MaTSa: Race Detection in Java

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

Abstract

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), 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 Renaissance 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 15× faster on average, while scaling to large programs not supported by other tools.

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

Keywords: 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. 2025. MaTSa: Race Detection in Java. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3759548.3763369

This work is licensed under a Creative Commons Attribution 4.0 International License.
VMIL '25, Singapore, Singapore
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2164-9/25/10
https://doi.org/10.1145/3759548.3763369

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, lack of reproducibility, non-deterministic behavior, complexity of concurrent execution, and subtle symptoms. Their timing-dependent nature 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 [20] and Happens-Before [9] relation checkers. Each technique has been used in both static and dynamic analysis 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[20], or statically as e.g., in Locksmith [18]. 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 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 [4, 21, 22]. 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 [17] and Go [5, 11]. 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) [12] 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 extra 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, Java uses 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.*

To address these challenges, Java TSan, the current state-of-the-art dynamic race detector for Java, makes 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. Design decisions like these 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 evaluate 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, 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 similar occurrences in real software.

Overall, this paper makes the following 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. To enable its use on large programs, MaTSa implements its dynamic analysis also with JIT compilation, in addition to interpretation.
- We compare MaTSa to Java TSan quantitatively and qualitatively. We evaluate both tools and compare their 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 MaTSa design principles as well as some important parts of the Java language specification. Section 4 compares the MaTSa design with Java TSan and other Java race detection tools. 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 (locks), 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. On 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, 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: $k_i = W \lor k_j = W$. And 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. 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, accesses by the other threads will have *happened-before* the accesses about to be executed in this thread.

In the example, 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 The Anatomy of a Race Detector

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

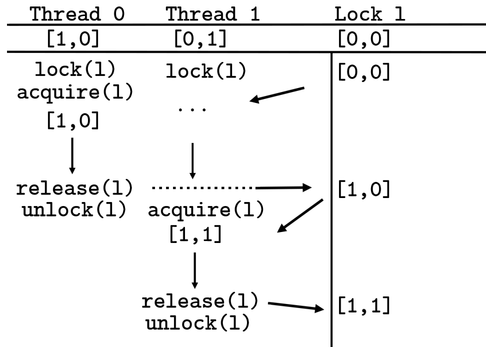

Figure 1. Vector clock transfer between 2 threads via a shared lock.

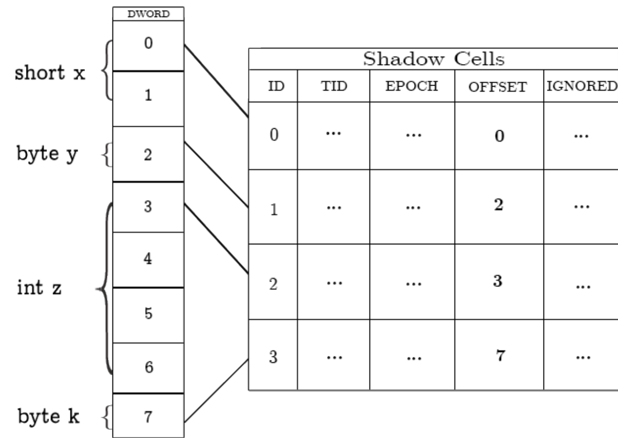

Figure 2. Offset of fields of different sizes.

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.

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 every shadow region corresponds to 8 bytes of application memory, 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.

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 128k 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 Garbage Collection-Aware Shadow Memory

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 Java TSan checks every object that gets freed/moved to clear/update the corresponding shadow region. This, however, adds 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

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

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% but are significantly faster and enable support for every garbage collector with contiguous heaps, without any modification in the collector itself.

Because (by default) the JVM allocates one fourth of the available physical memory and the shadow memory is 8 times larger than the heap, we have observed that in systems with more DRAM the peak memory consumptions was larger than in systems with less DRAM

3.4 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.

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.

3.5 Runtime

We extended the JVM template interpreter and the IR of both C1 and C2 compilers, by instrumenting every load and store instruction for fields and arrays. While C1 and C2 compiler instrumentation is not architecture specific, we only support the bytecode templates for x86 for the interpreter. In contrast to Java TSan, the current state-of-the-art dynamic race detector, which requires users to run entirely in interpretation mode, with significant overhead, our approach achieves 15× faster performance on average and up to 56× at best, by leveraging both the interpreter and the JIT compilers (C1 and C2), avoiding the cost of interpretation-only execution.

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

3.6 The Art Of Time Travel

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. 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.

We 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 64k events each.

To keep information about the stack of each memory access, we use a second shadow memory region, *Shadow History*. This region stores information on the position in the event buffer at the time of the access. We store 64 bits of information per access, including 16 bits for the BCI (*bytecode index*), 16 bits for ring_idx (*index in event buffer*) and 16 bits for history_epoch (*detect if the event buffer has been overwritten*).

3.7 Reconstructing The Crime Scene

Every detected race 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 check the access's 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.

3.8 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 `BCI` and the `method pointer`. The `BCI` is the bytecode index and is guaranteed to be unique per method. On each report, we add the `BCI` and current `method pointer` of the current code location and before each report, we check if we have an instance of the current `method pointer & BCI` 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. It is important to note that `method pointers` can be invalid after a garbage collection cycle (and particularly a class unloading event). To tackle that, we have disabled class unloading which we found adds negligible memory overhead to the application.

3.9 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 added `java.lang.System` methods `MaTSaLock(lockobject)` and `MaTSaUnlock(lockobject)`, which may be used to notify MaTSa about a lock or unlock event, respectively.

4 Comparison Against State-Of-The-Art

4.1 Java TSan

Execution Modes: A key architectural difference between MaTSa and Java TSan lies in their supported execution modes, which significantly impact runtime performance. The JVM begins by interpreting Java bytecode and collecting profiling information. Based on this data, frequently executed *("hot")* methods are compiled into optimized machine code by the C1 and C2 JIT compilers. MaTSa operates seamlessly across all execution tiers: the interpreter, C1, and C2. Running exclusively in interpreter mode ensures that no memory accesses are optimized away but at the cost of significantly slow execution. To stay consistent with this correctness constraint, we instrument compiled code during the parsing phases of C1 and C2 compilation. This guarantees that all relevant accesses are captured, while achieving performance that, despite being slower than uninstrumented compiled code, is still orders of magnitude faster than pure interpretation.

Volatile: According to the Java Memory Model [12], 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 3 shows an example of a volatile access used as a custom synchronization mechanism, in effect creating a happens-before 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 4 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: 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 initialized by a static class initializer will wait until

```
1  public class VolatileFalsePositive {
2      public static volatile int x = 15;
3      public static int y;
4      public static void main(String[] args) {
5          Thread t1 = new Thread(
6              () -> {
7                  int tmp; // int tmp = x;
8                  while ((tmp = x) != 42);
9                  y = tmp + 1;
10                 x = 2;
11             });
12         Thread t2 = new Thread(
13             () -> {
14                 int tmp = x;
15                 y = tmp + 1;
16                 x = 42;
17             });
18     }
19 }
```

Figure 3. Non-Race between volatile accesses

```
1  public class VolatileFalseNegative {
2      public static volatile int x = 15;
3      public static int y;
4      public static void main(String[] args) {
5          Thread t1 = new Thread(() -> {
6                  int tmp = x;
7                  y = tmp + 1;
8                  x = 2;
9              });
10         Thread t2 = new Thread(() -> {
11                 int tmp = x;
12                 y = tmp + 1;
13                 x = 2;
14             });
15 }   }
```

Figure 4. Race between volatile accesses

the initializer has finished, 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.

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,

[2]OOP as class initialization lock

```
1  public class ClassInitFalseNegative {
2      private static int counter = 1;
3      private static synchronized void sync()
4        { counter = 3; }
5      private static void notsync() { counter++; }
6      public static void main(String[] args) {
7          Thread thread1 = new Thread(() ->
8            { sync(); });
9          Thread thread2 = new Thread(() ->
10           { notsync(); });
11 }   }
```

Figure 5. Static Class Initializer False Negative

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 5 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. Examining the OpenJDK source code, 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 models 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 library, it can again form bogus happens-before edges in cases where a simple CompareAndSwap operation is performed, thus hiding actual races. To address this, we studied every lock in the library and added callbacks to MaTSa, significantly improving accuracy. This also 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 releasing it just before the lock is released[3]. One other

[3]MaTSa uses this trick to avoid requiring synchronization for its own vector clocks. 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.

aspect 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 can produce false negatives. In order to model thread join, we have added a callback to MaTSa when the child thread is stopping to save its vector clock into the thread object. Also, just before thread join returns to the caller, MaTSa 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 in one 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. In case of multiple races involving common locations, MaTSa will report one. The others will become visible in subsequent runs after the previous are patched or ignored.

4.2 Other Race Detection Tools

Extensive prior research has been conducted on race detection in Java. Static analysis tools such as Meta's Infer [2] and others [13, 14] can be sound, but are impractical due to high false positives and cannot handle reflection or dynamic code generation. Dynamic analyses such as DrFinder [3, 15] or hybrid systems [16] do not scale to large programs, or do not support recent versions of Java, custom synchronization and libraries. Additionally, Fray [10] is a dynamic concurrency testing suite, which works by adding annotations to parts of the code (like JUnit) that requires testing or by using an agent.

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 7 or 8. 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. This approach 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 against it. However, as it can only work for Java 7 or 8 modern applications (such as the benchmarks we chose) cannot run.

5 Evaluation

We evaluate MaTSa in terms of performance and compare it with Java TSan, the current state-of-the-art dynamic race detector for Java, using the DaCapo Benchmark suite [1], omitting applications that do not run successfully with OpenJDK v17 and v21. We also use MaTSa to detect and analyze races in the Renaissance Benchmark suite [19] which contains a multiple large applications (such as Spark) that cannot run with Java TSan due to missing support for JIT compilation. We used the G1 garbage collector in all experiments. We measure total runtime, peak memory consumption, and reported races. The system used for running the benchmarks had an Intel Xeon Gold 5512U cpu with 28 cores, 56 threads and max frequency of 3.7Ghz and 256GiB of RAM.

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 4 show average execution times for each benchmark run in MaTSa and the vanilla OpenJDK17, respectively. Column 5 shows the slowdown factor, *i.e.*, the overhead incurred by MaTSa instrumentation, checking, and output. Columns 6–8 show the same measurements for Java TSan and OpenJDK21, respectively. In every benchmark MaTSa outperforms Java TSan due to it leveraging JIT compilation. Column 3 shows the execution time without JIT for reference. Moreover, columns 9 and 10 show the number of race warnings produced by the two tools. To facilitate comparison, we group the number of warnings for Java TSan according to source code locations, as in MaTSa. Overall, MaTSa reported at least all code locations mentioned in Java TSan warnings in all benchmarks.

Table 2 compares peak memory usage per benchmark between the two tools. The numbers shown represent the *Maximum resident set size*. As above, the first column shows the benchmarks. Columns 1 and 2 show peak memory usage for MaTSa and the vanilla OpenJDK 17 in default configuration, respectively. Similarly, columns 4-5 show the same measurements for Java TSan, OpenJDK 21 in default configuration and their difference, respectively. The primary cause for the memory overhead is the additional shadow memory, used to keep track of historic accesses (Section 3.6). Java TSan keeps a fixed size circular array of the most recent accesses (size is configurable, with the cost of the traversal (search) and the possibility of missing a previous trace, while MaTSa records every access and preserves them until a GC (Section 3.3).

Benchmark	MaTSa				Java TSan			Warnings (grouped)	
	MaTSa	MaTSa Xint	JDK17	Slowdown	Java TSan	JDK21	Slowdown	MaTSa	Java TSan
avrora	1m56s	3m13s	1m15s	1.55x	8m23s	1m14s	6.80x	7	46(7)
batik	0m36s	1m1s	0m4s	9.00x	1m8s	0m4s	17.00x	0	0
biojava	7m30s	25m6s	0m12s	37.50x	25m13s	0m10s	151.30x	0	0
eclipse	6m0s	13m45s	0m37s	9.73x	13m30s	0m34s	23.82x	143	134(36)
fop	0m22s	0m46s	0m4s	5.50x	0m52s	0m4s	13.00x	0	0
graphchi	6m32s	11m52s	0m9s	43.56x	23m10s	0m8s	173.75x	0	0
jme	0m17s	0m35s	0m8s	2.12x	0m34s	0m8s	4.25x	0	0
jython	2m20s	7m22s	0m9s	15.56x	8m24s	0m9s	56.00x	0	0
luindex	2m37s	9m24s	0m7s	22.43x	7m9s	0m8s	53.62x	0	0
lusearch	1m38s	4m46s	0m3s	32.67x	79m56s	0m8s	599.50x	18	11(10)
pmd	1m13s	3m16s	0m9s	8.11x	68m43s	0m8s	515.38x	58	110(57)
sunflow	3m44s	4m2s	0m4s	56.00x	152m55s	0m6s	1529.17x	5	5(5)
xalan	0m40s	0m40s	0m3s	13.33x	32m20s	0m3s	646.67x	37	28(10)
zxing	1m3s	1m2s	0m3s	21.00x	1m04s	0m3s	21.33x	21	22(17)

Table 1. Performance and Race Warning Comparison

Benchmark	MaTSa			Java TSan		
	MaTSa	JDK17	Overhead	Java TSan	JDK21	Overhead
avrora	1.54	0.14	10.65x	0.50	0.13	3.80x
batik	3.77	0.59	6.45x	3.13	0.50	6.34x
biojava	5.55	2.15	2.58x	14.81	2.19	6.76x
eclipse	22.51	1.28	17.59x	10.07	1.40	7.19x
fop	4.73	0.52	9.22x	1.77	0.58	3.08x
graphchi	7.15	1.93	3.70x	9.57	2.82	3.39x
jme	0.47	0.22	2.06x	0.93	0.19	4.80x
jython	18.40	1.03	28.61x	11.30	1.34	8.43x
luindex	0.74	0.90	0.81x	3.41	0.96	3.55x
lusearch	10.08	3.25	3.10x	10.16	2.40	4.23x
pmd	17.87	2.62	6.82x	19.70	1.87	10.53x
sunflow	6.52	3.15	2.07x	14.33	2.74	5.23x
xalan	8.88	1.44	6.17x	4.52	1.42	3.18x
zxing	4.10	1.65	2.48x	3.66	1.05	3.49x

Table 2. Memory Usage Comparison (GBs)

Benchmark	**Warnings**	**Scala Related**	**Runtime**
akka-uct	252	221	5m50s
als	211	114	1m48s
chi-square	76	22	1m30s
db-shootout	87	0	5m19s
dec-tree	177	67	0m57s
dotty	5	5	0m44s
fj-kmeans	31	0	2m51s
future-genetic	83	0	3m28s
gauss-mix	176	134	18m9s
log-regression	210	100	1m5s
mnemonics	0	0	2m40s
movie-lens	304	172	5m41s
naive-bayes	215	106	3m5s
page-rank	132	46	4m4s
par-mnemonics	26	0	2m13s
philosophers	15	13	2m25s
reactors	42	30	7m24s
scala-doku	0	0	2m43s
scala-kmeans	0	0	0m21s

Table 3. Races in Renaissance

The luindex benchmark produces a counter-intuitive result. Less peak memory usage with MaTSa compared to vanilla JVM. Our analysis indicates that the introduced JIT instrumentation during the parsing phase prevents certain crucial optimizations, such as escape analysis and scalar replacement. These optimizations, reduce heap allocations thereby minimizing the need for GC. Consequently, the run with MaTSa triggered *31* garbage collection cycles, a significant increase compared to just *8* cycles in the default run. While more frequent GCs might seem problematic, in this specific case, they effectively reclaimed memory more aggressively, leading to a lower peak memory footprint despite the overhead of instrumentation.

Table 3 shows how many races were reported by MaTSa when running the Renaissance Benchmark suite [19]. The first column shows the name of each benchmark, the second column shows the total number of warnings generated by MaTSa, the third column shows how many of the reported warnings were verified to be true races. Due to the large number of warnings (which is common for dynamic race detection tools) we were only able to verify a sample of them. We consider a warning to be a candidate for further investigation if the method which the race was reported was a method directly related to the benchmark. For example for db-shootout, such a method would be `org.h2.foo()`. It is important to note that this work does not focus and does not handle custom synchronization within the scala standard library (and scala extensions such as akka). Therefore, most of the races reported in akka, als, chi-square, dec-tree, dotty, gauss-mix, log-regression, movie-lens, naive-bayes, page-rank and reactors are false positives due to missing happens-before edges created in the libraries. In addition, there have been a lot of races related to accesses in a constructor which are mostly false positive as `this` is not made available until after initialization has been completed. There is a case however in which such races might be true if for

some reason `this` is leaked within the constructor. Currently, it is not possible to form a pseudo happens-before edge between the initialization in a constructor and the access as it would significantly impact execution time, hence such cases have to be manually suppressed. Finally, we did not run the suite with Java TSan as the benchmarks were taking excessive time to complete, which is expected due to the sole use of the interpreter.

Due to space constraints, we have thoroughly analyzed the 7 verified races reported in `db-shootout` and some detected within the Java standard library across all benchmarks. The first race in `db-shootout` is related to the binary search function. While the map used is thread safe, the binarySearch function takes an extra argument `cachedIndex`, which is used as the start of the binary search. This index is updated after each binarySearch call without any synchronization or the use of volatile. Three of the six races are related to the unsynchronized read of an index of a hash map in find, and the writes in put and remove. The race is benign in find/put but can produce false results in find/remove as an element that has just been removed could be returned. One race in AutoDetectDataType due to caching the last detected type but not protecting the read/write to last. However, it is benign as the worst that can happen is recomputing the correct type. Finally, there is a data race in insertKey of the b-tree. To be exact, a new KeyStorage is created (which is an array of objects) and then assigned to field keys. Keys is then used in binary search with no synchronization between the write in insertKey and read in binarySearch which could expose the old KeyStorage to binary search instead.

As to races in Java's standard library, we have identified a race in String.hashCode which is benign and documented within the source code. Additionally, there is a race in Java's Class.getSimpleName, where a string is cached after the first call to the function and a check is performed to see if a class name has been written before, therefore creating a benign race which could lead to reinitialization of simpleName.

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 6, 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 7, 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.

```
1   private static Integer globalIdx = 0;
2   private static int inc() {
3     int rtn;
4     synchronized (globalIdx) {
5     // globalIdx += stride means:
6     // globalIdx = new Integer(globalIdx + stride)
7       rtn = globalIdx += stride;
8     }
9     return rtn;
10  }
```

Figure 6. Race in DaCapo 23.11-chopin

```
1  private Set<ApplicationArchive> allArchives;
2
3  public Set<ApplicationArchive>
4  getAllApplicationArchives() {
5      if (allArchives == null) {
6          HashSet<ApplicationArchive> ret =
7            new HashSet<>(applicationArchives);
8          ret.add(root);
9          allArchives =
10           Collections.unmodifiableSet(ret);
11     }
12     return allArchives;
13 }
```

Figure 7. Race in Quarkus

6 Conclusion

We have developed MaTSa, a tool that dynamically detects 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 fast and efficient. When applied to a set of benchmarks, MaTSa discovered a variety 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 and VMware's University Research Fund. 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 for providing crucial hints for the interpreter instrumentation code.

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. doi: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. doi:10.1145/3276514

[3] Eric Bodden and Klaus Havelund. 2008. Racer: Effective race detection using AspectJ. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 155–166.

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

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

[6] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. *SIGPLAN Not.* 44, 6 (jun 2009), 121–133. doi: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. doi: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. doi: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. doi:10.1145/359545.359563

[10] Ao Li, Byeongjee Kang, Vasudev Vikram, Isabella Laybourn, Samvid Dharanikota, Shrey Tiwari, and Rohan Padhye. 2025. Fray: An Efficient General-Purpose Concurrency Testing Platform for the JVM. arXiv:2501.12618 [cs.PL] https://arxiv.org/abs/2501.12618

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

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

[13] 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. doi:10.1145/1190216.1190265

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

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

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

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

[18] 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. doi:10.1145/1889997.1890000

[19] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. doi:10.1145/3314221.3314637

[20] 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. doi:10.1145/265924.265927

[21] 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. doi:10.1145/1791194.1791203

[22] 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. doi:10.1007/978-3-642-29860-8_9

Received 2025-07-09; accepted 2025-08-11