detected, it is crucial to be able to print stack traces, for both accesses involved. Printing the stack trace for the current access is easy, as we report races during runtime and the current stack trace is available to use. However, past stack traces are difficult to formulate. That stack is likely to have changed after that access, or that thread could have terminated by the time the race is detected.

To generate the stack trace for the previous access, we need to store information about the function entry and exit events per thread, relative to the memory accesses, in a trace. When a race is detected, the offending memory access can then be located in the trace, and the function entry/exit events can be followed to recreate the stack trace around that access.

We record function enter and exit events and store them in event buffers. We keep multiple event buffers, stored in a circular buffer called the History Buffer. Each event contains the address of the function (48 bits) and the BCI of its caller (16bits). A **null** address indicates a function exit event. By default we store 16 event buffers, with a capacity of 65k events each.

In order to have information about the stack of each memory access, we use a second shadow memory region, *Shadow History*. This region stores information regarding the position in the event buffer at the time of the access. We store 64 bits of information per access, consisting of the following:

- bci (*bytecode index*) (16 bits)
- ring_idx (*index in the circular buffer*) (16 bits)
- history_idx (*index in the event buffer*) (16 bits)
- history_epoch (*crucial to detect if the event buffer has been overwritten*)

Every time a race is detected, it involves a shadow cell for the race thread, and a shadow cell for a previous access to the same memory location by another (previous) thread. To recreate the stack for the previous access, we look up its shadow history cell (*we use the same principle to map a shadow history cell with a shadow cell so fetching is performed in constant time*). We then traverse the event buffer specified by **ring_idx** up to **history_idx** to formulate a stack trace that leads up

to the access event by pushing function entry events and popping function exit events. Using the `bci`, we calculate line numbers for each function enter event in the trace.

If the event buffer reaches maximum capacity, we transition to the next event buffer in the ring buffer. After the transition, the new event buffers epoch is incremented to invalidate older traces. In addition, we copy the current stack into this new event buffer to maintain as much information as possible. In order to leave some extra room in the event buffer, we maintain a separate pointer that points to a copy of the stack at the time of the transition.

3.6 Report Map

The shadow memory mappings used in TSan and MaTSa allow race detection per memory location. However, an error in the code may cause a multitude of data races at run time. Reporting the same line of code repeatedly only because it involves many memory locations in an execution, may be quite confusing for the developer. To produce clear, concise, and useful race warnings, MaTSa summarizes the observed races into a set of warnings that avoid repetition. This reduction is additional to the marking of shadow cells using the **isIgnored** field to avoid reporting the same location twice. Note that we cannot use this field to avoid reporting the same code location twice, as each memory address will correspond to a different **isIgnored** flag.

To detect when multiple races involve the same location in the code, we use a hash map that stores the `bcp`. The `bcp` is the bytecode pointer and is guaranteed to be unique. On each report, we add the `bcp` of the current code location and before each report, we check if we have an instance of the current `bcp` in the map. If we do, then we have already reported a race on this code location and thus we can ignore it. This policy may result in hiding some of the call paths leading to the offending memory access; however, usually these are code paths that do not add new information pertaining to the race. For example, it is, more often than not, the case that a data race *e.g.,* in a data structure can be reached from hundreds of code points where the data structure is used. Listing all possible code paths in such cases adds little towards understanding and fixing the race while complicating the warnings and increasing memory overheads unnecessarily.

3.7 Custom Synchronization Mechanisms

While we do support widely used synchronization mechanisms, such as `java.util.concurrent` locks, synchronized blocks, and JNI locks we cannot track user-defined synchronization. This can result in false-positive race reports. To tackle that, we have included functions that may be used to notify MaTSa about a lock or unlock event, to avoid false-positive reports:

```
System.MaTSaLock(lockobject)
System.MaTSaUnlock(lockobject)
```

This is the mechanism used to support `java.util.concurrent` locks.An example of a very naive spinlock could have the form shown in Figure 4.

4 COMPARISON AGAINST STATE-OF-THE-ART

4.1 Java TSan

Volatile: According to the Java Memory Model [11], volatile fields are guaranteed to make any modification on them visible to other threads that may access them. MaTSa ignores accesses to volatile fields as by definition they can not be racy. On the other hand, Java TSan performs a vector clock release before a write to a volatile variable and a vector clock acquire after a read from a volatile variable. While this may be correct for custom synchronization implementations, it is also possible for a race to hide behind this pseudo-happens-before edge. Figure 5 shows an example of a volatile access used as a custom synchronization mechanism, in effect creating a happens-before

```
public class Spinlock {
    private AtomicBoolean lock = new AtomicBoolean(false);

    public void lock() {
        while (!lock.compareAndSet(false, true))
        { /* Spin */ }
        /* notify MaTSa
          right after lock acquisition
        */
        System.MaTSaLock(lock);
    }

    public void unlock() {
        /* notify MaTSa
           just before lock release
        */
        System.MaTSaUnlock(lock);
        lock.set(false);
    }
}
```

Fig. 4. A Custom Spinlock Implementation

edge between the write to the volatile field in Thread **t2** and the spinning loop in Thread **t1**. To avoid reporting this as a race, Java TSan introduces a happens before between the write to the volatile variable and the read in the spinning loop. On the other hand, Figure 6 shows an example in which a true race on shared field **y** can hide due to the happens-before edges drawn by volatile accesses in Java TSan. Specifically, while the read from and write to volatile field **x** cannot race, the concurrent accesses to static field **y** are not protected by any synchronization. Modeling the volatile accesses to **x** as acquire and release for the read and write, respectively, will create a bogus happens-before edge between the write to **x** in Thread **t1** and the read from **x** in Thread **t2**. This bogus synchronization will hide a true warning on field **y**. We ran this example 100 times, of which MaTSa reported **100** races while Java TSan **0**. Overall, MaTSa will report the true race at the cost of also reporting the false positive, while Java TSan will avoid the false positive, at the cost of not reporting the true race.

Static Class Initializers: According to the JMM, static class initializers form a happens-before edge between the initialization of a static field and its accesses. Other threads accessing a static field that is to be initialized by a static class initializer will wait until the initializer has finished, hence forming this happens-before edge. This is a JMM requirement, thus it has to be implemented within the JVM, as there is no Java lock, object lock or generated bytecode that can be used to force this wait. Essentially, there is a lock within the JVM, not visible to the Java application, which introduces a happens-before edge between static class initializers and static field accesses thereafter [2]. To track that, MaTSa allocates a separate, internal vector clock modeling this internal JVM lock. MaTSa then performs a vector clock acquire on every static field access, to acquire the vector clock state after the initializer has finished.

[2] OOP as class initialization lock

```java
public class VolatileFalsePositive {
    public static volatile int x = 15;
    public static int y;
    public static void main(String[] args)
    {
        Thread t1 = new Thread(
            () - > {
                int tmp; // int tmp = x;
                while ((tmp = x) != 42);
                y = tmp + 1;
                x = 2;
            });

        Thread t2 = new Thread(
            () - > {
                int tmp = x;
                y = tmp + 1;
                x = 42;
            });
    }
}
```

Fig. 5. Non-Race between volatile accesses

```java
public class VolatileFalseNegative {
    public static volatile int x = 15;
    public static int y;
    public static void main(String[] args)
    {
        Thread t1 = new Thread(
            () -> {
                int tmp = x;
                y = tmp + 1;
                x = 2;
            });

        Thread t2 = new Thread(
            () -> {
                int tmp = x;
                y = tmp + 1;
                x = 2;
            });
    }
}
```

Fig. 6. Race between volatile accesses

On the other hand, Java TSan performs similar vector clock operations but using the vector clock of the class object. This conflation of the JVM internal lock per static initializer, with the object lock of the class object, may introduce bogus happens-before edges, as the latter is also used across every **static synchronized** method. Figure 7 shows a case where using the class object lock to model class initializer synchronization may hide a true race in Java TSan. Specifically, method **notsync** needs to wait for the class initializer according to the JMM, as it contains an access to the static field. At the same time, method **sync** will write to the class object vector clock at exit. Reusing the same vector clock for both happens-before edges, will falsely transfer the vector clock of thread **t1** to thread **t2** due to the access to static field **counter**. This imprecision is caused by using a single vector clock to model two actual locks; the JVM lock implementing the JMM requirement on static initializers, and the class object lock. Unfortunately, this hides the true race between the accesses to **counter** in the two static methods. Indeed, we repeated this test **100** times with both tools, of which, MaTSa reported **100** races while Java TSan reported **0**.

Library and Custom Synchronization. By examining the OpenJDK source code, it is clear that all java.util.concurrent locks are implemented by using some atomic CPU instruction (with CompareAndSwap being the most favorable) through Java's Unsafe module. Java TSan handles such operations by releasing and acquiring the vector clock of Unsafe class. While this is a clean and easy solution as it takes little changes to source code and minimal knowledge of the java.util.concurrent library, it can again form bogus happens-before edges in cases where a simple CompareAndSwap operation is performed, thus hiding actual races in some cases. To improve precision, we have studied every lock in the library and added callbacks to MaTSa, which significantly improves accuracy. Additionally, this excludes the probability of races within MaTSa itself, which protects a vector clock by acquiring it after the corresponding lock has been successfully acquired and

```
public class ClassInitFalseNegative {
    private static int counter = 1;

    private static synchronized void sync() { counter = 3; }
    private static void notsync() { counter++; }

    public static void main(String[] args) throws Exception {
        Thread thread1 = new Thread(() -> { sync(); });
        Thread thread2 = new Thread(() -> { notsync(); });
    }
}
```

Fig. 7. Static Class Initializer False Negative

releasing it just before the lock is released[3]. One other aspect that is handled differently is **thread join**. Java TSan does not monitor the use of join and creates a happens-before edge between parent and child thread after the child thread has stopped, which by itself defeats the whole purpose of thread join and can produce false negatives. In order to conform to thread join, we have added a callback to MaTSa when the child thread is stopping, this is to save its vector clock into the thread object. Additionally, we have added a callback to MaTSa just before thread join returns to the caller, this transfers the vector clock of the child thread to the parent thread, safely forming a happens-before edge.

Grouping and Reporting Warnings. Java TSan may report a race to the same code location more than once if the stack trace that led to the access is different from those previously reported. While this is useful and can allow users to detect multiple races by just running the tool once, it makes the output too long and too difficult to read and understand. MaTSa reports each race once per code location, by grouping them by the BCP (ByteCode Pointer). Because of this approach, MaTSa may report significantly fewer warnings than Java TSan, which is visible in evaluation 5, corresponding to the same races.

4.2 Race Detection for Java

There is a lot of past work on race detection for Java. Static analysis tools such as Meta's Infer [2] and others [12, 13] use static analysis. Dynamic analyses include DrFinder [14] or hybrid systems [15]. These have limited scalability and do not scale to very large programs and threads, or do not support recent versions of Java, custom synchronization and libraries.

As of this moment, apart from Java TSan, RoadRunner [7] is the most relevant dynamic analysis for race detection to MaTSa. RoadRunner is written in Java, and supports Java 8 or 7. Instead of adding callbacks in the VM code, RoadRunner relies on a Java agent and bytecode rewriting, essentially adding callbacks dynamically and leveraging JIT compilation. While this approach may prove faster, it requires significantly more effort from the user to build and requires understanding of its core if extra synchronization must be tracked. Additionally, the reports provided were less informative, as stack traces included the tool's instrumented functions and did not include the trace of the previous racy access. We attempted to build RoadRunner and compare it against it. However,

[3]MaTSa uses this trick to avoid requiring synchronization for its own vector clocks, by protecting them from concurrent access. This only works when acquire and release operations are used within critical sections protected by the corresponding lock, however, and will not work for acquire and release operations inserted to model happens-before edges created by other types of synchronization, such as CompareAndSwap, volatile accesses, or atomic increment.

Benchmark	MaTSa			Java TSan			Runtime	Slowdown
	MaTSa	Xint v17	Slowdown	Java TSan	Xint v21	Slowdown	Reduction	Reduction
avrora	1m26s	0m27s	3.19x	2m12s	0m25s	5.28x	34.84%	39.58%
batik	1m19s	0m22s	3.59x	1m18s	0m19s	4.11x	-1.28%	12.65%
biojava	28m21s	11m21s	2.50x	36m30s	4m50s	7.55x	22.32%	88.79%
eclipse	15m58s	4m7s	3.88x	15m22s	3m26s	4.48x	-3.90%	13.92%
fop	0m55s	0m18s	3.06x	1m6s	0m17s	3.88x	16.66%	81.63%
graphchi	16m15s	4m51s	3.45x	16m36s	3m14s	5.13x	2.10%	32.74%
jme	0m40s	0m17s	2.35x	0m40s	0m13s	3.08x	0.00%	23.70%
jython	7m44s	3m24s	2.27x	12m0s	2m0s	6.00x	35.50%	62.16%
luindex	11m45s	3m48s	3.00x	10m7s	2m49s	3.59x	-16.14%	16.43%
lusearch	19m14s	6m8s	2.84x	34m47s	7m0s	4.97x	47.70%	42.85%
pmd	3m05s	0m44s	3.30x	27m13s	0m27s	60.48x	88.67%	94.54%
sunflow	14m44s	2m25s	5.79x	48m9s	2m7s	22.75x	69.40%	74.54%
xalan	1m53s	0m28s	3.64x	9m22s	0m29s	19.38x	79.89%	81.21%
zxing	3m42s	0m36s	6.56x	2m12s	0m31s	4.26x	-68.18%	-53.99%

Table 1. Performance Results

as it can only work for Java 7 or 8, it failed to build for newer versions of Java (17+) due to the wide use of deprecated APIs.

5 EVALUATION

We evaluate MaTSa in terms of performance and accuracy. We moreover compare it qualitatively with Java TSan, the current state-of-the-art dynamic race detector for Java. We use the DaCapo Benchmark suite [1], omitting applications that do not run successfully with OpenJDK v17 and v21. We used the G1 garbage collector in all experiments. We measure total runtime, peak memory consumption, and reported races.

We repeated each benchmark 5 times and calculated the average runtime. We report the maximum number of race warnings among the 5 runs. We ran these benchmarks on a system with the following specifications:

- Intel core i7-10700k @ 4.7GHz, 8 cores, 16 threads
- 32GB DDR4 RAM @ 2133MHz
- 1TB NVMe
- Ubuntu 24.04

Note that MaTSa is built by extending OpenJDK 17, while Java TSan extends OpenJDK 21. We chose to implement MaTSa for OpenJDK 17 as it is a widely used version of Java (albeit older), and currently the latest supported version for many production applications not yet ported to OpenJDK 21. Thus, there may be a performance difference due to the different JVM implementations. To factor out these differences and compare the overhead incurred by each tool, we also show the performance of each benchmark when run on the vanilla version of each JVM.

5.1 Performance Evaluation

Table 1 presents the average runtime for each benchmark named in the first column. Columns 2 and 3 show average execution times for each benchmark run in MaTSa and the vanilla OpenJDK17 in interpreter mode, respectively. Column 4 shows the slowdown factor, *i.e.*, the overhead incurred by MaTSa instrumentation, checking, and output. Columns 5-7 show the same measurements for TSan and OpenJDK21 in interpreter mode, respectively. This is a fair comparison as both tools only run in interpreter mode. Finally, the last two columns show the reduction in total average execution

Benchmark	MaTSa			Java TSan			Memory Reduction	Overhead Reduction
	MaTSa	Xint v17	Overhead	Java TSan	Xint v21	Overhead		
avrora	240.21 MB	88.68 MB	2.71x	1.13 GB	88.03 MB	12.84x	78.74%	78.89%
batik	1.56 GB	436.84 MB	3.57x	2.77 GB	420.86 MB	6.58x	43.68%	45.74%
biojava	1.86 GB	401.07 MB	4.64x	6.63 GB	395.73 MB	16.76x	71.94%	72.31%
eclipse	6.43 GB	676.14 MB	9.51x	9.04 GB	664.78 MB	13.60x	28.87%	67.05%
fop	1.48 GB	209.91 MB	7.05x	2.08 GB	208.98 MB	9.94x	28.84%	29.07%
graphchi	2.66 GB	687.37 MB	3.87x	7.56 GB	1.18 GB	6.41x	64.81%	39.62%
jme	246.23 MB	220.28 MB	1.12x	1.34 GB	154.33 MB	8.68x	81.62%	87.09%
jython	7.70 GB	269.22 MB	28.61x	6.77 GB	362.87 MB	18.66x	-13.73%	-53.32%
luindex	616.80 MB	155.40 MB	3.97x	1.25 GB	167.82 MB	7.45x	50.65%	46.71%
lusearch	3.28 GB	287.20 MB	11.42x	5.22 GB	199.54 MB	26.16x	37.16%	56.34%
pmd	3.99 GB	1.32 GB	3.02x	15.06 GB	986.90 MB	15.26x	73.49%	80.20%
sunflow	802.23 MB	468.74 MB	1.71x	9.41 GB	189.53 MB	50.00x	91.23%	96.58%
xalan	3.18 GB	270.46 MB	11.76x	3.31 GB	196.52 MB	16.85x	3.92%	30.20%
zxing	764.26 MB	345.41 MB	2.21x	4.11 GB	324.23 MB	12.68x	81.40%	82.57%

Table 2. Memory Usage Comparison

time and the reduction in slowdown compared to the corresponding vanilla JVM. The last column is there as the difference in time may be also attributed to the performance difference between the interpreters of the two JVM versions. In most benchmarks, MaTSa outperforms Java TSan. For batik, graphchi, and luindex, MaTSa is somewhat slower, although this difference is because the OpenJDK 21 interpreter often runs faster than the OpenJDK17 interpreter in these benchmarks. For zxing, MaTSa performs significantly slower than Java TSan. We believe this is due to additional overhead caused by tracking `java.util.concurrent` locks in MaTSa; Java TSan does not support these locks without code annotations.

Table 2 compares peak memory usage per benchmark between the two tools. We used **GNU Time** to measure memory usage. The numbers shown represent the *Maximum resident set size*. As above, the first column shows the benchmarks. Columns 2 and 3 show peak memory usage for MaTSa and the vanilla OpenJDK 17 interpreter, respectively. Column 4 shows the overhead factor; the difference amounts to MaTSa shadow memory, metadata, vector clocks, and the difference in object header size. Similarly, columns 5-7 show the same measurements for Java TSan, OpenJDK 21 in interpreter mode, and their difference, respectively. Finally, the last column shows how much less peak memory was required by MaTSa compared to Java TSan to run each benchmark.

Note that in almost all experiments, the memory overhead differs from the expected 4x for the following reasons. First, we expect the shadow memory to be 4x the heap size, which is only a fraction of the total program memory. Second, MaTSa requires additional memory to store vector clocks and other metadata. Third, shadow memory pages are allocated on write, via page-faults, this might mean a large part of the allocated page, remains unused.

Table 3 shows how many races were reported by MaTSa and Java TSan. The first column shows the name of each benchmark application. The second column shows the number of race warnings produced by MaTSa. In order to evaluate how many potential races may be hidden due to our design decision to reset shadow memory after each GC, the third column shows the number of warnings produced by MaTSa when run without any GC. We achieve this by setting a large enough heap and selecting the epsilon Garbage Collector [5], which will never perform a GC. We manually analyzed the reported warnings for most applications in both cases, and concluded which of the reported races are true. Unfortunately, we were unable to analyze all of the warnings produced for the eclipse application, due to the complexity of the codebase when including library dependencies, custom synchronization mechanisms that we are not able to track and due to the

Benchmark	Warnings (grouped)			Races	
	MaTSa	MaTSa No GC	Java TSan	MaTSa	Java TSan
avrora	7	7	46(7)	7	7
batik	0	0	0	0	0
biojava	0	0	0	0	0
eclipse	143	143	134(36)	-	-
fop	0	0	0	0	0
graphchi	0	0	0	0	0
jme	0	0	0	0	0
jython	0	0	0	0	0
luindex	0	0	0	0	0
lusearch	18	19	11(10)	18	10
pmd	58	63	110(57)	58	57
sunflow	5	6	5(5)	5	5
xalan	37	37	28(10)	37	10
zxing	21	21	22(17)	21	17
dacapo-core	3	3	3	3	3

Table 3. Race Detection Comparison

fact that a lot of the races were not detected in the eclipse code but in external dependencies. For the remaining benchmarks, the two race detectors performed as follows. In avrora, MaTSa reports 7 warnings, all of which are true races, while Java TSan reports 46 warnings, which amount to the same 7 code locations when grouped accordingly. In batik, biojava, fop, graphchi, jme, jython, luindex, both tools do not detect any races. In lusearch, MaTSa produced 18 warnings, missing one warning due to GC, while Java TSan produced 11 warnings, amounting to 10 discrete code locations. We verified that all of the reported warnings amount to true races, albeit benign. Specifically, MaTSa detects races in `String.hashcode`, `Map.get`, `Map.getnode`, `enum.valueof`, and `NativeMethodAccessorImpl.invoke`. Java TSan does not report these as races because they are known to be benign in the Java standard library, by explicitly suppressing their warnings. Other than those, both tools report the same races. In pmd, MaTSa reports 58 warnings, of 63 races observed without GC, while Java TSan reports 110 warnings that can be grouped into 57 code locations. In short, both tools report the same races, other than a benign races in the standard library that is explicitly suppressed. The difference in warnings produced by MaTSa amounts to one true race that was not detected when resetting the shadow memory at every GC. That race was not detected by Java TSan either. In sunflow, both tools report the same races. When running MaTSa without GC, there is an additional warning that refers to library code. The additional warning when running without GC occurs in sunflow code and we verified amounts to a true race. Java TSan does not report that race either. In xalan, MaTSa generates 37 warnings, while Java TSan 28, which correspond to 10 pairs of racy accesses. We verified that all warnings are true races. MaTSa reports some known benign races in the Java library that are explicitly suppressed in Java TSan. In zxing, MaTSa generates 21 warnings, while Java TSan generates 22, amounting to 17 distinct pairs of racy accesses. We verified all warnings to be races. The difference is due to 4 benign races in `setEndIndex`, `setBeginIndex` and `buffer.position` (twice). These benign races are not reported by Java TSan due to a bogus happens-before relation generated due to the use of the volatile primitive (volatile read), which however does not implement mutual exclusion in this case. Finally, in dacapo-core, the driver application for the other benchmark applications, both tools

```
  private static Integer globalIdx = 0;
  private static int inc() {
    int rtn;
    synchronized (globalIdx) {
       /*
       globalIdx += stride means:
       globalIdx = new Integer(globalIdx + stride)
       */
       rtn = globalIdx += stride;
    }
    return rtn;
  }
```

Fig. 8. Race in DaCapo 23.11-chopin

report 3 warnings, all of which are true races. One of the three has since been fixed in the public repository for the suite.

Many of the differences in reported races between the two tools are due to the different approaches used to detect happens-before edges and to the different ways of suppressing previously reported races, as presented in Section 4.1. Regarding happens-before detection, MaTSa makes callbacks to our API after each `java.util.concurrent` lock operation, which currently includes reentrant locks, semaphores, stamped locks and countdownlatch, while Java TSan does not do that. Additionally, Java TSan assumes a happens-before edge between the read and write to a volatile variable, which can hide races. Finally, MaTSa follows a different approach to ignore detected races as stated in Section 3.6, while Java TSan compares the stack traces of the reported races and decides whether that exact trace the led to the race was previously reported. This means that MaTSa will group races which refer to a code line previously reported, while LLVM will report them separately.

5.2 Notable Detected Races

MaTSa has detected a race in DaCapo's core release version which has since been patched. The race, shown in Figure 8, was related to an Integer that was used as a synchronization object, being incremented inside the synchronized block. However, Integer objects in Java are immutable, meaning an increment effectively changed the object reference to point to a new Integer object, different from the one used to synchronize, which allowed other threads to access the same critical section. This issue was fixed in the public DaCapo repository, although the fix is still not in the stable release.

Additionally, we used MaTSa to check for data races in the current stable version of **Quarkus** (3.8). We detected a data race, shown in Figure 9, related to a null check on an object. If the object is null, a new object is created; however, there is no synchronization between the check and the write, meaning that another thread can see the old value of the object (null in our case) and create a new object as well. The bug has been submitted as an issue to Quarkus's repository along with other detected races.

6 CONCLUSION

We have developed MaTSa, a tool that aims to dynamically detect data races in Java applications. It uses FastTrack, a Happens-Before algorithm to detect Happens-Before edges between shared memory accesses. MaTSa takes advantage of Java object headers and the Java memory model, to be as fast and efficient as possible. When applied to a set of benchmarks, MaTSa discovered a variety

```
private Set<ApplicationArchive> allArchives;

public Set<ApplicationArchive> getAllApplicationArchives() {
    if (allArchives == null) {
        HashSet<ApplicationArchive> ret = new HashSet<>(applicationArchives);
        ret.add(root);
        allArchives = Collections.unmodifiableSet(ret);
    }
     return allArchives;
    }
```

Fig. 9. Race in Quarkus

of races and provided informative reports as to their origin. We are continuing to explore ways to make our tool faster and more insightful. The design patterns we present are not limited to Java and can be applied to various garbage-collected and managed languages.

ACKNOWLEDGMENTS

This work was in part funded by the European Union project AERO, No 101092850. We thank Periklis Papoutsis for helping us understand the mechanics of JVM and OpenJDK, as well as the CARV JVM Group for their feedback. We additionally thank all the contributors of the Java TSan Project that provided crucial hints for the instrumentation code.

A ALTERNATIVE STACK TRACING

A.1 Introduction

Modern debugging tools, such as ThreadSanitizer (TSan) [20] or ManagedThreadSanitizer (MaTSa) , rely heavily on stack traces to identify the source of bugs, including data races, memory leaks, and buffer overflows. These tools help developers understand the context in which a memory access error, or data race, occurred by providing detailed stack traces that pinpoint the execution path leading to the fault. Recording stack traces efficiently is crucial as it ensures that valuable historical context is preserved for analysis without introducing excessive performance or memory overhead. Excessive memory pressure can degrade the performance of the application or even result in out-of-memory errors. This report explores methodologies for recording stack traces per memory access with minimal memory impact while maintaining sufficient detail to assist in postmortem and real-time debugging. The techniques discussed are essential for reducing the memory footprint of historic stack traces in complex systems. Although this report focuses on the Java programming language, the techniques used can be applied to different languages with little modification.

A.2 Design

The main problem we are called to solve when it comes to recording *history* of an execution, is memory usage. Recording every event can become really expensive, especially when a memory location is accessed frequently (e.g. in a loop).

This report proposes recording a limited amount of function enter/exit events and also limiting the amount of recorded accesses per memory location.

A.2.1 Recording Function Events. We will record every function enter and exit event **per thread**, in a buffer called the **event buffer**. The buffer size is configurable and defaults to 65k events. Each event it stores has the following form:

- **function address** : 48 bits *NULL in case of function exit*
- **code location** : 16 bits

We use **function address** to get information such as the name of the function and the file the function is defined in.

Code location is used to calculate the line number in which the function call was made at. In case of Java, we store the BCI (which is guaranteed to be at most 16 bits).

A.2.2 History Buffer - Handling Overflows in Event Buffer. It is obvious that an event buffer of 65k events, is relatively small for a large application. In case of overflow, we can wrap around and start storing at the beginning, essentially making it behave like a circular buffer. The problem with this approach is that information previously stored, will be lost. This is a problem for information that is still relevant, i.e., functions called near the bottom of the stack that have not yet returned. There are two options, either have larger event buffers, or multiple smaller. We propose to use smaller buffers arranged in a circular buffer, called the **history buffer**. This option has the advantage that multiple smaller buffers allow younger access information to be preserved longer, since wrapping around will mean that older buffers will be overwritten, while at the same time relevant events can be maintained for longer time, requiring shorter buffer traversals to recreate the stack.

Handling overflows now requires additional steps and information to be stored. To be precise, we need:

- **Event Buffer Epoch**
- **Stack Snapshot**

The **event buffer epoch**, is used as a state counter for each event buffer. A different value than the current would indicate that event buffer was overwritten.

Stack Snapshot, as the name suggests, is the copy of the current stack trace at the time of the event buffer transition. We use a separate pointer for the current stack to save space in the event buffer.

On each overflow, we move to a different event buffer and increment its epoch and make a snapshot of the current stack. Including a copy of the current stack with every buffer at the point of buffer initialization ensures that stack traces will not need to traverse any buffers prior to this one.

A.2.3 Storing information per memory access. Each memory access may have its own stack trace. This means, that for every memory access we have to store some metadata (cells). Storing and retrieving metadata has to be done in constant time so it does not increase the overhead substantially, while maintaining a relatively small memory footprint.

The information we need to store is the following:

- **Event Buffer Index**
- **Event Buffer Epoch**
- **History Buffer Index**
- **Access Code Location**
- **Thread ID**

The **Event Buffer Index** stores the index in the current thread's active event buffer, while **Event Buffer Epoch** stores that buffer's epoch. **History Buffer Index** indicates which **event buffer** was used during the access and **Access Code Location** is used to estimate a line number for the access *can be similar to the one we record in function enter/exit and be relative to each method.* **Thread ID** is to keep the index of the accessing thread.

The size for each field is configurable, however we recommend not using over 8 bytes of memory overall, to leverage hardware's fast loads and stores.

Storing metadata for every access can be expensive in terms of memory, so we opted to limit the information per memory location to a constant value *(default value for our testing was **4** metadata cells per memory location (variable).)*

A.2.4 Shadow Size and Mapping. We dedicate a custom memory region for storing information mentioned on section C, called the shadow memory [**?**]. The shadow memory is pre-allocated with mmap, and the actual allocation happens via page-faults.

The size of the shadow memory is relative to the number of cells stored per memory location. In our tests with **4 metadata cells per memory location** we ended using **4x the application memory**, in our case we only recorded heap accesses so that was **4x the heap size**.

In addition to a relatively small memory footprint, we also need a way to map its memory location to a shadow region that contains N metadata cells. In our case:

$$\text{heap}_{\text{offset}} = x - \text{heap}_{\text{base}}$$
$$\text{shadow}_{\text{index}} = \text{heap}_{\text{offset}} \times N$$
$$\text{shadow}_{\text{address}} = \text{shadow}_{\text{base}} + \text{shadow}_{\text{index}}$$

Where the Java heap starts at $\text{heap}_{\text{base}}$, and shadow memory starts at $\text{shadow}_{\text{base}}$

REFERENCES

[1] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[2] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: compositional static race detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276514

[3] Community. Accessed August 2024. ThreadSanitizer Documentation. https://clang.llvm.org/docs/ThreadSanitizer.html

[4] Community. Accessed August 2024. ThreadSanitizer Go Documentation. https://github.com/google/sanitizers/wiki/ThreadSanitizerGoManual

[5] Aleksey Shipilev et al. 2018. JEP 318: Epsilon: A No-Op Garbage Collector. https://openjdk.org/jeps/318. Accessed: 2024-10-30.

[6] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. *SIGPLAN Not.* 44, 6 (jun 2009), 121–133. https://doi.org/10.1145/1543135.1542490

[7] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1806672.1806674

[8] Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2023. TeraHeap: Reducing Memory Pressure in Managed Big Data Frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 694–709. https://doi.org/10.1145/3582016.3582045

[9] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[10] Looking inside a Go Race Detector 2017. https://www.infoq.com/presentations/go-race-detector/

[11] Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java memory model. *SIGPLAN Not.* 40, 1 (jan 2005), 378–391. https://doi.org/10.1145/1047659.1040336

[12] Mayur Naik and Alex Aiken. 2007. Conditional must not aliasing for static race detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 327–338. https://doi.org/10.1145/1190216.1190265

[13] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. *SIGPLAN Not.* 41, 6 (June 2006), 308–319. https://doi.org/10.1145/1133255.1134018

[14] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. *SIGPLAN Not.* 38, 10 (June 2003), 167–178. https://doi.org/10.1145/966049.781528

[15] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. *SIGPLAN Not.* 38, 10 (June 2003), 167–178. https://doi.org/10.1145/966049.781528

[16] OpenJDK TSan project 2019. https://wiki.openjdk.org/display/tsan/Main

[17] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 3 (jan 2011), 55 pages. https://doi.org/10.1145/1889997.1890000

[18] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* 15, 4 (nov 1997), 391–411. https://doi.org/10.1145/265924.265927

[19] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[20] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2011. Dynamic race detection with LLVM compiler. In *Proceedings of the Second International Conference on Runtime Verification* (San Francisco, CA) *(RV'11)*. Springer-Verlag, Berlin, Heidelberg, 110–114. https://doi.org/10.1007/978-3-642-29860-8_9