Goldilocks: A Race-Aware Java Runtime

Tayfun Elmas  
Koç University  
İstanbul, Turkey  
telmas@ku.edu.tr

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

Serdar Tasiran  
Koç University  
İstanbul, Turkey  
stasiran@ku.edu.tr

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 a very 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

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 non-deterministic. For this and other reasons, both the Java Memory Model (JMM) [10] and the C++ Memory Model (C++MM) [2] explicitly indicate that there are no “benign races” and 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 racy program 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 [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 are. The Goldilocks runtime for Java provides a new exception, DataRaceException, 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 awareness of the programmer and allows him to explicitly handle them. When this exception is caught, one could terminate the program gracefully, or retry the accesses that caused the race. If the exception is not caught, the default action is to terminate the two threads that are involved in the race condition. 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 [11, 9].

To support a DataRaceException, a runtime must 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 [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 [14, 7], for example) and this idea appears viable today.

The Goldilocks algorithm is based on an intuitive, gen-
eral 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 ac-
quiring a lock in this lockset. A Goldilockset generalizes this
as follows. At each point in an execution, the Goldilockset
for a shared variable \( x \) may contain locks, volatile variables,
and thread ids. A thread can perform a race-free access to
\( x \) iff its thread id is in the Goldilockset, or if it first reads a
volatile variable or acquires a lock 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 up-
dated during the execution as synchronization operations are
performed. As a result, Goldilocksets are a compact, intu-
tive way to precisely represent the happens-before relation-
ship. 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 eas-
ily be generalized to handle other synchronization prim-
itives such as software transactions [18] and adapted to the
handle memory models of languages other than Java, such as
C++MM. To facilitate this, in this paper (differently from
[6]) we present GOLDILOCKS on a generic memory model and
then show how the algorithm can be specialized to JMM.

**Outline** In the rest of this section we discuss related work
on dynamic data race detection (Section 1.1). In Sec-
tion 2, we introduce a generic memory model that gen-
eralizes JMM, and we define race conditions in this model.
Section 3 presents the GOLDILOCKS dynamic race-detection
algorithm. Section 4 describes the implementation of the
algorithm and optimizations for reducing runtime overhead.
Section 5 concludes the paper.

### 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 algo-
rithm 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 pat-
terns 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 re-
lation [4, 15, 17] using vector clocks [12]. Hybrid tech-
niques [14, 20] combine lockset and happens-before analysis.
For example, RaceTrack [20] uses a basic vector clock algo-
rithm 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 re-
duce 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 knowl-
edge, 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 [6], the idea that programming platforms
should be able to guarantee sequential consistency for all
programs has gained significant support. [11] and [9] pro-
vide platforms with explicit memory model exceptions. [11]
and [9] define stronger but simpler contracts than JMM and
C++MM, which enable efficient hardware implementations
that support the memory model exceptions. We believe that
this is necessary to reduce overhead to levels acceptable for
deployed code.

### 2. A GENERIC MEMORY MODEL

In this section, we present generic a memory model and
express the Java Memory Model as a special case of it. This
generic model allows a uniform treatment of the various syn-
chronization constructs in Java. We also believe that mem-
ory 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 synchroniza-
tion variables (Sync). We use \( x \) and \( o \) to refer to data and
synchronization variables, respectively. Threads in a pro-
gram execute actions from the following categories:

- **Data variable accesses:** \( \text{read}(t, x, v) \) by thread \( t \) reads
  the current value \( v \) of a data variable \( x \), and \( \text{write}(t, x, v) \)
  by thread \( t \) writes the value \( v \) to \( x \).

- **Synchronization operations:** When threads synchro-
nize using a synchronization mechanism, a thread \( t_i \)
exeutes a notification action, which is then observed by
other threads \( t_j \). Such a notification-observation pair de-
defines a “synchronizes-with” edge from the for-
mer 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:** \( \text{sync-source}(t, o) \)
    by thread \( t \) creates a synchronizes-with source by
    writing to a synchronization variable \( o \). Lock re-
    leases and volatile variable writes in Java are syn-
    chronization source actions.

  - **Synchronization sink actions:** \( \text{sync-sink}(t, o) \)
    by thread \( t \) creates a synchronizes-with sink by
    reading from a synchronization variable \( o \). Lock ac-
    quires and volatile variable reads in Java are syn-
    chronization sink actions.

**Multithreaded Executions.** An execution \( E \) is repre-
sented by a tuple \( E = (Tid, Act, W, \frac{p_0}{\rightarrow}, \frac{s_0}{\rightarrow}) \).
• $\text{Tid}$ is the set of identifiers for threads involved in the execution. Each newly-forked thread is given a new unique id from $\text{Tid}$.

• $\text{Act}$ is the set of actions that occur in this execution. $\text{Act}_{|t}$ is the set of actions performed by $t \in \text{Tid}$, and $\text{Act}_{|\varnothing}$ (resp. $\text{Act}_{|\varnothing}$) are the sets of actions performed on data variable $x$ (resp. synchronization variable $o$).

• $W$ is the write-seen function on read accesses to data variables, so that $W(\text{read}(t, x, v))$ returns a write action $\text{write}(u, x, v)$ whose write is seen by the read action. We assume an initial write for each variable before any reads, so the function $W$ is total.

• $\overrightarrow{po}_{|t}$ is the program order per thread $t$. For each thread $t$, $\overrightarrow{po}_{|t}$ is a total order over $\text{Act}_{|t}$ and gives in which order the actions were issued to execute. This order is sometimes referred to as the observed execution order.

• $\overrightarrow{s{o-o}}_{|o}$ is the synchronization order per synchronization variable $o \in \text{Sync}$. For each $o \in \text{Sync}$, $\overrightarrow{s{o-o}}_{|o}$ is a total order over $\text{Act}_{|o}$.

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

A synchronization operation $\alpha_1$ by thread $t_1$ synchronizes with $\alpha_2$ by thread $t_2$, denoted $\alpha_1 \overrightarrow{sw} \alpha_2$, if $\alpha_1$ is a sync-source on some synchronization variable $o$, $\alpha_2$ is a sync-sink on $o$, and $\alpha_1 \overrightarrow{s{o-o}} \alpha_2$.

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

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 $\alpha$, $\alpha \overrightarrow{hb} W(\alpha)$ cannot happen, and $W(\alpha)$ cannot be overwritten by another write action $\beta$ such that $W(\alpha) \overrightarrow{hb} \beta \overrightarrow{hb} \alpha$.

• The union of the program orders for all $t \in \text{Tid}$ and the synchronization orders for all variables $o \in \text{Sync}$ 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. In this model, accesses from different threads are interleaved into a total order, and every read sees the value of the most recent write in this total order. Sequential consistency is widely considered to be the only simple-enough model with which writing useful concurrent programs is practical. Formally, an execution $E = \langle \text{Tid}, \text{Act}, W, \overrightarrow{po}_{|t}, \overrightarrow{s{o-o}}_{|o} \rangle$ is sequentially consistent if there exists a total order $\overrightarrow{SC}$ over $\text{Act}$ satisfying the following:

• For every thread $t \in \text{Tid}$, $\overrightarrow{SC}$ respects the program order $\overrightarrow{po}_{|t}$, i.e., $\overrightarrow{po}_{|t} \subseteq \overrightarrow{SC}$.

• Every read $\alpha = \text{read}(x)$ sees the most recent write to $x$ in $\overrightarrow{SC}$, i.e., there is no other $\beta = \text{write}(x)$ such that $W(\alpha) \overrightarrow{SC} \beta \overrightarrow{SC} \alpha$.

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 $E = \langle \text{Tid}, \text{Act}, W, \overrightarrow{po}_{|t}, \overrightarrow{s{o-o}}_{|o} \rangle$ contains a happens-before race if there are two conflicting actions, $\alpha, \beta \in \text{Act}_{|\varnothing}$ accessing a data variable $x$, such that neither $\alpha \overrightarrow{hb} \beta$ nor $\beta \overrightarrow{hb} \alpha$ 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 [10, 2] 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 $E = \langle \text{Tid}, \text{Act}, W, \overrightarrow{po}_{|t}, \overrightarrow{s{o-o}}_{|o} \rangle$. For simplicity of exposition, we initially do not distinguish between read and write accesses.

The Goldilocks algorithm processes the actions in $\text{Act}$ one at a time, as a sequence. Before a thread $t$ performs an action $\alpha$ in $\text{Act}$, $t$ notifies the Goldilocks algorithm that $\alpha$ is about to occur. The order in which these notifications from different threads are interleaved and processed by Goldilocks is represented mathematically by $\pi$, where $\pi(i)$ is the $i$-th action in the sequence. This linear order, by construction, respects the program order for each thread.

For a discussion of sequential consistency at the source code vs. the byte-code levels, and the requirements on Java compilers and runtimes in order to provide source-code-level sequential consistency, see Section 3.4.
regardless of which linearization
accesses, Goldilocks declares a race at one of these
potentially updated. For every data variable
all of its instance fields are initialized to the empty set. Af-
set dates the Goldilockset of variables. Initially, the Goldilock-
which a thread can perform a
in the execution, and (ii) the synchronization variables on
ately before processing action
Figure 1: Transferring ownership of \( x \), and \( GLS(x) \)

ApplyLocksetRules(\( \pi(i) \)):
1. if \( \pi(i) = \text{sync-source}(t, o) \)
   foreach \( x \in \text{Data} \):
   if \( t \in GLS(x) \)
   \( GLS(x) := GLS(x) \cup \{ o \} \)
2. if \( \pi(i) = \text{sync-sink}(t, o) \)
   foreach \( x \in \text{Data} \):
   if \( o \in GLS(x) \)
   \( GLS(x) := GLS(x) \cup \{ t \} \)
3. if \( \pi(i) = \text{write}(t, x) \) or \( \pi(i) = \text{read}(t, x) \)
   if \( t \in GLS(x) \) or \( GLS(x) = \emptyset \)
   \( GLS(x) := \{ t \} \)
else
   throw a DataRaceException on \( x \)

Figure 2: The core lockset update rules

and the synchronization total order for each synchronization
variable.

The Goldilocks algorithm maintains for each data variable \( x \), its Goldilockset is a set \( GLS(x) \subseteq \text{Tid} \cup \text{Sync} \).
Roughly speaking, \( GLS(x) \), the Goldilockset of \( x \) immedi-
ately 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 up-
dates the Goldilocksets of variables. Initially, the Goldilock-
set \( GLS(x) \) is empty for all data variables, including static
ones. When a new object is created, the Goldilockset for all of its
instance fields are initialized to the empty set. After
every action, the Goldilockset of every data variable \( x \) is
potentially updated. For every data variable \( x \), three sim-
ple rules specify how \( GLS(x) \) is updated after \( \pi(i) \) based on
whether \( \pi(i) \) is (1) a synchronization source, (2) a synchro-
nization 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 vari-
able \( x \), 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 indi-
cates 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 \((\text{pre})\) and synchronizes-with \((\text{sw})\) edges be-
tween consecutive actions are indicated in the figure. Fig. 1
(a) illustrates direct ownership transfer from \( t_i \) to \( t_j \). Af-
After accessing \( x \), \( t_i \) performs a sync-source operation (lock
release) on synchronization variable \( L_x \). Later, \( t_j \) obtains
ownership of \( x \) by executing a sync-source operation (lock ac-
quire) on synchronization variable \( L_x \). Fig. 1 (b) illustrates
transitive ownership transfer. Threads \( t_i \) and \( t_j \) do not syn-
chronize on the same synchronization variable. Instead, the
synchronization involves a chain of synchronizes-with edges
between other threads and synchronization variables. \( t_k \)
synchronizes with \( t_j \) via synchronization variable \( o_1 \) and, later
\( t_k \) synchronizes with \( t_i \) via synchronization variable \( o_2 \).

Rules 1 and 2 in Figure 2 require updating the lockset
of each data variable. A naive implementation of this algo-
rithm 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.

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.,
ever reports false alarms. The full proof of the original Gol-
dilocks algorithm for Java can be found in [5].

Theorem 1 (Correctness). Let \( E = (\langle Tid, Act, W, \text{po}, \text{so} \rangle) \) be a well-formed execution, \( x \) a data variable,
and \( \pi \) a linear order on Act as described earlier. Let \( i < j \),
and let \( \pi(i) \) and \( \pi(j) \) be two accesses to \( x \) performed by
threads \( t_i \) and \( t_j \), with no other action \( \pi(k) \) in between \((i < k < j) \) accessing \( x \). Then \( t_j \in GLS(x) \), i.e., the access
\( \pi(j) \) is declared to be race free by the Goldilocks algorithm
iff \( \pi(i) \rightarrow^{hh} \pi(j) \).
write access. We extend the basic happens-before relationship between any two accesses to a thread can access

3.2 Distinguishing Read and Write Accesses

Lb

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) $GLSW(x)$, the “write Goldilocks set of x”, and (ii) $GLS^R(t, x)$, the “read Goldilocks set of t and x” for each thread t. The update rules in ApplyLocksetRules are adapted to maintain these Goldilocks sets, 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 [6].

3.3 Specializing Goldilocks to the JMM

The Java Memory Model requires that all synchronization operations be ordered by a total order $\preceq$, whereas in our execution model, a separate total order $\preceq^o$ 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 non-volatile field. The byte-code instructions $x$load and $x$store 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 $m_o$. In the JMM, each release(o) synchronizes with the next acquire(o) operation in $\preceq^o$.

Volatile. 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 $x$load and $x$store 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, i.e., $join(t)$. For each thread 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 $\preceq^o_{\bar{t}}$ such that: $fork(t) \preceq_{\bar{t}} start(t) \preceq_{\bar{t}} end(t) \preceq_{\bar{t}} join(t)$.

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 wait/notify(All), and the synchronization idioms the java.util.concurrent package such as semaphores and barriers, since these primitives are built using locks and volatile variables.
3.4 Dynamic Race Detection and Sequential Consistency

The Java and C++ memory models provide the data-race freedom (DRF) property [10, 2]. 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 [11, 9] 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. Ideally, such strong memory models should be able to relate every race-free execution prefix of the compiled and executed program to a sequentially-consistent, race-free execution of the source code.

4. IMPLEMENTING GOLDILOCKS

There are two published implementation of the Goldilocks algorithm, both of which monitor the execution at the Java bytecode level. At this level, each variable access or synchronization operation corresponds to a single bytecode instruction, and each bytecode 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 bytecode 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 [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 →_{syn} for each thread t and the synchronization orders →_{syn} for each synchronization variable o. 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 lockset GLS(x) is computed using owner(x) and pos(x) 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_i, temporarily, we represent GLS(x) explicitly. GLS(x) is

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

![Figure 4: The synchronization event list](image)
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 GLS(x) or the last event (\(\alpha_n\)) 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 race check, owner(x) set to \(t_j\) and pos(x) is set to the tail of the synchronization event list.

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.

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 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 \(lock(x)\), which is randomly selected among the locks held by the most recent accessor thread. When a thread \(t\) accesses \(x\) and if \(lock(x)\) 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 saves on the runtime of lockset computation when the happens-before edge between pos(x) and tail is immediate, i.e., is accomplished by a direct transfer of ownership from one thread to another through a single synchronizes-with edge.

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. While running GOLDILOCKS on long executions, 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 the “partially-eager” lockset evaluation. We move pos(x) forward towards the tail to a new position pos'(x), and partially evaluate a lockset 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 existing static analysis methods 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 ReCJava [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 \(O(n)\) operation (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 [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 \texttt{DataRaceDetection} could be viable.

Later work on \textsc{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. Experiments in [7] compare a number of race-detection algorithms, including just-in-time compiled implementations of of \textsc{FastTrack} and \textsc{Goldilocks} in \textsc{RoadRunner} [8]. \textsc{FastTrack} achieves significantly better overheads than both implementations of \textsc{Goldilocks}. The low overheads achieved by \textsc{FastTrack} provide further support that a practical race-aware runtime for deployed programs supporting a \texttt{DataRaceException} can be built. It is reported in [7] that additional short-circuit checks similar to ones we discussed above dramatically reduce the runtime of \textsc{FastTrack}. Most of these checks can be incorporated into \textsc{Goldilocks} implementations as well.

5. CONCLUSION

We present a race-aware runtime for Java, which incorporates a novel algorithm, \textsc{Goldilocks}, for precise dynamic race detection. The runtime provides a \texttt{DataRaceException}, and thus ensures that executions remain sequentially consistent at the bytecode level. Experiments with \textsc{Goldilocks} have demonstrated that the runtime overhead of supporting a \texttt{DataRaceException} can be made reasonable. With improvement of static analysis techniques, further optimizations, and hardware support, we believe that the runtime overhead can be made acceptable for continuous monitoring of program execution during debugging and after deployment.

6. ACKNOWLEDGMENTS

This research was supported by a gift from 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).

7. REFERENCES