MaTSa: Race Detection in Java

Alexandros Emmanouil Antonakakis
Iacovos G. Kolokasis
Foivos S. Zakkak
Angelos Bilas
Polyvios Pratikakis

ICS-FORTH, University of Crete, Red Hat

VMIL 2025
Singapore

Motivation

- **Data races in Java** are difficult to **detect, debug, and reproduce**
- **Current race detection tools** often suffer from:
  - **Low precision** → false positives, false negatives
  - **High overhead** → slow performance, small tests only, difficult for large setups
  - **Dynamic code generation** → Unseen code in large frameworks, missed bugs
- **External tools** frequently **ignore key nuances** of the **Java Memory Model (JMM)**

Objectives

- Build a **race detector inside OpenJDK**
  - JIT, GC
- Handle JMM semantics correctly
  - Less false negatives, support for implicit orderings
- Provide **informative race reports** (stack traces, source lines)
- Scale to large real-world applications
  - Large runs, large codes

Contributions

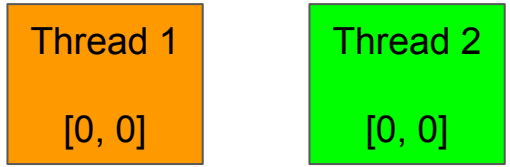
- **MaTSa**: *Managed Thread Sanitizer*
  - Implements **FastTrack** happens-before with vector clocks (but differently)
  - Integrated directly into **OpenJDK (interpreter + JIT)**
  - Tracks memory accesses, synchronization, and happens-before relations
  - **Fast stack reconstruction** for past accesses, better error messages
- **Comprehensive comparison** with Java TSan
- **Experimental evaluation** on DaCapo, Renaissance, Quarkus
- **Impact**: Found unknown races, upstream fixes

Algorithm (briefly)

- Happens-before with vector clocks

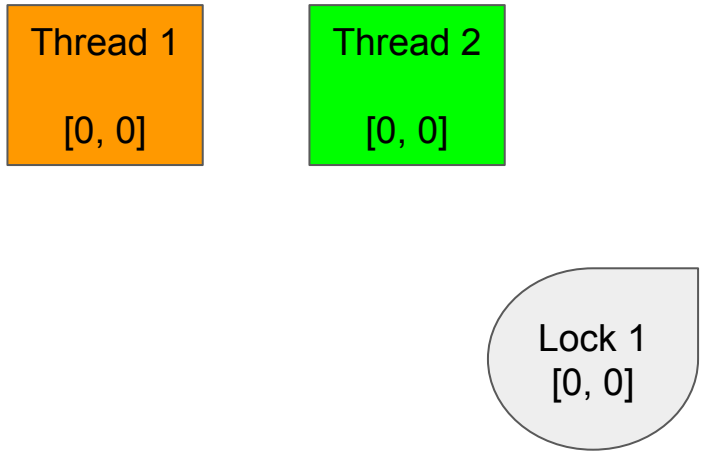
Algorithm (briefly)

- Happens-before with vector clocks
  - Every thread gets a vector clock



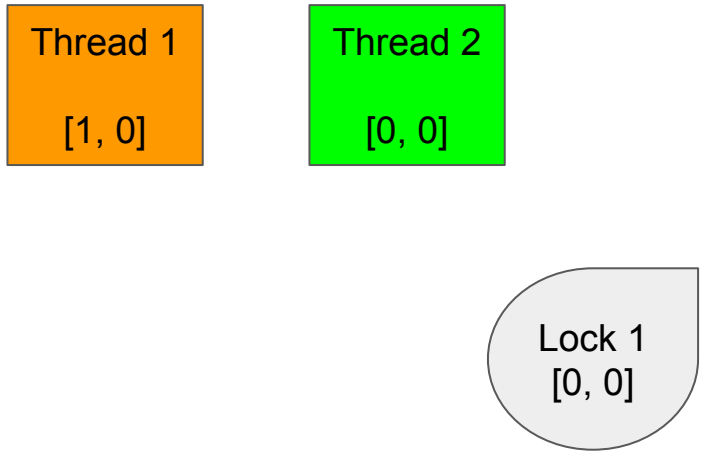
Algorithm (briefly)

- Happens-before with vector clocks
  - Every thread gets a vector clock
  - Every lock gets a vector clock



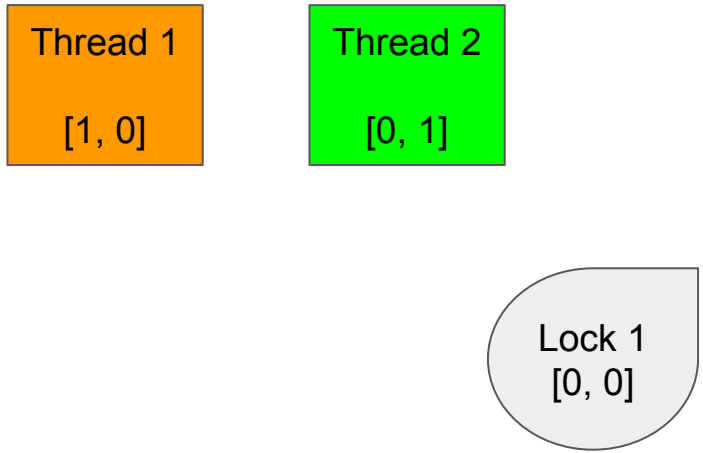
Algorithm (briefly)

- Happens-before with vector clocks
  - Every thread gets a vector clock
  - Every lock gets a vector clock



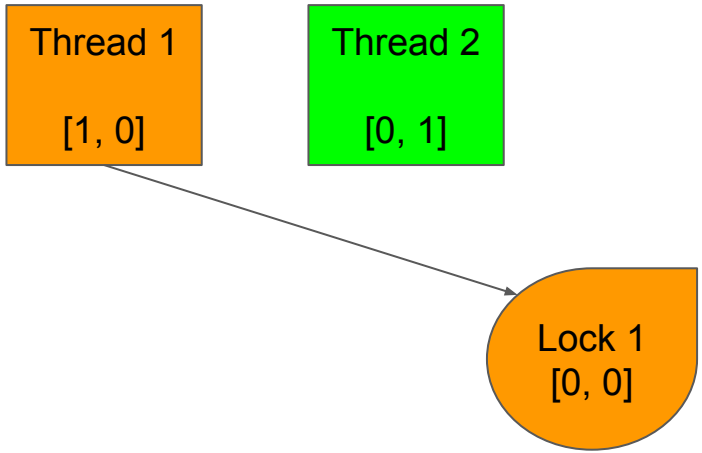
Algorithm (briefly)

- Happens-before with vector clocks
  - Every thread gets a vector clock
  - Every lock gets a vector clock



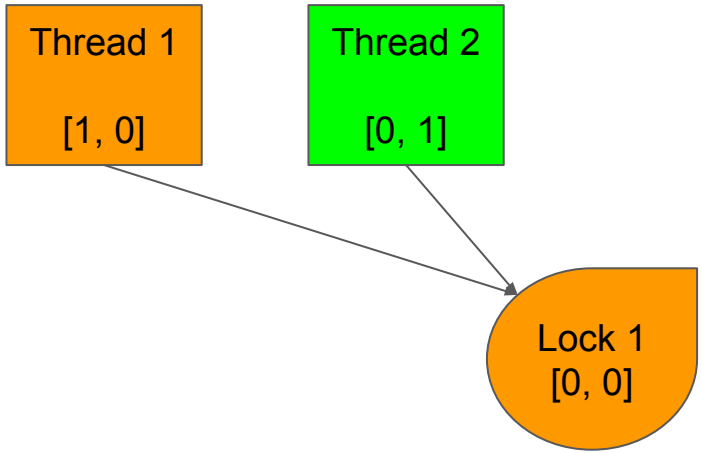
Algorithm (briefly)

- Happens-before with vector clocks
  - Every thread gets a vector clock
  - Every lock gets a vector clock
  - Thread synchronizes with lock
    - Clocks get synchronized



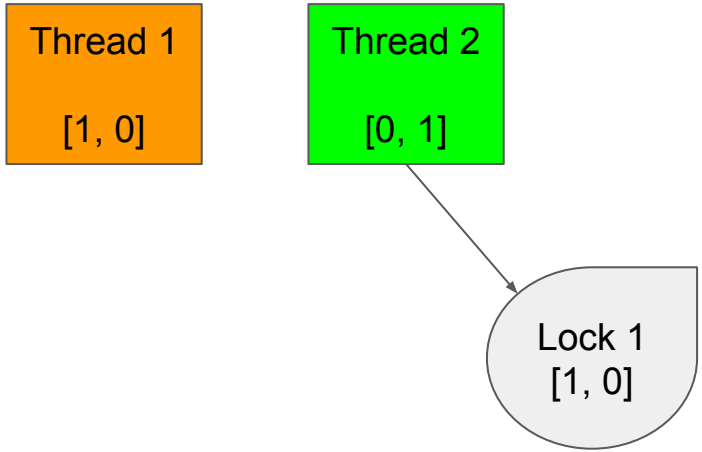
Algorithm (briefly)

- Happens-before with vector clocks
  - Every thread gets a vector clock
  - Every lock gets a vector clock
  - Thread synchronizes with lock
    - Clocks get synchronized



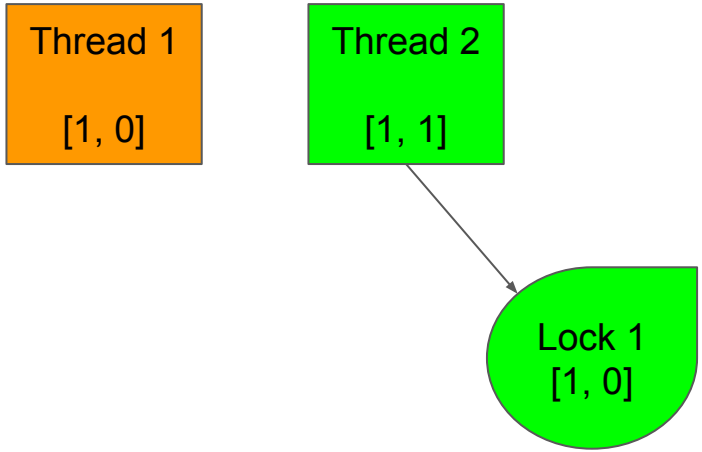
Algorithm (briefly)

- Happens-before with vector clocks
  - Every thread gets a vector clock
  - Every lock gets a vector clock
  - Thread synchronizes with lock
    - Clocks get synchronized



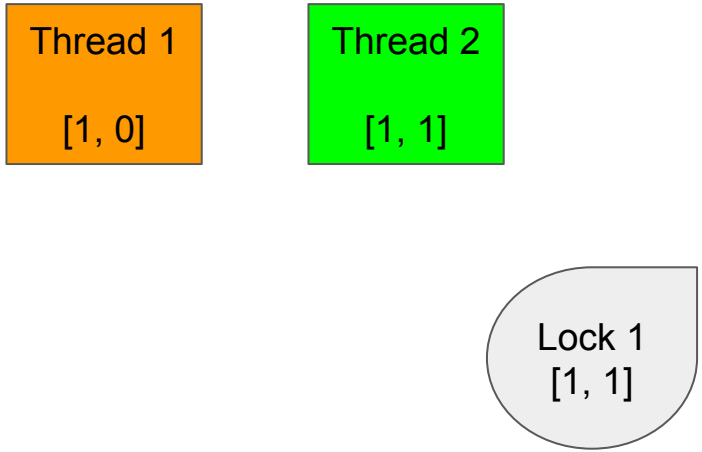
Algorithm (briefly)

- Happens-before with vector clocks
  - Every thread gets a vector clock
  - Every lock gets a vector clock
  - Thread synchronizes with lock
    - Clocks get synchronized

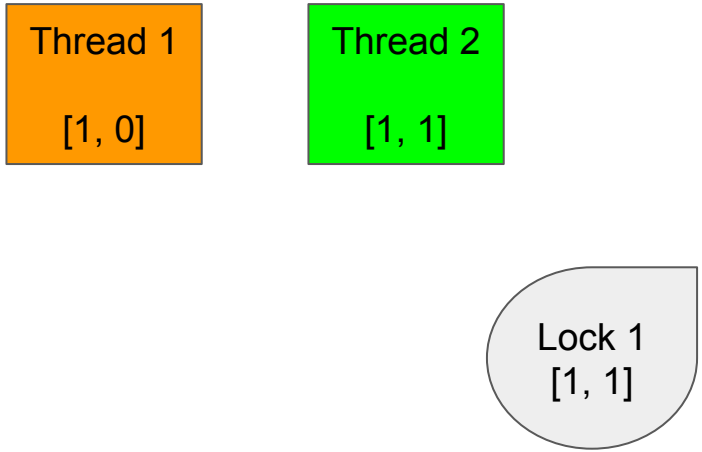


Algorithm (briefly)

- Happens-before with vector clocks
  - Every thread gets a vector clock
  - Every lock gets a vector clock
  - Thread synchronizes with lock
    - Clocks get synchronized
    - Captures happens-before



Algorithm (briefly)



- Happens-before with vector clocks
  - Every thread gets a vector clock
  - Every lock gets a vector clock
  - Thread synchronizes with lock
    - Clocks get synchronized
    - Captures happens-before
- On every access
  - Log minimum info: thread ID, thread time, write
  - Fast representation, shadow memory
  - Remember 4 “last” accesses
  - Compare, if no happens-before and one access is a write, **race**!
    - **Reconstruct stack** for previous access, print report

Design Points

- **Shadow Memory** for efficient metadata storage
  - Stores access log, stack reconstruction logs
  - 4x Heap size
- **GC precision trade-off**
  - GC **migrates** objects during compaction, evacuation, tenuring
  - **Very high cost** of tracking migrations and relocating shadow access logs
  - How about **scrapping all info** on every GC? (Turns out warnings rarely lost!)
  - **Unmap-remap** the shadow: Very fast, less total memory used

Design Points

- **GC object migration remains an issue**
  - Every object is also a lock
  - Solution: Vector clock **embedded** in object header
  - Follows object migration
  - Works with every GC

Design Points

- **Stack trace reconstruction**: “Time Travel” for past accesses
  - **Reconstruct** stack trace for past accesses (but fast)
  - Idea: Use stack event buffers, **append**-only, **restart** logging often
  - Immutable buffers, pointers from accesses work
  - “**Replaying**” stack is fast, small buffers
- **Report Map** for concise, non-duplicate race warnings
  - Based on source line information (Reconstructed from BCI)
  - Avoid reporting race for same lines of code, even if different objects
  - Reduces redundant warnings, easier to debug

Implementation

- Instrumented **interpreter** bytecodes (x86) and **C1/C2 IR** to cover loads/stores
  - C1 & C2 instrumented during parsing phase
    - Intentionally **limit optimizations** to account for all accesses
  - On every access: create access cell, check for race, add to shadow memory log
- Add callbacks on most `java.util.concurrent` synchronization mechanisms
  - Establishes happens-before edges
- Instrument every **synchronized method/block** enter/exit

Design comparison: MaTSa vs Java TSan

Nuance	MaTSa	Java TSan
Execution tiers	Interpreter + C1/C2 JIT	Interpreter-only
Volatile fields	**Ignored**	Modeled as a lock/unlock operation
Static class initializers	**Separate** internal vector clock	Reuses class object vector clock
Thread.join	Modeled via callbacks at stop/join	Not modeled

Volatile Fields

```
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 }
```

Volatile Fields

```
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 }
```

```
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              () -> {
7                  int tmp = x;
8                  y = tmp + 1;
9                  x = 2;
10             });
11         Thread t2 = new Thread(
12             () -> {
13                 int tmp = x;
14                 y = tmp + 1;
15                 x = 2;
16             });
17     }
18 }
```

Evaluation setup

- Benchmarks
  - DaCapo (subset on JDK 17/21)
  - Renaissance
  - Quarkus (3.8)
- Metrics: Runtime, memory used, number of (grouped) race warnings
- Environment: Xeon Gold 5512U (28C/56T), 256 GiB RAM; G1 GC used

Performance Results

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)

Memory Usage

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

Race Warnings in Renaissance

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

Overall Results

- Performance: MaTSa is **~15x faster** on average than Java TSan
  - Up to **56x** faster
- Scale: MaTSa handles **large** apps (e.g., Spark via Renaissance)
  - Impractical with interpreter-only tools
- Memory: Overhead dominated by **shadow/stack** history

The memory overhead paradox

- Observed while running **luindex**
  - RSS peak ~740MB with MaTSa **enabled**
  - RSS peak ~900MB with MaTSa **disabled**
- JIT instrumentation during the parsing phase prevents certain optimizations
- Results in **more** garbage collection cycles
  - MaTSa triggered 31 GCs
  - Vanilla triggered 8 GCs

A notable race

```
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   }
```

Limitations

- **Custom synchronization** not auto-modeled
- Scala/akka library internals may create missing HB edges
  - **False positives** (see Renaissance results)
- **Constructor**-internal writes may appear racy
  - Cannot be modeled away, as **object leakage** cases matter

Conclusion

- MaTSa is the **first fast, precise JMM-aware race detector**
- Integrates cleanly into the JVM
  - Usable in **production-sized code**
- **OpenJDK-native implementation**
  - JIT support
- Extensible to other managed languages (e.g., Kotlin, Scala)