DOI:10.1145/1839676.1839698

Goldilocks: A Race-Aware Java Runtime

By Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran

Abstract

We present GOLDILOCKS, a Java runtime that monitors program executions and throws a `DataRaceException` when a data race is about to occur. This prevents racy accesses from taking place, and allows race conditions to be handled before they cause errors that may be difficult to diagnose later. The `DataRaceException` is a valuable debugging tool, and, if supported with reasonable computational overhead, can be an important safety feature for deployed programs. Experiments by us and others on race-aware Java runtimes indicate that the `DataRaceException` may be a viable mechanism to enforce the safety of executions of multithreaded Java programs.

An important benefit of `DataRaceException` is that executions in our runtime are guaranteed to be race free and thus sequentially consistent as per the Java Memory Model. This strong guarantee provides an easy-to-use, clean semantics to programmers, and helps to rule out many concurrency-related possibilities as the cause of errors. To support the `DataRaceException`, our runtime incorporates the novel Goldilocks algorithm for precise dynamic race detection. The Goldilocks algorithm is general, intuitive, and can handle different synchronization patterns uniformly.

1. INTRODUCTION

When two accesses by two different threads to a shared variable are enabled simultaneously, i.e., at the same program state, a race condition is said to occur. An equivalent definition involves an execution in which two threads make conflicting accesses to a variable without proper synchronization actions being executed between the two accesses. A common way to ensure race freedom is to associate a lock with every shared variable, and to ensure that threads hold this lock when accessing the shared variable. The lock release by one thread and the lock acquire by the next establish the required synchronization between the two threads.

Data races are undesirable for two key reasons. First, a race condition is often a symptom of a higher-level logical error such as an atomicity violation. Thus, race detectors serve as a proxy for more general concurrency-error detection when higher-level specifications such as atomicity annotations do not exist. Second, a race condition makes the outcome of certain shared variable accesses nondeterministic. For this and other reasons, both the Java Memory Model (JMM)[10] and the C++ Memory Model (C++MM)[2] define well-synchronized programs to be programs whose executions are free of race conditions. For race-free executions, these models guarantee sequentially consistent semantics; in particular, every read deterministically returns the value of the "last" write. This semantics is widely considered to be the only simple-enough model with which writing useful concurrent programs is practical. For executions containing race conditions, the semantics is either completely undefined (as is the case for C++MM[2]) or is complicated enough that writing a useful and correct program with "benign races" is a challenge.

Detection and/or elimination of race conditions has been an area of intense research activity. The work presented in this paper (and initially presented in Elmas et al.[6]) makes two important contributions to this area.

First, for the first time in the literature, we propose that race conditions should be language-level exceptions just like null pointer dereferences and indexing an array out of its bounds. The GOLDILOCKS runtime for Java provides a new exception, `DataRaceException`,[a] that is thrown precisely when an access that causes an actual race condition is about to be executed. Since a racy execution is never allowed to take place, this guarantees that the execution remains sequentially consistent.

The `DataRaceException` brings races to the programmer's attention explicitly. When this exception is caught, the recommended course of action is to terminate the program gracefully. This is because, for racy Java programs, a wide range of compiler optimizations are allowed by the JMM, and this makes it complicated to relate program executions to source code. If the exception is not caught, the Goldilocks runtime terminates the thread that threw the exception. While not recommended, the programmer could also choose to make optimistic use of a `DataRaceException` by, for instance, retrying the code block that led to the race, hoping that the conflicting thread has performed the synchronization necessary to avoid a race in the meantime. Since the first paper on the GOLDILOCKS runtime,[6] the idea that certain concurrency errors, especially ones that result in sequential consistency violations, should result in exceptions has gained significant support and several implementations of the idea have been investigated.[9, 11]

To support a `DataRaceException`, a runtime must

[a] We define `DataRaceException` as a subclass of the `RuntimeException` class in Java.

The original version of this paper was published in the *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

incorporate a precise yet efficient race detection mechanism. In this context, false positives in race detection cannot be tolerated. The second contribution of our work is the GOLDILOCKS algorithm, a novel, precise, and general algorithm for detecting data races at runtime. In Elmas et al.,[6] we presented an implementation of the GOLDILOCKS algorithm in a Java Virtual Machine (JVM) called Kaffe.[19] Our experiments with GOLDILOCKS on benchmarks brought up the new possibility that the overhead of post-deployment precise race detection in a runtime may be tolerable. There has been significant progress in the efficiency of precise race detection since the GOLDILOCKS runtime was first published (see Flanagan and Freund, and Pozniansky and Schuster,[7, 14] for example) and this idea appears viable today.

The GOLDILOCKS algorithm is based on an intuitive, general representation for the happens-before relationship as a generalized lockset (Goldilockset) for each variable. In the traditional use of the term, a lockset for a shared variable x at a point in an execution contains a set of locks. A thread can perform a race-free access to x at that point by first acquiring a lock in this lockset. A Goldilockset is a generalization of a lockset. In Java, locks and volatile variables are synchronization objects, and acquiring and releasing a lock, as well as reading from and writing to a volatile variable, are synchronization operations. To reflect this, at each point in an execution, the Goldilockset for a shared variable x may contain thread ids, locks, and volatile variables. A thread can perform a race-free access to x iff its thread id is in the Goldilockset, or if it first acquires a lock that is in the Goldilockset, or reads a volatile variable that is in the Goldilockset. In other words, the Goldilockset indicates the threads that have the ownership of x and the synchronization objects that protect access to x at that point. The Goldilockset is updated during the execution as synchronization operations are performed. As a result, Goldilocksets are a compact, intuitive way to precisely represent the happens-before relationship. Thread-local variables, variables protected by different locks at different points of the execution, and event-based synchronization with condition variables are all uniformly handled by GOLDILOCKS. Furthermore, Goldilocksets can easily be generalized to handle other synchronization primitives such as software transactions[18] and adapted to handle memory models of languages other than Java, such as C++MM. To facilitate this, in this paper (differently from Elmas et al.[6]) we present GOLDILOCKS on a generic memory model and then show how the algorithm can be specialized to JMM.

1.1. Related work

Dynamic Race Detection: There are two approaches to dynamic data-race detection, one based on locksets and the other based on the happens-before relation. Eraser[16] is a well-known lockset-based algorithm for detecting race conditions dynamically by enforcing the locking discipline that every shared variable is protected by a unique lock. In spite of the numerous papers that refined the Eraser algorithm to reduce the number of false alarms, there are still cases, such as dynamically changing locksets, that cannot be handled precisely. Precise lockset algorithms exist for Cilk programs,[3] but they cannot handle concurrency patterns implemented using volatile variables such as barrier synchronization.

There is a significant body of research on dynamic data-race detection based on computing the happens-before relation[4, 7, 14, 15, 17] using vector clocks.[12] Hybrid techniques[14, 20] combine lockset and happens-before analysis. For example, RaceTrack[20] uses a basic vector clock algorithm to capture thread-local accesses to objects thereby eliminating unnecessary and imprecise applications of the Eraser algorithm. Similarly, MultiRace[14] presents DJIT+, a vector clock algorithm with several optimizations to reduce the number of checks at an access, including keeping distinct vector clocks for reads and writes and using a lockset algorithm as a fast-path check. To the best of our knowledge, FASTTRACK,[7] which builds on DJIT+, is the best-performing vector clock-based algorithm in the literature. By exploiting some access patterns, FASTTRACK reduces the cost of vector clock updates to $O(1)$ on average. We provide a qualitative comparison of the GOLDILOCKS and FASTTRACK algorithms in Section 4.3. Vector clock and GOLDILOCKS are both precise, but the generalized locksets in GOLDILOCKS provide an intuitive representation of how shared variables are protected at each point the execution.

Concurrency-Related Exceptions: Since proposed first by the authors in Elmas et al.,[6] the idea that programming platforms should be able to guarantee sequential consistency for *all* programs has gained significant support. Marino et.al.[11] and Lucia et.al.[9] provide platforms with explicit memory model exceptions. Both platforms define stronger but simpler contracts than JMM and C++MM, which enable efficient hardware implementations that support the memory model exceptions.

2. A GENERIC MEMORY MODEL

In this section, we present a generic memory model and express the JMM as a special case of it. This generic model allows a uniform treatment of the various synchronization constructs in Java. We also believe that memory models at different levels (e.g., the hardware level) and for different languages (e.g., C++MM) can be expressed as instances of this model. This allows GOLDILOCKS to be applied in these settings directly.

Variables and Actions: Program variables are separated into two categories: data variables (*Data*) and synchronization variables (*Sync*). We use x, and o to refer to data and synchronization variables, respectively. Threads in a program execute actions from the following categories:

- *Data variable accesses*: $\mathsf{read}(t, x, v)$ by thread t reads the current value v of a data variable x, and $\mathsf{write}(t, x, v)$ by thread t writes the value v to x.
- *Synchronization operations*: When threads synchronize using a synchronization mechanism, a thread t_i executes a notification action, which is then observed by other threads t_j. Such a notification–observation pair defines a "synchronizes-with" edge from the former action to the latter. We classify actions that serve as sources and sinks of a synchronizes-with edge as synchronization source and sink actions, respectively.

– *Synchronization source actions*: sync-source(t, o) by thread t creates a synchronizes-with source by writing to a synchronization variable o. Lock releases and volatile variable writes in Java are synchronization source actions.
– *Synchronization sink actions*: sync-sink(t, o) by thread t creates a synchronizes-with sink by reading from a synchronization variable o. Lock acquires and volatile variable reads in Java are synchronization sink actions.

Multithreaded Executions: An execution E is represented by a tuple $E = \langle Tid, Act, W, \xrightarrow{po}., \xrightarrow{so}.\rangle$.

- Tid is the set of identifiers for threads involved in the execution. Each newly forked thread is given a new unique id from Tid.
- Act is the set of actions that occur in this execution. $Act|_t$ is the set of actions performed by $t \in Tid$, and $Act|_x$ (resp. $Act|_o$) are the sets of actions performed on data variable x (resp. synchronization variable o).
- W is a total function that maps each read(t, x, v) in Act to a write(u, x, v) in Act. W is used to model the write-seen relationship between a read of x and the write to x it sees. In a race-free, sequentially consistent execution, this is the last write before read(t, x, v). In order to make the function W total, we assume an initial write for each variable before any reads.
- $\xrightarrow{po}_t$ is the program order per thread t. For each thread t, $\xrightarrow{po}_t$ is a total order over $Act|_t$ and gives in which order the actions were issued to execute. This order is sometimes referred to as the observed execution order.
- $\xrightarrow{so}_o$ is the synchronization order per synchronization variable $o \in Sync$. For each $o \in Sync$, $\xrightarrow{so}_o$ is a total order over $Act|_o$.

Synchronizes-With and Happens-Before: Given an execution with program and synchronization orders, we extract two additional orders called the synchronizes-with ($\xrightarrow{sw}$) and happens-before ($\xrightarrow{hb}$) orders. Data races are defined using these orders.

A synchronization operation α_1 by thread t_1 *synchronizes with* α_2 by thread t_2, denoted $\alpha_1 \xrightarrow{sw} \alpha_2$, if α_1 is a sync-source on some synchronization variable o, α_2 is a sync-sink on o, and $\alpha_1 \xrightarrow{so}_o \alpha_2$.

The *happens-before* partial order $\xrightarrow{hb}$ on the execution E is defined as the transitive closure of the program orders $\xrightarrow{po}_t$ for all $t \in Tid$ and the synchronizes-with order $\xrightarrow{sw}$.

In this paper, we focus only on *well-formed* executions,[10] which respect (i) the intra-thread semantics and (ii) the semantics of the synchronization variables and operations. In addition, well-formed executions satisfy two essential requirements for data-race detection:

- *Happens-before consistency*: This property makes use of the happens-before order to restrict the write-seen relationship. For example, for a read action α, $\alpha \xrightarrow{hb} W(\alpha)$ cannot happen, and $W(\alpha)$ cannot be overwritten by another write action β such that $W(\alpha) \xrightarrow{hb} \beta \xrightarrow{hb} \alpha$.
- The union of the program orders for all $t \in Tid$ and the synchronization orders for all variables $o \in Sync$ is a valid partial order. During an execution, our data-race detection algorithm examines a linearization of this partial order and identifies the happens-before edges between data accesses.

Sequential Consistency: Sequential consistency is a property that allows programmers to use an interleaving model of execution where accesses from different threads are interleaved into a total order, and every read sees the value of the most recent write. Sequential consistency is widely considered to be the only simple-enough model with which writing useful concurrent programs is practical. Formally, an execution $E = \langle Tid, A, W, \xrightarrow{po}., \xrightarrow{so}.\rangle$ is *sequentially consistent* if there exists a total order $\xrightarrow{SC}$ over Act satisfying the following:

- For every thread $t \in Tid$, $\xrightarrow{SC}$ respects the program order $\xrightarrow{po}_t$, i.e., $\xrightarrow{po}_t \subseteq \xrightarrow{SC}$.
- Every read α = read(x) sees the most recent write to x in $\xrightarrow{SC}$, i.e., there is no other β = write(x) such that $W(\alpha) \xrightarrow{SC} \beta \xrightarrow{SC} \alpha$.

Data Races: Two data variable accesses are called *conflicting* if they refer to the same shared data variable and at least one of them is a write access.

One frequently used definition of a race condition involves a program state in which two conflicting accesses by two different threads to a shared data variable are simultaneously enabled. To distinguish this definition from others, let us refer to this condition as a *simultaneity race*. The definition of a race condition used in most work on dynamic race detection is what we call a *happens-before race* and involves two conflicting accesses not ordered by the happens before relationship, i.e., not separated by proper synchronization operations. For C++, these two definitions of a race condition have been shown to be equivalent.[2] This proof also generalizes to Java executions. Formally, an execution $E = \langle Tid, Act, W, \xrightarrow{po}, \xrightarrow{so}.\rangle$ contains a happens-before race if there are two conflicting actions, $\alpha, \beta \in Act|_x$ accessing a data variable x, such that neither $\alpha \xrightarrow{hb} \beta$ nor $\beta \xrightarrow{hb} \alpha$ holds. Conversely, the execution is *race free* if every pair of conflicting accesses to a data variable are ordered by happens-before.

The well-formedness of an execution guarantees that if the execution has no race conditions, then it is sequentially consistent. The GOLDILOCKS runtime makes use of this and the `DataRaceException` to guarantee for all programs (whether racy or not) that every concurrent execution is sequentially consistent at the byte-code level. This does not restrict the GOLDILOCKS runtime's use as a debugging tool, because, for the Java and C++ memory models, it has been proven[2, 10] that if a program has a racy execution, then it is guaranteed to have at least one execution that is sequentially consistent and racy. Thus, it is sufficient to restrict one's attention to looking for races in sequentially consistent executions only.

3. THE GOLDILOCKS ALGORITHM

In this section, we describe our algorithm for detecting data races in an execution $E = \langle Tid, Act, W, \xrightarrow{po}., \xrightarrow{so}.\rangle$. For simplicity of exposition, we initially do not distinguish between read and write accesses.

The GOLDILOCKS algorithm processes the actions in *Act* one at a time, as a sequence. Before a thread t performs an action α in *Act*, t notifies the GOLDILOCKS algorithm that α is about to occur. The order in which these notifications from different threads are interleaved and processed by GOLDILOCKS is represented mathematically by π, where $\pi(i)$ is the i-th action in the sequence. This linear order, by construction, respects the program order for each thread, and the synchronization total order for each synchronization variable.[b]

The GOLDILOCKS algorithm maintains for each data variable a "Goldilockset": a map *GLS* such that, for every data variable x, its Goldlilockset is a set $GLS(x) \subseteq Tid \cup Sync$. Roughly speaking, $GLS_i(x)$, the Goldilockset of x immediately before processing action $\pi(i)$, consists of (*i*) the id's of threads that can access x in a race-free way at that point in the execution, and (ii) the synchronization variables on which a thread can perform a sync-sink access in order to gain race-free access to x.

As GOLDILOCKS processes each action $\pi(i)$ from E, it updates the Goldilocksets of variables. Initially, the Goldilockset $GLS(x)$ is empty for all data variables, including static ones. When a new object is created, the Goldilockset for all of its instance fields is initialized to the empty set. After every action, the Goldilockset of every data variable x is potentially updated. For every data variable x, three simple rules specify how $GLS(x)$ is updated after $\pi(i)$ based on whether $\pi(i)$ is (1) a synchronization source, (2) a synchronization sink, or (3) a read or write access to x, as shown in the procedure *ApplyLocksetRules* in Figure 2.

If the action $\pi(i)$ is a synchronization operation on a variable o, we update the lockset $GLS(x)$ for every data variable x in *Data*. If $\pi(i)$ is a sync-source operation, rule 1 adds o to $GLS(x)$ if it contains the id t of the current accessor thread. Intuitively, this represents that a later sync-sink operation by a thread u on synchronization variable o will be sufficient for u to gain race-free access to x. This is formalized by rule 2. If $\pi(i)$ is a sync-sink(o) operation, rule 2 checks whether the synchronization variable o is in $GLS(x)$. If this is the case, then t is added to the Goldilockset.

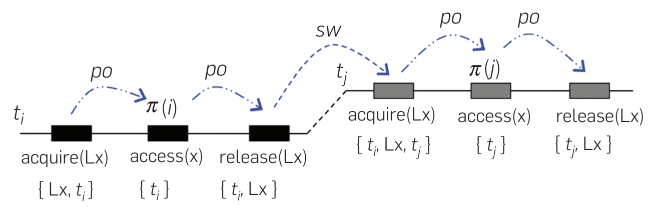
If the action $\pi(i)$ is an access to a data variable x, rule 3 checks the Goldilockset of the variable $GLS(x)$ to decide whether this access is race free. If $GLS(x)$ is empty, it indicates that x is a fresh variable which has not been accessed so far and any access to x at this point is race free. If $GLS(x)$ is not empty, only threads whose id's are in $GLS(x)$ can perform race-free accesses to x. If the accessing thread's id t is not in $GLS(x)$ then we throw a `DataRaceException` on x. Otherwise, the access to x is race free and $GLS(x)$ becomes the singleton $\{t\}$, indicating that, without further synchronization operations, only t can access x in a race free manner.

Figure 1 shows two cases where the ownership of a data variable x is transferred from a thread t_i to another thread t_j, and indicates how the Goldilocksets evolve in each case. Program order ($\xrightarrow{po}$) and synchronizes-with ($\xrightarrow{sw}$) edges between consecutive actions are indicated in the figure. Figure 1a illustrates direct ownership transfer from t_i to t_j. After accessing x, t_i performs a sync-source operation (lock release) on synchronization variable `Lx`. Later, t_j obtains ownership of x by executing a sync-sink operation (lock acquire) on synchronization variable `Lx`. Figure 1b illustrates transitive ownership transfer. Threads t_i and t_j do not synchronize on the same synchronization variable. Instead, the synchronization involves a chain of synchronizes-with edges between other threads and synchronization variables. t_i synchronizes with t_k via synchronization variable `o1` and, later t_k synchronizes with t_j via synchronization variable `o2`.

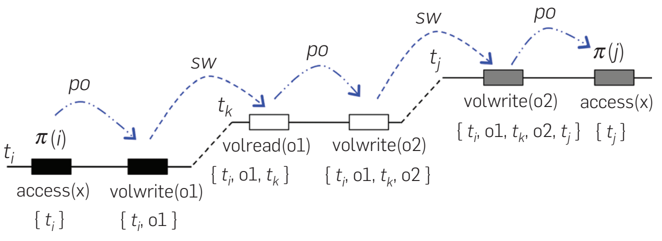
Rules 1 and 2 in Figure 2 require updating the lockset of each data variable. A naive implementation of this algorithm would be too expensive for programs that manipulate large heaps. In Section 4, we present an efficient way of implement our algorithm by representing Goldilocksets implicitly and by applying update rules lazily.

b A racy read may appear earlier in π than the write that it sees. If an execution contains a data race between a pair of accesses, GOLDILOCKS declares a race at one of these accesses regardless of which linearization π is picked.

Figure 1. Transferring ownership of *x*, and *GLS*(*x*).



(a) Direct ownership transfer using lock *Lx*



(b) Transitive ownership transfer using volatiles o1 and o2

Figure 2. The core lockset update rules.

```
ApplyLocksetRules(π(i)):

1. if π(i) = sync-source(t, o)
       foreach x ∈ Data:
           if t ∈ GLS(x)
               GLS(x) := GLS(x) ∪ {o}

2. if π(i) = sync-sink(t, o)
       foreach x ∈ Data:
           if o ∈ GLS(x)
               GLS(x) := GLS(x) ∪ {t}

3. if π(i) = write(t, x) or π(i) = read(t, x)
       if t ∈ GLS(x) or GLS(x) = ∅
           GLS(x) := {t}
       else
           throw a DataRaceException on x
```

Correctness: The following theorem expresses the fact that the Goldilocks algorithm is both *sound*, i.e., detects all actual races in a given execution, and *precise*, i.e., never reports false alarms. The full proof of the original Goldilocks algorithm for Java can be found in Elmas et al.[5]

Theorem 1 (Correctness). *Let* $E = \langle Tid, Act, W, \xrightarrow{po}., \xrightarrow{so}. \rangle$ *be a well-formed execution, x a data variable, and π a linear order on Act as described earlier. Let* $i < j$, *and let π(i) and π(j) be two accesses to x performed by threads* t_i *and* t_j, *with no other action π(k) in between* $(i < k < j)$ *accessing x. Then* $t_j \in GLS_j(x)$, *i.e., the access π(j) is declared to be race free by the* Goldilocks *algorithm iff* $\pi(i) \xrightarrow{hb} \pi(j)$.

3.1. Example: precise data-race detection

In this section, we illustrate on an example the Goldilocks algorithm and how Goldilocksets capture the synchronization mechanism protecting access to a variable at each point in an execution. In this example, earlier lockset algorithms would have erroneously declared a race condition.

Consider the execution shown in Figure 3 in which all actions of `T1` happen first, followed by all actions of `T2` and then of `T3`. This example mimics a scenario in which an object is created and initialized and then made visible globally by `T1`. This `Int` object (referred to as *o* from now on) is a container object for its `data` field (referred to as *x* from now on). The object *o* is referred to by different global variables (`a` and `b`) and local variables (`tmp1`, `tmp2`, and `tmp3`)

Figure 3. Precise data-race detection example.

```
Class Int {  int data; }
Int a, b; // Global variables
```

Execution	Goldilockset update rule applied	$GLS(x)$
Thread 1 (`T1`):		
`tmp1 = new Int;`	Initialize lockset	$\emptyset$
`tmp1.data = 0;`	First access	{`T1`}
`acquire(La);`	`La` $\in GLS(x) \rightarrow$ *add* `T1` *to* $GLS(x)$	{`T1`}
`a = tmp1;`	No access to *x*	{`T1`}
`release (La);`	`T1` $\in GLS(x) \rightarrow$ *add* `La` *to* $GLS(x)$	{`T1,La`}
Thread 2 (`T2`):		
`acquire(La);`	`La` $\in GLS(x) \rightarrow$ *add* `T2` *to* $GLS(x)$	{`T1,La,T2`}
`tmp2 = a;`	No access to *x*	{`T1,La,T2`}
`release(La);`	`T2` $\in GLS(x) \rightarrow$ *add* `La` *to* $GLS(x)$	{`T1,La,T2`}
`acquire(Lb);`	`Lb` $\in GLS(x) \rightarrow$ *add* `T2` *to* $GLS(x)$	{`T1,La,T2`}
`b = tmp2;`	No acces s to *x*	{`T1,La,T2`}
`release(Lb);`	`T2` $\in GLS(x) \rightarrow$ *add* `Lb` *to* $GLS(x)$	{`T1,La,T2,Lb`}
Thread 3 (`T3`):		
`acquire(Lb);`	`Lb` $\in GLS(x) \rightarrow$ *add* `T3` *to* $GLS(x)$	{`T1,La,T2,Lb,T3`}
`b.data = 2;`	`T3` $\in GLS(x) \rightarrow$ Race-free access	{`T3`}
`tmp3 = b;`	No access to *x*	{`T3`}
`release(Lb);`	`T3` $\in GLS(x) \rightarrow$ *add* `Lb` *to* $GLS(x)$	{`T3,Lb`}
`tmp3.data = 3;`	`T3` $\in GLS(x) \rightarrow$ Race-free access	{`T3`}

at different points in this execution. The contained variable *x* is protected by synchronization on the container object *o*, and during the execution, the lock (`La` or `Lb`) protecting *o* and *x* changes depending on which variable (`a` or `b`) points to *o*. Notice that, `T2` changes the protecting lock of the container object *o* from `La` to `Lb`, without accessing *x*. Figure 3 shows the Goldilocks update rules applied on *GLS*(*x*) for each action and the resulting value of *GLS*(*x*).

Observe that the update rules allow a variable's Goldilockset to grow during the execution. This enables them to represent threads transfering ownership using different synchronization variables during the execution. In this example, this ownership transfer takes place without the variable itself being accessed. For example, after `T2` finishes in Figure 2, *GLS*(*x*) has the value {`T1`, `La`, `T2`, `Lb`}, meaning that a thread can access *x* without data race by locking either `La` or `Lb`. Then `T3` makes `Lb` the only protecter lock by acquiring `Lb` and accesses *x*.

3.2. Distinguishing read and write accesses

The basic Goldilocks algorithm in Figure 2 tracks the happens-before relationship between any two accesses to a variable *x*. In order to perform race detection, we must check the happens-before relationship only between conflicting actions, i.e., at least one action in the pair must be a write access. We extend the basic Goldilocks algorithm by keeping track of (i) $GLS^W(x)$, the "write Goldilockset of *x*", and (ii) $GLS^R(t, x)$, the "read Goldilockset of *t* and *x*" for each thread *t*. The update rules in *ApplyLocksetRules* are adapted to maintain these Goldilocksets, but have essentially the same form as the rules in Figure 2. In the extended algorithm, it is sufficient to check happens-before between the current access to *x* and the most recent accesses (in the linear order *π*) to *x*. How this extension is performed for Java can be found in Elmas et al.[6]

3.3. Specializing Goldilocks to the JMM

The JMM requires that *all* synchronization operations be ordered by a total order $\xrightarrow{so}$, whereas in our execution model, a separate total order $\xrightarrow{so}_o$ per synchronization variable is sufficient.

Data Variables and Operations: In Java, every data variable is in the form of (*o*, *d*) where *o* is an object and *d* is a nonvolatile field. The byte-code instructions *x load* and *xstore* access memory to read from and write to fields of objects, respectively (*x* changes depending on the type of the field).

The JMM specifies three synchronization mechanisms: monitors, volatile fields, and fork/join operations.

Monitors: In Java, a monitor per object (denoted by m_o) provides a reentrant lock for each object *o*. Acquiring the lock of an object *o* (acquire(*o*)) corresponds to a sync-sink operation on m_o, while releasing the lock of *o* (release(*o*)) corresponds to a sync-source operation on m_o. Nested acquires and releases of the same lock are treated as no-ops. In the JMM, each release(*o*) synchronizes with the next acquire(*o*) operation in $\xrightarrow{so} m_o$.

Volatile Variables: Each volatile variable is denoted (*o*, *v*) where *o* is an object, and *v* is a volatile field. Each read volread(*o*, *v*) from a volatile variable (*o*, *v*), and each write volwrite(o, v) to (o, v) is implemented by the *xload* and *xstore* byte-code instructions, respectively. While volread(o, v) corresponds to a sync-sink, volwrite(o, v) corresponds to sync-source operation on (o, v). In the JMM, there is a synchronizes-with relationship between each volread(o, v) and the volwrite(o, v) that it sees.

Fork/Join: Creating a new thread with id t (fork(t)) synchronizes with the first action of thread t, denoted start(t). The last action of thread t, denoted end(t) synchronizes with the join operation on t, denoted join(t). For each thead t, fork(t) and end(t) correspond to sync-source operations on a (fictitious) synchronization variable $\bar{t}$, and start(t) and join(t) correspond to sync-sink operations on $\bar{t}$. The JMM guarantees that for each thread t, there exists an order $\xrightarrow{so}_{\bar{t}}$ such that: fork(t) $\xrightarrow{so}_{\bar{t}}$ start(t) $\xrightarrow{so}_{\bar{t}}$ end(t) $\xrightarrow{so}_{\bar{t}}$ join(t).

Handling other Synchronization Mechanisms: Using the lockset update rules above, GOLDILOCKS is able to uniformly handle various approaches to synchronization such as dynamically changing locksets, permanent or temporary thread-locality of objects, container-protected objects, ownership transfer of variable without accessing the variable (as in the example in Section 3.1). Furthermore, GOLDILOCKS can also handle the synchronization idioms in the `java.util.concurrent` package such as semaphores and barriers, since these primitives are built using locks and volatile variables. The happens-before edges induced by the wait/notify(All) construct are computed by simply applying the Goldilockset update rules to the acquire and release operations invoked inside wait.

3.4. Race detection and sequential consistency

The Java and C++ memory models provide the data-race freedom (DRF) property.[2, 10] The DRF property guarantees that if all sequentially consistent executions of a *source* program are race free, then the *compiled* program only exhibits these sequentially consistent executions of the source program, after any compiler and hardware optimizations permitted by the memory model. The GOLDILOCKS algorithm check races by monitoring the executions of the compiled program, and assumes that the compiler and the runtime it is built on (hardware or virtual machine) conform to the language and the memory model specifications. Therefore, if the source program is race free, then any execution of the compiled program corresponds to a sequentially consistent execution of the source program, and no `DataRaceException` is thrown.

If the source program has a race, the GOLDILOCKS runtime still ensures that all executions of the compiled program will run under the sequential consistency semantics, i.e., sequential consistency is guaranteed at the byte-code level. This is accomplished by preventing accesses that will cause a data race and throwing a `DataRaceException` right before that access. However, in the case of a racy program, the JMM permits compiler optimizations that result in executions that are not sequentially consistent behaviors of the original source code. In this case, the JMM and the DRF property are not strong enough to allow the GOLDILOCKS runtime to relate byte-code level executions to executions of the source-level program, which makes debugging hard.

To use GOLDILOCKS for debugging purposes, this difficulty can be remedied by disabling compiler optimizations. For post-deployment use, a stronger memory model[9, 11] that is able to relate each (racy and race-free) execution of the compiled program to a sequentially consistent execution of the source program is needed.

4. IMPLEMENTING GOLDILOCKS

There are two published implementation of the GOLDILOCKS algorithm, both of which monitor the execution at the Java byte-code level. At this level, each variable access or synchronization operation corresponds to a single byte-code instruction, and each byte-code instruction can be associated with a source code line and/or variable.

The first GOLDILOCKS implementation, by the authors of this paper, was carried out in Kaffe,[19] a clean-room implementation of the Java virtual machine (JVM) in C. In Kaffe, we integrated GOLDILOCKS into the interpreting mode of Kaffe's runtime engine. Implementing the algorithm in the JVM enables fast access to internal data structures of the JVM that manage the layout of object in the memory and using the efficient mechanisms that exist in the JVM internally, such as fast mutex locks.

The second implementation of GOLDILOCKS is by Flanagan and Freund and was carried out using the ROADRUNNER dynamic program analysis tool.[8] In ROADRUNNER, GOLDILOCKS is implemented in Java and injected by byte-code instrumentation at load-time of the program. This allows the algorithm to benefit from Java compiler optimizations and just-in-time compilation and to be portable to any JVM. Flanagan and Freund showed that this implementation is competitive with ours in Kaffe for most of the common benchmarks.[7]

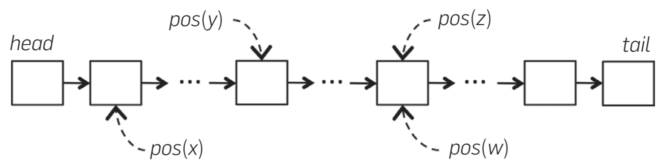
In the following, we present the most important implementation features and optimizations. The implementation is described based on the core algorithm presented in Figure 2. The extension of the implementation that distinguishes read and write accesses can be found in Elmas et al.[6]

4.1. Implicit representation and lazy evaluation of Goldilocksets

For programs with a large number of data variables, representing Goldilocksets explicitly for each data variable and implementing the GOLDILOCKS algorithm as described in Figure 2 may have high memory and computational cost. We avoid the memory cost by representing the Goldilocksets *implicitly* and the computational cost by evaluating Goldilocksets *lazily* as described below.

Instead of keeping a separate Goldilockset $GLS(x)$ for each variable x, we represent $GLS(x)$ implicitly as long as no access to x happens and is computed temporarily when an access happens. At this point, the Goldilockset is a singleton, and we continue to represent it implicitly until the next access. For this, we keep the synchronization events in a single, global linked list called the *synchronization-event list* and represent by its *head* and *tail* pointers in Figure 4. The ordering of these events in the list is consistent with the program order $\xrightarrow{po}_t$ for each thread t and the synchronization orders $\xrightarrow{so}_o$ for

Figure 4. The synchronization event list.



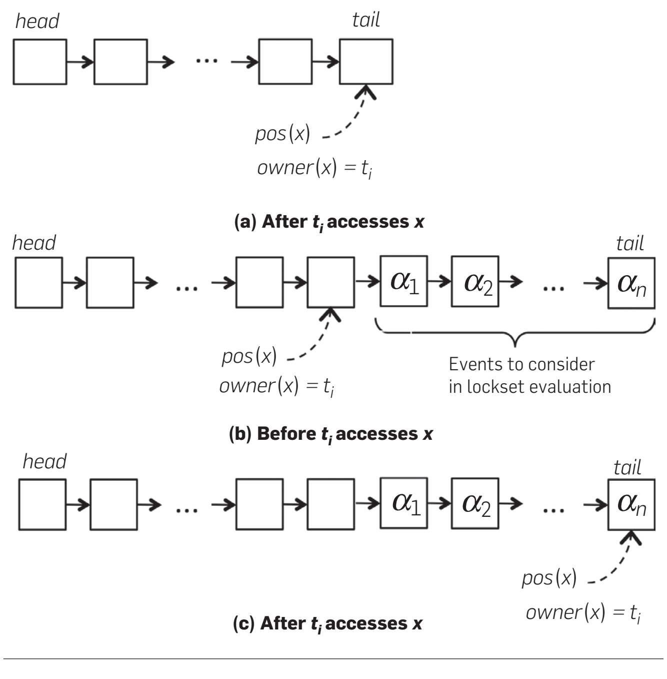
each synchronization variable o.[c] When a thread performs a synchronization action α, it must append a corresponding event to the synchronization-event list atomically with the event. In Kaffe, we make sure this is the case by modifying the implementations of the Java synchronization actions.

In order to represent $GLS(x)$, each variable x in the program is associated with two bits of information regarding the most recent access to x: $owner(x)$ stores the id of the thread that most recently accessed x, and $pos(x)$ points to the last synchronization event in the list that was taken into account when $GLS(x)$ was last computed.

Figure 4 shows four variables pointing to entries in the synchronization-event list. Figure 5 shows how the Goldilockset $GLS(x)$ is computed when x is accessed.

5(a): After each access to x by a thread t_i, $owner(x)$ is set to t_i, and $pos(x)$ is set to point the *tail* of the synchronization event list.
5(b): Right before an access to x by thread t_j, temporarily, we represent $GLS(x)$ explicitly. $GLS(x)$ is initially $\{owner(x)\}$ and is updated by processing the synchronization events between $pos(x)$ (denoted by $\alpha_1, \ldots, \alpha_n$) and *tail* according to the rules 1 and 2 of Figure 2. This process stops either when t_j is added to $GLS(x)$ or the last event (α_n) is processed. In the former case, no race is reported according to the rule 3 of Figure 2. In the latter case, a race is reported since $t_j \notin GLS(x)$ after the evaluation.
5(c): After the check, $owner(x)$ is set to t_j and $pos(x)$ is set to the *tail* of the synchronization event list.

Figure 5. Lazy evaluation of Goldilockset *GLS*(*x*).



[c] For Java, there is a total order on all synchronization operations, and the entries in the list are in this order.

The implementation does not use any extra threads for race detection. The algorithm is performed in a decentralized manner by instrumented threads of the program being checked. For each data variable x, we use a unique lock to make atomic the Goldilockset update and the race-freedom check for each access to x and to serialize all the race-freedom checks for x.

4.2. Performance optimizations

Short-Circuit Checks: A cheap-to-evaluate sufficient condition for a happens-before edge between the last two accesses to a variable can reduce race-detection overhead. We make use of two such conditions, called short-circuit checks, and bypass the traversal of the synchronization event list when these checks succeed. In this case, the final Goldilockset of the variable consists of the id of the thread that accessed it last.

We employ two constant-time short-circuit checks. First, when the last two accesses to a shared variable are performed by the same thread t, the happens-before relationship is guaranteed by the program order of t. This is detected by checking whether $owner(t)$, the last accessor thread, is the same as the thread performing the current access.

In the second short-circuit check, we determine whether the variable x is protected by the same lock during the last two accesses to x. For this, we associate with each variable x a lock $alock(x)$, which is randomly selected among the locks held by the most recent accessor thread. When a thread t accesses x and if $alock(x)$ is held by t, then that access is race free.

Direct Ownership Transfer: A sound but imprecise third optimization is to consider only the subset of synchronization events executed by the current and last accessing thread when examining the portion of the synchronization event list between $pos(x)$ and *tail*. This check is not constant time, but we found that it succeeds often enough to improve GOLDILOCKS overhead.

Garbage Collection: The synchronization events list is periodically garbage-collected when there are entries in the beginning of the list that are not relevant for the Goldilockset computation of any variable. This is the case when an entry in the list is not reachable from $pos(x)$ for any data variable x, and is tracked by maintaining incremental reference counts for each list entry.

Partially Eager Evaluation: Sometimes the synchronization event list gets too long and it is not possible to garbage-collect the event list when variable x is accessed early in an execution but is not used afterwards. We address this problem by "partially eager" Goldilockset evaluation. We move $pos(x)$ forward towards the *tail* to a new position $pos'(x)$, and partially evaluate a Goldilockset $GLS(x)$ of x by processing events (i.e., running *ApplyLocksetRules* on them) between *pos*(*x*) and *pos*′(*x*). During the next access the evaluation of *GLS*(*x*) starts from the stored Goldilockset, not from {*owner*(*x*)}.

Sound Static Race Analysis: The runtime overhead of race detection is directly related to the number of data variable accesses checked and synchronization events that occur. To reduce the number of accesses checked at runtime, we use static analysis at compile time to determine accesses that are guaranteed to be race free. While implementing Goldilocks in Kaffe, we worked with two static analysis tools for this purpose: Chord[13] and RccJava.[1]

4.3. Race-detection overhead

At the time of the original Goldilocks work, the vector clock algorithm[12] was the only precise dynamic-race-detection algorithm in the literature. The vector clock algorithm, for an execution with *n* threads, requires for every thread and synchronization variable a separate vector clock (VC) of size *n* and performs *O*(*n*) operations (merging or comparing two VCs) whenever a synchronization operation or data access happens. In preliminary research, compared to a straightforward implementation of vector clocks, we found Goldilocks overhead to be significantly less.[6]

In Elmas et al.,[6] we measured the overhead of the Goldilocks implementation inside Kaffe on a set of widely used Java benchmarks. This implementation required us to run all programs in interpreted (not just-in-time compiled) mode. We found that, with powerful static analysis tools eliminating much of the monitoring, we were able to obtain a slowdown of within approximately 2 for all benchmarks. Without static elimination of some checks, overheads remained high; some benchmarks experienced slowdowns of over 15. The overhead results with static pre-elimination were encouraging in that they showed precise race detection to be a practical debugging tool, and they indicated that, with further optimizations, post-deployment runtime race detection to support `DataRaceDetection` could be viable.

Later work on FastTrack,[7] a dynamic race detector based on vector clocks, is able to avoid worst-case performance of vector clocks much of the time using optimizations for common cases. Flanagan and Freund[7] compare a number of race-detection algorithms, including just-in-time compiled implementations of FastTrack and Goldilocks in RoadRunner. FastTrack achieves significantly better overheads than both implementations of Goldilocks. The low overheads achieved by FastTrack provide further support that a practical race-aware runtime for deployed programs supporting a `DataRaceException` can be built. It is reported in Flanagan and Freund[7] that additional short-circuit checks similar to ones we discussed above dramatically reduce the runtime of FastTrack. Most of these checks can be incorporated into Goldilocks implementations as well.

5. CONCLUSION

We have presented a race-aware runtime for Java incorporating a novel algorithm, Goldilocks, for precise dynamic race detection. The runtime provides a `DataRaceException`, and thus ensures that executions remain sequentially consistent at the byte-code level. Experiments with Goldilocks have demonstrated that the runtime overhead of supporting a `DataRaceException` can be made reasonable.

Acknowledgments

We would like to thank Hans Boehm, Cormac Flanagan, Steve Freund, and Madan Musuvathi for their critique of this paper. This research was supported by the Software Reliability Research Group at Microsoft Research, Redmond, WA, by the Scientific and Technical Research Council of Turkey (TUBITAK) under grant 104E058, and by the Turkish Academy of Sciences (TUBA).

References

1. Abadi, M, Flanagan, C., Freund, S.N. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.* (2006).
2. Boehm, H.-J., Adve, S.V. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conf. on Programming Language Design and Implementation* (PLDI 2008)
3. Cheng, G.-I., Feng, M., Leiserson, C.E., Randall, K.H., Stark, A.F. Detecting data races in Cilk programs that use locks. In *ACM Symposium on Parallel Algorithms and Architectures* (SPAA 1998).
4. Christiaens, M. De Bosschere, K. TRaDe: Data race detection for Java. *Proc. Intl. Conference on Computational Science*. V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, and C. Tan, eds. (ICCS 2001).
5. Elmas, T., Qadeer, S., Tasiran, S. Goldilocks: Efficiently computing the happens-before relation using locksets. *Technical Report MSR-TR-2006–163, Microsoft Research* (2006).
6. Elmas, T., Qadeer, S., Tasiran, S. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2007).
7. Flanagan, C., Freund, S.N. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2009).
8. Flanagan, C., Freund, S.N. The RoadRunner dynamic analysis framework for concurrent programs, In *ACM Workshop on Program Analysis for Software Tools and Engineering* (PASTE 2010).
9. Lucia, B., Ceze, L., Strauss, K., Qadeer, S., Boehm H.-J. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th International Symposium on Computer Architecture* (ISCA 2010).
10. Manson, J., Pugh, W., Adve, S.V. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL 2005).
11. Marino, D., Singh, A., Millstein, T., Musuvathi, M., Narayanasamy, S. DRFx: A simple and efficient memory model for concurrent programming languages. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2010).
12. Mattern, F. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms* (1988).
13. Naik, M., Aiken, A., Whaley, J. Effective static race detection for Java. In *Proceedings 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2006).
14. Pozniansky, E., Schuster, A. Multirace: Efficient on-the-fly data race detection in multithreaded C++ programs: Research articles. *Concurr. Comput. Pract. Exp.* (2007).
15. Ronsse, M., Bosschere, K.D. RecPlay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.* (1999).
16. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.* (1997).
17. Schonberg, E. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 1989).
18. Shavit, N., Touitou, D. Software transactional memory. In *Symposium on Principles of Distributed Computing* (1995).
19. Wilkinson, T. Kaffe: A JIT and interpreting virtual machine to run Java code. http://www.transvirtual.com (1998).
20. Yu, Y., Rodeheffer, T., Chen, W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (SOSP 2005).

Tayfun Elmas (telmas@ku.edu.tr), Koç University, Istanbul, Turkey.

Serdar Tasiran (stasiran@ku.edu.tr), Koç University, Istanbul, Turkey.

Shaz Qadeer (qadeer@microsoft.com), Microsoft Research, Redmond, WA.