Goldilocks: A Race and Transaction-Aware Java Runtime

Tayfun Elmas
Koc University, Istanbul, Turkey
telmas@ku.edu.tr

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

Serdar Tasiran
Koc University, Istanbul, Turkey
stasiran@ku.edu.tr

Abstract

Data races often result in unexpected and erroneous behavior. In addition to causing data corruption and leading programs to crash, the presence of data races complicates the semantics of an execution which might no longer be sequentially consistent. Motivated by these observations, we have designed and implemented a Java runtime system that monitors program executions and throws a `DataRaceException` when a data race is about to occur. Analogous to other runtime exceptions, the `DataRaceException` provides two key benefits. First, accesses causing race conditions are interrupted and handled before they cause errors that may be difficult to diagnose later. Second, if no `DataRaceException` is thrown in an execution, it is guaranteed to be sequentially consistent. This strong guarantee helps to rule out many concurrency-related possibilities as the cause of erroneous behavior. When a `DataRaceException` is caught, the operation, thread, or program causing it can be terminated gracefully. Alternatively, the `DataRaceException` can serve as a conflict-detection mechanism in optimistic uses of concurrency.

We start with the definition of data-race-free executions in the Java memory model. We generalize this definition to executions that use transactions in addition to locks and volatile variables for synchronization. We present a precise and efficient algorithm for dynamically verifying that an execution is free of data races. This algorithm generalizes the Goldilocks algorithm for data-race detection by handling transactions and providing the ability to distinguish between read and write accesses. We have implemented our algorithm and the `DataRaceException` in the Kaffe Java Virtual Machine. We have evaluated our system on a variety of publicly available Java benchmarks and a few microbenchmarks that combine lock-based and transaction-based synchronization. Our experiments indicate that our implementation has reasonable overhead. Therefore, we believe that in addition to being a debugging tool, the `DataRaceException` may be a viable mechanism to enforce the safety of executions of multithreaded Java programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification — formal methods, reliability, validation; D.2.5 [*Software Engineering*]: Testing and Debugging — debugging aids, diagnostics, error handling and recovery, monitors; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — mechanical verification

General Terms Algorithms, Languages, Reliability, Verification

Keywords Data-race Detection, Java Runtime, Runtime Monitoring, Software Transactions

1. Introduction

Data races in concurrent programs are often symptomatic of a bug and may have unintended consequences. Unanticipated data races may have non-deterministic effects and are not desired. Detection and/or elimination of race conditions has been an area of active research. This paper presents a novel approach to detecting data-race conditions in Java programs and makes the following technical contributions.

- We present Goldilocks: a novel, precise lockset-based dynamic race detection algorithm. Goldilocks, unlike previous variants of lockset algorithms, can uniformly and naturally handle all synchronization idioms such as thread-local data that later becomes shared, shared data protected by different locks at different points in time, and data protected indirectly by locks on container objects. This paper presents a generalized version of the Goldilocks race-detection algorithm [8] that adds to it two key capabilities: explicitly handling software transactions [22] as a high-level synchronization idiom, and distinguishing between read and write accesses.
- An implementation of the generalized Goldilocks algorithm was carried out inside the Kaffe Java virtual machine. This implementation incorporates two techniques that significantly enhance its performance: partially-lazy evaluation and implicit representation of locksets, and a sequence of cheap and sufficient checks for race freedom which allow bypassing of lockset computation.
- To reduce the runtime overhead of race detection, we apply existing sound static race analysis tools beforehand to determine and record in Java class files the variables and accesses that are guaranteed to be race free. The capability to skip checks on these accesses in a sound manner, combined with the implementation optimizations result in a precise dynamic race checker whose performance on benchmarks is competitive with or, in some cases, significantly better than existing dynamic race checkers for Java programs.
- We present the first formalization of race conditions for programs that use software transactions along with other Java synchronization primitives. The extended Goldilocks algorithm and its implementation in the Kaffe JVM explicitly handle transactions.
- Our Java runtime provides a new runtime exception, `DataRaceException`, that is thrown precisely when an access that causes an actual race condition is about to be executed. This brings races to the awareness of the programmer and allows him to explicitly handle them.

The Java memory model (JMM) precisely defines the semantics for executions including races. We view the primary purpose of this to be specifying constraints for Java compilers and virtual machines in order to provide sanity and security guarantees. Otherwise, the semantics of Java executions with races are very difficult to make use of. The JMM defines well-synchronized programs to be free of data races. We expect that programmers will strive to ensure that their programs are race free and want to operate under sequential consistency semantics. In this view, an actual race condition is analogous to an out-of-bounds array access or a null pointer dereference. The Java runtime presented in this paper raises a `DataRaceException` when an actual (not potential) race condition is about to occur. This provides a mechanism for the program to always operate under sequentially consistent semantics. If the programmer has provided exception handling code, the program continues execution with sequentially consistent semantics, otherwise the thread that threw the exception is terminated. It is up to the programmer to interpret the race condition as symptomatic of a bug and gracefully terminate an operation or the entire program and fix the bug, or, to make more optimistic use of the `DataRaceException` as a mechanism for conflict detection between threads, and, say, roll back the effects of the block of code that triggered the `DataRaceException`.

To support `DataRaceException`, a race detection algorithm needs to be precise, i.e., not produce false alarms or warnings about potential races. It is not acceptable to interrupt an execution because of a potential or false race. Our dynamic race detection algorithm generalizes Goldilocks, the first precise lockset-based algorithm for dynamic race detection. The lockset update rules in Goldilocks uniformly handle all synchronization disciplines. The preliminary version of Goldilocks, while being as precise as a vector-clock-based algorithm, was found to be comparable in performance overhead to a prototype implementation of a lockset-based algorithm from the literature [20]. In this paper, we generalize Goldilocks to distinguish between read and write accesses and to handle software transactions explicitly. We implemented this race detection algorithm as part of the Kaffe Java virtual machine [24] which now provides a precise `DataRaceException`. A number of performance optimizations and new implementation features improve the performance of Goldilocks significantly beyond what was reported in [8].

To reduce the runtime overhead, we apply static analyses [1, 17] to eliminate dynamic checks on variables that can be proved to be race free statically. With this pre-elimination, the overheads we obtained on Java benchmarks are competitive with or better than other dynamic race detection algorithms in the literature. These results indicate that the precision of our algorithm does not come at a performance cost. For many benchmark programs, we found the overhead of precise race detection with our approach to be low enough to be possibly tolerable even for deployed code. In other benchmarks and for programs in which performance is of utmost importance, the safety-performance trade-off provided by `DataRaceException` may not be acceptable. For these programs, our Java runtime can serve as a debugging tool that produces no false alarms.

We imagine, at least in the near future, that programmers will use software transactions [22] to manage only some of the accesses to shared data[1] and other Java synchronization primitives to manage the rest – possibly to some of the same variables that were accessed within transactions at other points in the execution. We provide the first formalization of race conditions for executions that use software transactions in addition to other Java synchronization primitives. We extended Goldilocks' lockset update rules in order to handle transactions and also extended our implementation in Kaffe to provide `DataRaceException` support for executions with software transactions. The nature of the lockset rules and the implementation made it possible to integrate this feature without significant restructuring.

We require that the semantics of software transactions to specify when a happens-before relationship exists between two transactions without reference to the particular transactions implementation. For instance, in this paper, we use the specification that there is a happens-before edge between transactions that access at least one common variable. Our formulation of race conditions facilitates a modular check for races and strong atomicity for programs that use software transactions. The implementer of software transactions verifies that the implementation provides the happens-before edges in the transactions specification. Our race checker assumes that these happens-before edges are implemented properly, and checks for race-freedom of all accesses, but disregards the internal operation of the transactions implementation. The transaction implementation is only required to provide a list of the shared variables accessed by each transaction and a commit point. If no `DataRaceException` is thrown, then sequential consistency and strong atomicity are guaranteed. Further, the specification rather than the implementation of transactions is used in order to make this check cheaper. We demonstrate this way of handling transactions in our runtime on a hand-coded transactional data structure.

Section 2 provides examples that motivate the new features of the Java runtime we built. Section 3 introduces our mathematical model of Java program executions and defines race conditions for executions that contain transactions. Section 4 presents our dynamic race-detection algorithm. Section 5 describes the implementation of the algorithm and optimizations for reducing runtime overhead. Section 6 presents results of experiments using our Java runtime.

2. Motivating examples

Example 1: This example demonstrates the use of `DataRaceException` to terminate gracefully a thread about to perform an access that would have caused a data race before the race leads to an error. Pseudocode for this example is given in Figure 1 and is adapted from the Apache ftp-server benchmark used in [17]. The `run()` and `close()` methods shown belong to an object representing an ftp connection and are executed by two separate threads. The thread executing the `run()` method, in a `do .. while` loop services requests issued by the user on the command line, one command per iteration of the loop. A time-out thread executes the `close()` method of connection objects that have been idle too long. In the original benchmark, which did not contain the `try ... catch` block for `DataRaceException`, it is possible for the time-out thread to run the `close()` method and set, for instance, the `m_writer` field of the connection object to `null` right before the other thread accesses it at line 10 of the `run()` method. This race condition causes a `NullPointerException`. Making `run()` a synchronized method would make it impossible for the time-out thread to execute `close()`, thus, in a correct implementation, finer-grain synchronization needs to be used.

In the modified code that uses `DataRaceException`, when the thread executing `run()` is about to access `m_writer` in line 10 after the unsynchronized access by the thread running `close()`, a `DataRaceException` is thrown. The `run()` thread catches this exception, stops processing commands on this connection, exits the `do...while` loop and allows the time-out thread to close the connection. Note that, although in this example a race condition has an immediately noticeable consequence, in other examples

[1] Because most existing transaction implementations do not support I/O and may require libraries to be re-compiled [4]

```
public void run() {
 // ...
 // Initialize connection
 try {
   do {
     String commandLine = m_reader.readLine();
     ...
     m_request.parse(commandLine);
     if(!hasPermission()) {
        m_writer.send(530, "permission", null);
        continue;
      }
      // execute command
      service(m_request, m_writer);
    } while(!m_isConnectionClosed);
 } catch (DataRaceException e) {
      // Error message: "Connection closed!"
      break;
 }
}

public void close() {

  synchronized(this) {
    if(m_isConnectionClosed)    return;
    m_isConnectionClosed = true;
  }
  ...
  m_request = null;
  m_writer = null;
  m_reader = null;
}
```

Figure 1. Example 1: Demonstrating use of `DataRaceException`

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

Thread 1:              Thread 2:       Thread 3:
-------------------    --------        -------------
tmp1 = new IntBox();   acq(ma);        acq(mb);
tmp1.data = 0;         tmp2 = a;       b.data = 2;
acq(ma)                rel(ma);        tmp3 = b;
a = tmp1;              acq(mb);        rel(mb);
rel(ma);               b = tmp2;       tmp3.data = 3;
                       rel(mb);
```

Figure 2. Example 2

the observable manifestation of the race condition may occur far enough in the execution to make it very difficult to trace its origin.

Example 2: To support `DataRaceException`, a precise and efficient dynamic race analysis is needed. Purely vector-clock-based algorithms are precise but typically computationally expensive [16]. Lockset-based algorithms are efficient, but are not precise. They check adherence to a particular synchronization discipline and declare false races when this discipline is violated. Example 2 (Figure 2) makes use of two idioms typically not handled by earlier lockset algorithms: objects that are local to different threads at different points in the execution, and objects protected by locks of container objects. The Goldilocks race detection algorithm declares no false alarms in this example while other lockset-based algorithms do.

Consider an execution in which all actions of `Thread 1` happen first, followed by all actions of `Thread 2` and then of `Thread` 3. This example mimics a scenario in which an object is created and initialized and then made visible globally by `Thread 1`. This `IntBox` object (referred to as "o" from now on) is a container object

```
Foo {int data; Foo nxt};

Thread 1:           Thread 2:                Thread 3:
------------------  -------------            ------------
Foo t1 = new Foo(); Foo iter;                Foo t3;
t1.data = 42;       atomic {                 atomic {
atomic {             for (iter = head;        t3 = head;
  t1.nxt = head;          iter != null;       head =
  head = t1;              iter = iter.nxt)     t3.nxt;
}                       iter.data = 0;       }
                    }                        t3.data++;
```

Figure 3. Example 3

```
public class Account {
  double bal;
  public synchronized withdraw(int amt) { bal -= amt;}}

Account savings, checking;

Thread 1:                    Thread 2:
-------------------------    ------------------------
atomic {                     checking.withdraw(42);
  savings.bal -= 42;
  checking.bal += 42;
}
```

Figure 4. Example 4

for its `data` field, and, is referred to by different global variables at different points in this execution. Furthermore, the contained variable o`.data` is protected by synchronization on the container object o, and ownership transfer of o`.data` from one thread to another sometimes takes place without the o`.data` field being accessed at all. How our algorithm correctly captures the absence of a race in this case is explained in Section 4.1.

Example 3: In this example (Figure 3), software transactions and thread-locality are used at different times in the execution to protect access to the same shared data. A race checking algorithm that does not take into account transactions as a synchronization primitive would declare a false race in this example. Here, `Foo` objects form a linked-list, and while a `Foo` object is a member of the linked list, access to it is managed by software transactions. A `Foo` object referred to by `t1` is created and initialized by `Thread 1`, during which time it is thread-local. Then, in an atomic transaction, it is added to the head of the linked list by `Thread 1`. While in the linked list, this `Foo` object is modified by the loop in `Thread 2`, which, in an atomic software transaction, modifies the `data` fields of all `Foo` objects in the list. `Thread 3` removes the `Foo` object from the list in an atomic transaction. After this, the object is local to `Thread 3`. Observe that it is possible to make this example more sophisticated, for instance, by having the `Foo` object be shared among threads and making it lock-protected after its removal from the list. A correct implementation of software transactions would create a happens-before relationship between the transactions in `Thread 1`, 2 and 3. A race checking algorithm that is not aware of these happens-before edges would falsely declare a race between the accesses to the `data` field of the `Foo` object in `Thread 1` and `Thread 3`. Section 4.2 explains how the generalized Goldilocks algorithm handles this example.

Example 4: In this example (Figure 4), shared data is either protected by an object lock or is accessed within a transaction, which might lead one to believe at first glance that there should be no race conditions. `checking` and `savings` are `Account` objects with a synchronized `withdraw` method. `Thread 1` contains a software transaction that transfers money from `savings` to `checking`. `Thread 2` simply performs a withdrawal using the synchronized withdraw method. Since the software transaction implementation might be using a mechanism other than the object locks on checking and savings to implement the atomic transaction, there is a potential race condition between Thread 1 and 2's accesses to checking.bal. This race condition should be signaled regardless of the synchronization mechanism used by the transaction implementation, since this mechanism should not be visible to the programmer. Observe that completely ignoring accesses inside transactions while performing dynamic race checking would overlook the race in this case.

3. Preliminaries

This section presents the formalism required to explain the Goldilocks algorithm in Section 4. The reader may skip ahead to Sections 4.1 and 4.2 for an informal understanding of Goldilocks' lockset update rules applied to Examples 2 and 3 in the previous section.

Tid represents the set of thread identifiers and $Addr$ represents the set of object identifiers. Each object has a finite collection of fields. $Field$ represents the set of all fields and is a union of two disjoint sets, the set $Data$ of *data* fields and the set $Volatile$ of *volatile* fields. A *data variable* is a pair (o, d) consisting of an object o and a data field d. A *synchronization variable* is a pair (o, v) consisting of an object o and a volatile field v. Each thread in a program executes a sequence of actions. Actions are categorized into the following kinds:

- $SyncKind =$
  $\{acq(o), rel(o) \mid o \in Addr\} \cup$
  $\{read(o,v), write(o,v) \mid o \in Addr \land v \in Volatile\} \cup$
  $\{fork(u), join(u) \mid u \in Tid\} \cup$
  $\{commit(R,W) \mid R, W \subseteq Addr \times Data\}$
- $DataKind =$
  $\{read(o,d) \mid o \in Addr \land d \in Data\} \cup$
  $\{write(o,d) \mid o \in Addr \land d \in Data\}$
- $AllocKind = \{alloc(o) \mid o \in Addr\}$
- $Kind = SyncKind \cup DataKind \cup AllocKind$

The action kind $alloc(o)$ represents allocation of a new object o. The action kinds $read(o, d)$ and $write(o, d)$ respectively read and write the data field d of an object o. A thread is said to *access* a variable (o, d) if it executes an action of kind either $read(o, d)$ or $write(o, d)$. Of course, other kinds of actions (such as arithmetic computation, function calls, etc.) also occur in a real execution of a Java program but these actions are irrelevant for our exposition of race conditions and have consequently been elided.

The action kinds $acq(o)$ and $rel(o)$ respectively represent a thread acquiring and releasing a lock on object o. We use a special field $l \in Volatile$ containing values from $Tid \cup \{null\}$ to model the semantics of an object lock. An action of kind $acq(o)$ being performed by thread t blocks until $o.l = null$ and then atomically sets $o.l$ to t. An action of kind $rel(o)$ being performed by thread t fails if $o.l \neq t$, otherwise it atomically sets $o.l$ to $null$. Although we assume non-reentrant locks for ease of exposition in this paper, our techniques are easily extended to handle reentrant locks. The action kind $commit(R, W)$ represents the committing of a transaction that reads and writes the sets of shared data variables R and W respectively. We do not allow transaction bodies to include synchronization, therefore $R, W \subseteq Addr \times Data$. The commit action is explained in more detail in Sections 4 and 5.

The action kinds $read(o, v)$ and $write(o, v)$ respectively represent a read of and a write to the volatile field v of an object o. An action of kind $fork(u)$ creates a new thread with identifier u. An action of kind $join(u)$ blocks until the thread with identifier u terminates. In this paper, when referring to an action, we sometimes only name its kind when the rest of the information is clear from the context.

An execution $\Sigma = (\sigma, \xrightarrow{eso})$ of a program consists of a function $\sigma : Tid \to \mathcal{N} \to Kind$ and a total order $\xrightarrow{eso}$ (extended synchronization order) on the set $\{(t, n) \mid \sigma(t, n) \in SyncKind\}$. If $commit$ actions are projected out of $\xrightarrow{eso}$, the remaining total order is required to be the synchronization order associated with the execution as given by the Java memory model. Since we view transactions as high-level synchronization operations, we include the $commit$ action for each transaction in the extended synchronization order to represent the ordering of the transaction with respect to other synchronization operations. We use σ to define the program order of thread t denoted by $\xrightarrow{po}_t$. The relation $\xrightarrow{po}_t$ is a total order on the set $\{t\} \times \mathcal{N}$ such that $(t, m) \xrightarrow{po}_t (t, n)$ iff $m < n$. The relation $\xrightarrow{eso}$ is a total order on the subset of actions that perform synchronization. The *extended synchronizes-with* partial order $\xrightarrow{esw}$ is defined to be the smallest transitively-closed relation that satisfies the following conditions:

- If $\sigma(t, m) = rel(o), \sigma(u, n) = acq(o)$, and $(t, m) \xrightarrow{eso} (u, n)$, then $(t, m) \xrightarrow{esw} (u, n)$.
- If $\sigma(t, m) = write(o, v), \sigma(u, n) = read(o, v)$, and $(t, m) \xrightarrow{eso} (u, n)$, then $(t, m) \xrightarrow{esw} (u, n)$.
- If $\sigma(t, m) = fork(u)$, then $(t, m) \xrightarrow{esw} (u, n)$ for all $n \in \mathcal{N}$.
- If $\sigma(t, m) = join(u)$, then $(u, n) \xrightarrow{esw} (t, m)$ for all $n \in \mathcal{N}$.
- If $\sigma(t, m) = commit(R, W)$, $\sigma(u, n) = commit(R', W')$, $(t, m) \xrightarrow{eso} (u, n)$, and $(R \cup W) \cap (R' \cup W') \neq \emptyset$, then $(t, m) \xrightarrow{esw} (u, n)$.

The last item above expresses an interpretation of software transactions in which a transaction synchronizes with (and happens before) another iff they access at least one common variable. In this view, two transactions that access disjoint sets of variables do not synchronize with each other.

The *extended happens-before relation*, denoted by $\xrightarrow{ehb}$, is a relation on the set $Tid \times \mathcal{N}$. It is the transitive closure of the union of $\xrightarrow{esw}$ with the program order $\xrightarrow{po}_t$ for each thread $t \in Tid$. There is an *extended race* on data variable (o, d) if there exist $t, u \in Tid$ and $m, n \in \mathcal{N}$ such that both $(t, m) \xrightarrow{ehb} (u, n)$ and $(u, n) \xrightarrow{ehb} (t, m)$ are false and one of the following conditions hold:

1. $\sigma(t, m) = write(o, d)$ and $\sigma(u, n) \in \{read(o, d), write(o, d)\}$
2. $\sigma(t, m) = write(o, d)$, $\sigma(u, n) = commit(R, W)$, and $(o, d) \in R \cup W$
3. $\sigma(t, m) = read(o, d)$, $\sigma(u, n) = commit(R, W)$, and $(o, d) \in W$

Observe that for executions that contain no transaction commit actions, the extended synchronization order, the extended synchronizes-with relation, and the extended happens-before relation coincide with the synchronization order, the synchronizes-with relation and the happens-before relation as defined in the Java memory model. The projection of the $\xrightarrow{eso}$ order onto the $commit$ actions corresponds to the atomic order of transactions as defined in [11]. Our treatment of transactions simply formalizes one possible semantics of transactions and does not introduce any extra synchronization or serialization to transaction implementations. In the transaction semantics we model using the extended happens-before relation, pairs of shared variable accesses where both accesses belong to some transaction are considered to be race-free.

```
1. $\sigma(t,n) \in \{read(o,d), write(o,d)\}$:
      if $LS(o,d) \neq \emptyset$ and $t \notin LS(o,d)$
         report data race on $(o,d)$
      $LS(o,d) := \{t\}$

2. $\sigma(t,n) = read(o,v)$:
      foreach $(o',d)$:
         if $(o,v) \in LS(o',d)$ add $t$ to $LS(o',d)$

3. $\sigma(t,n) = write(o,v)$:
      foreach $(o',d)$:
         if $t \in LS(o',d)$ add $(o,v)$ to $LS(o',d)$

4. $\sigma(t,n) = acq(o)$:
      foreach $(o',d)$:
         if $(o,l) \in LS(o',d)$ add $t$ to $LS(o',d)$

5. $\sigma(t,n) = rel(o)$:
      foreach $(o',d)$:
         if $t \in LS(o',d)$ add $(o,l)$ to $LS(o',d)$

6. $\sigma(t,n) = fork(u)$:
      foreach $(o',d)$:
         if $t \in LS(o',d)$ add $u$ to $LS(o',d)$

7. $\sigma(t,n) = join(u)$:
      foreach $(o',d)$:
         if $u \in LS(o',d)$ add $t$ to $LS(o',d)$

8. $\sigma(t,n) = alloc(x)$:
      foreach $d \in Data$: $LS(x,d) := \emptyset$

9. $\sigma(t,n) = commit(R,W)$:
      foreach $(o',d)$:
         if $LS(o',d) \cap (R \cup W) \neq \emptyset$
            add $t$ to $LS(o',d)$
         if $(o',d) \in R \cup W$:
            if $LS(o',d) \neq \emptyset$ and $\{t, TL\} \cap LS(o',d) = \emptyset$
               report data race on $(o',d)$
            $LS(o',d) := \{t, TL\}$
         if $t \in LS(o',d)$
            add $R \cup W$ to $LS(o',d)$
```

Figure 5. The lockset update rules for the Goldilocks algorithm

Other ways of specifying the interaction between strongly-atomic transactions and the Java memory model can easily be incorporated into our definition of extended races. For instance, one can choose to define the extended synchronizes-with order to include *all* atomic order edges, or to include a synchronizes-with edge from transaction $commit(R,W)$ to $commit(R',W')$ if $R' \cap W \neq \emptyset$. The algorithms and tools presented in this paper can easily be adapted to such alternative interpretations of strong-atomicity.

4. The generalized Goldilocks algorithm

In this section, we describe our algorithm for detecting data races in an execution $\Sigma = (\sigma, \xrightarrow{eso})$. The algorithm assumes that the execution is provided to it as some linearization of the extended happens-before relation $\xrightarrow{ehb}$. Formally, a linearization of an execution Σ is a function π that maps $\mathcal{N}$ one-one to $Tid \times \mathcal{N}$ such that if $(t,m) \xrightarrow{ehb} (u,n)$ then $\pi^{-1}(t,m) \leq \pi^{-1}(u,n)$. If an execution contains a data-race between a pair of accesses, our algorithm declares a race at one of these accesses regardless of which linearization is picked. For simplicity, the exposition in this section does not distinguish between read and write accesses. This distinction and its effect on the Goldilocks algorithm is explained in the next section.

The algorithm uses an auxiliary map LS from $(Addr \times Data)$ to $Powerset((Addr \times Volatile \cup Data) \cup Tid \cup \{TL\})$. For each data variable (o,d), the lockset $LS(o,d)$ may contain volatile variables, data variables, thread identifiers, or a special value TL (Transaction Lock). The value TL models a fictitious global lock that is acquired and released at the commit point of each transaction. Initially, the map LS assigns the empty set to every data variable (o,d). The algorithm updates LS as each element in the linearization of σ is processed. The set of rules for these updates is shown in Figure 5. The intuitive interpretation of a lockset $LS(o,d)$ is as follows:

- If $LS(o,d)$ is empty, it indicates that (o,d) is a fresh variable which has not been accessed so far. Therefore, an access to (o,d) when $LS(o,d)$ is empty is necessarily race-free. $LS(o,d)$ is initialized to the empty set and is reset to the empty set whenever o is an object returned by a memory allocation.
- If a thread identifier t is in $LS(o,d)$, then t is an owner of (o,d) and an access by t to (o,d) is race-free.
- If a lock (o',l) is in $LS(o,d)$, a thread can become an owner of (o,d) by acquiring the lock (o',l). This is because the thread that last accessed (o,d) released (o',l) subsequently.
- If a volatile variable (o',v) is in $LS(o,d)$, a thread can become an owner of (o,d) by reading (o',v). This is because the thread that last accessed (o,d) wrote to (o',v) subsequently.
- If TL is in $LS(o,d)$, then the last access to (o,d) was performed inside a transaction. Hence, there will be no race on (o,d) if the next access is also performed inside a transaction.
- If a data variable (o',d') is in $LS(o,d)$, a thread can become an owner of (o,d) by accessing (o',d') inside a transaction. This is because the thread that last accessed (o,d) also accessed (o',d') in a transaction subsequently.

Given this interpretation of $LS(o,d)$, we conclude that an access of (o,d) by a thread t outside any transaction is race-free if and only if at that point in the execution either $LS(o,d)$ is empty or $t \in LS(o,d)$. After this access, $LS(o,d)$ contains only t, representing the constraint that between this access and any future accesses to (o,d) by other threads, there must be synchronization actions to hand over ownership of (o,d). Similarly, an access of (o,d) by a thread t inside a transaction is race-free if and only if at that point in the execution either $LS(o,d)$ is empty or $t \in LS(o,d)$ or $TL \in LS(o,d)$. After this access, $LS(o,d)$ contains only t and TL, representing the constraint that the next access to (o,d) by a thread t' different from t must either be inside a transaction or there must be synchronization actions to hand over ownership of (o,d) to t'.

The rules in Figure 5 take as input a linearization π of $\Sigma = (\sigma, \xrightarrow{eso})$. The pair (t,n) used in the rules represents the value of $\pi(i)$ for an arbitrary $i \in \mathcal{N}$. Note that each of the rules 2–7 and 9 requires 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 5, we present an efficient scheme to implement our algorithm by applying these updates lazily.

The following theorem expresses the fact that our algorithm is both sound and precise. Given an execution $\Sigma = (\sigma, \xrightarrow{eso})$, we write that (t,n) accesses the data variable (o,d) in Σ if either

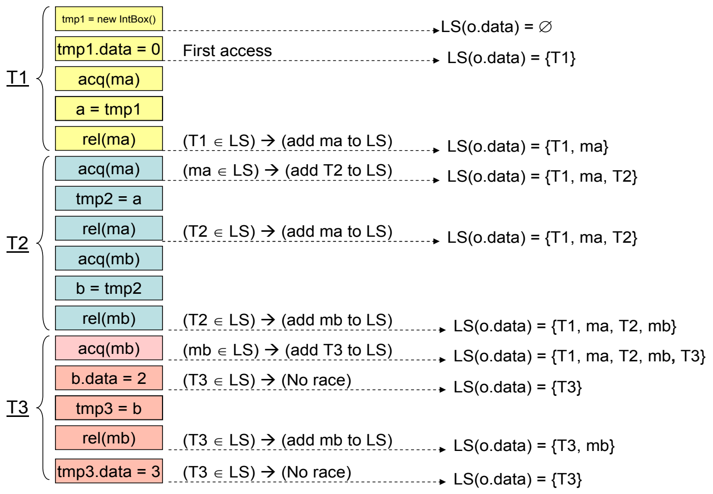

Figure 6. Evolution of $LS(\texttt{o.data})$ on Example 2 (Section 2).

$\sigma(t, n) \in \{read(o, d), write(o, d)\}$ or $\sigma(t, n) = commit(R, W)$ and $(o, d) \in R \cup W$.

THEOREM 1 (Correctness). *Let $\Sigma = (\sigma, \xrightarrow{eso})$ be an execution, π a linearization of Σ, and (o, d) a data variable. Let LS_b be the value of the lockset map LS as computed by the generalized Goldilocks algorithm just before processing the b-th element of π. Suppose $a, b \in \mathcal{N}$ are such that $a < b$, $\pi(a)$ and $\pi(b)$ access (o, d) in Σ, and $\pi(j)$ does not access (o, d) in Σ for all $j \in [a+1, b-1]$. Suppose $\pi_a = (t, m)$ and $\pi_b = (u, n)$. Then the following statements are true:*

1. *$u \in LS_b(o, d)$ iff $\pi(a) \xrightarrow{ehb} \pi(b)$.*
2. *$TL \in LS_b(o, d)$ iff $\sigma(t, m) = commit(R, W)$ and $(o, d) \in R \cup W$.*

The proof of this theorem is an extension of the proof of correctness of the original Goldilocks algorithm [9]. The extension to deal with transactions is mostly straightforward.

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 4.1). 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.

4.1 Precise data-race detection

In this section, we use an execution (Figure 6) of the program in Example 2 from Section 2 to demonstrate the precision of the Goldilocks algorithm compared to other lockset-based algorithms from the literature. Unlike Goldilocks which is both sound and precise, other lockset algorithms based on the Eraser algorithm [20] are sound but not precise.

The most straightforward lockset algorithm is based on the assumption that each shared variable is protected by a fixed set of locks throughout the execution. Let $LH(t)$ represent the set of locks held by thread t at a given point in an execution. This algorithm attempts to infer this set by updating the lockset $LS(o, d)$ of a data variable (o, d) to be the intersection $LH(t) \cap LS(o, d)$ at each access to (o, d) by a thread t. If this intersection becomes empty, a race is reported. This approach is too conservative since it reports a false race in many situations, such as during unprotected variable initialization and when the lock protecting a variable changes over time. For instance, this basic algorithm will report a data-race on `o.data` at the very first access of the execution in Figure 6.

Variants of lockset algorithms in the literature use additional mechanisms such as a state machine per shared variable in order to handle special cases such as thread locality and object initialization. However, these variants are neither sound nor precise, and they all report false alarms in scenarios similar to the one in the example above. For instance, in spite of using the state-machine accompanying the Eraser algorithm [20], a data-race will be reported at the last access (`tmp3.data = 3`) to `o.data` in the execution.

The fundamental limitation of existing lockset algorithms is that the lockset of a variable only becomes smaller with time. On the other hand, our algorithm's lockset update rules allow a variable's locksets to grow during the execution; in fact, the lockset of a variable may be modified even without the variable being accessed. These aspects of our algorithm are crucial for detecting changes of ownership during the execution. The evolution of the lockset of `o.data` due to our update rules is illustrated in Figure 6.

The vector-clock algorithm does not declare a false race in this example and similar scenarios. However, it accomplishes this at significantly increased computational cost compared to our optimized implementation of the lockset update rules.

4.2 Handling transactions

In this section, we use the program in Example 3 from Section 2 to demonstrate how Goldilocks handles synchronization due to transactions. Consider an execution of this program shown in Figure 7. This execution begins in a state in which `head` = `null` and consequently the linked list of `Foo` objects is empty. Let `o` denote the address of the object allocated by the first statement. The transaction of `Thread 1` happens-before the transaction of `Thread 2` because both transactions access the variables `head` (address denoted by `&head`) and `o.data`. The transaction of `Thread 2` happens-before the transaction of `Thread 3` for exactly the same reason. Consequently, the accesses to `o.data` at `t1.data = 42` by `Thread 1`, at `iter.data = 0` by `Thread 2`, and at `t3.data++` by `Thread 3` are ordered by the happens-before relation. Goldilocks is able to detect these happens-before edges and verify that there is no data-race between the three accesses. Figure 7 shows the evolution of the lockset for `o.data`. The figure treats the end of a transaction as its commit point and shows the application of rule 9 from Figure 5 there.

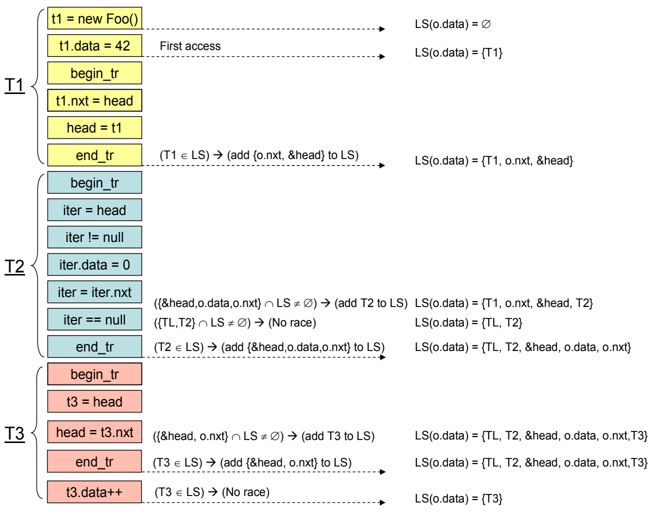
T1
t1 = new Foo()
LS(o.data) = ∅
t1.data = 42
First access
LS(o.data) = {T1}
begin_tr
t1.nxt = head
head = t1
end_tr
(T1 ∈ LS) → (add {o.nxt, &head} to LS)
LS(o.data) = {T1, o.nxt, &head}
T2
begin_tr
iter = head
iter != null
iter.data = 0
iter = iter.nxt
({&head,o.data,o.nxt} ∩ LS ≠ ∅) → (add T2 to LS)
LS(o.data) = {T1, o.nxt, &head, T2}
iter == null
({TL,T2} ∩ LS ≠ ∅) → (No race)
LS(o.data) = {TL, T2}
end_tr
(T2 ∈ LS) → (add {&head,o.data,o.nxt} to LS)
LS(o.data) = {TL, T2, &head, o.data, o.nxt}
T3
begin_tr
t3 = head
head = t3.nxt
({&head, o.nxt} ∩ LS ≠ ∅) → (add T3 to LS)
LS(o.data) = {TL, T2, &head, o.data, o.nxt,T3}
end_tr
(T3 ∈ LS) → (add {&head, o.nxt} to LS)
LS(o.data) = {TL, T2, &head, o.data, o.nxt,T3}
t3.data++
(T3 ∈ LS) → (No race)
LS(o.data) = {T3}

Figure 7. Evolution of $LS(\texttt{o.data})$ on Example 3 (Section 2).

5. Implementation

We implemented the generalized Goldilocks algorithm in Kaffe [24], a clean room implementation of the Java virtual machine in C. Our implementation is integrated into the interpreting mode of Kaffe's runtime engine[2]. The pseudocode for the Goldilocks implementation is given in Figure 8. We defer the explanation of how transactions are handled in the implementation to Section 5.3 to make the initial presentation simpler.

We store synchronization events in a linked list called the *synchronization event list* and represented by its *head* and *tail* pointers in Figure 8. Events are stored in this list in the synchronization order as defined in the Java Memory Model. Synchronization events are represented by the *Cell* data structure, which stores the synchronization action kind and the thread performing the action. When a thread performs a synchronization action, it atomically appends (see *Enqueue-Synch-Event*) the corresponding cell to the synchronization event list.

The race-freedom check presented in this paper is implemented in a decentralized fashion. For each action α that a thread performs, it calls *Handle-Action*(t, α). For each shared data variable (o, d), to serialize the lockset update and race-freedom checks for each access to (o, d), our implementation uses a unique lock $KL(o, d)$. Before a thread accesses (o, d), it acquires $KL(o, d)$ and performs the lockset update (computes the lockset associated with the access) and race-freedom check explained in Theorem 1. In this way, each thread carries out the race-freedom check for the accesses it performs, and these checks are linearly ordered by $KL(o, d)$. The implementation of the variable access in Kaffe is not protected by $KL(o, d)$.

The lockset update rules in our race-detection algorithm may require $LS(o, d)$ to be updated for each synchronization event. This potentially expensive operation and the memory cost of explicitly representing the individual locksets is avoided by performing lazy evaluation of locksets as described below.

The record *Info* corresponds to an access to a data variable (o,d) during the execution. The $owner$ field of *Info* is the id of the thread performed the access. pos is a pointer into the synchronization event list, to the $cell$ representing the last synchronization event

```
$record\ Cell$ {                    $record\ Info$ {
    $thread$: $Tid$;                    $pos$: $ref(Cell)$;
    $action$: $Action$;                 $owner$: $Tid$;
    $next$: $ref(Cell)$;}               $alock$: $Addr$;
                                        $ls$: $\wp(Addr \times Volatile \cup Data) \cup Tid \cup \{TL\}$
                                        $xact$: $Boolean;$ }

$head, tail$: $ref(Cell)$;
$ReadInfo$: $(Addr \times Data \times Tid) \longrightarrow Info$;
$WriteInfo$: $(Addr \times Data) \longrightarrow Info$;

Initially $head := new\ Cell$; $tail := head$;
$ReadInfo := EmptyMap$; $WriteInfo := EmptyMap$;

Enqueue-Synch-Event ($t$, $\alpha$):
1    $tail \rightarrow thread := t$;
2    $tail \rightarrow action := \alpha$;
3    $tail \rightarrow next := new Cell()$;
4    $tail := tail \rightarrow next$;

Check-Happens-Before ($info_1$, $info_2$):
1  if ($info_1.xact \wedge info_2.xact$)
2      $return$; // no race for transactional variables
3  if (($info_2.owner \neq info_1.owner$)
4      $\wedge$ ($info_1.alock$ is not held by $info_2.owner$)) {
5      Apply-Lockset-Rules ($info_1.ls$, $info_1.pos$, $info_2.pos$, $info_2.owner$) ;
6      $info_2.alock$ := (choose randomly a lock held by $info_1.owner$);
7  }

Handle-Action ($t$, $\alpha$):
1  if ($\alpha \in \{acq(o),\ rel(o),\ fork(u),\ join(u),\ read(o, v)$
           $write(o, v)\ finalize(x),\ terminate(t)\}$) {
2      Enqueue-Synch-Event ($t$, $\alpha$);

3  } else if ($\alpha\ = read(o, d)$) {
4      $info$ := $new Info$ ();
5      $info.owner$ := t;
6      $info.pos$ := $tail$;
7      $info.xact$ := Is-In-Transaction ($t$);
8      $info.ls$ := $\{t\}$;
9      $ReadInfo(o, d, t) := info$
10     Check-Happens-Before ($WriteInfo(o, d)$, $info$);

11 } else if ($\alpha\ = write(o, d)$) {
12     $info$ := $new Info$ ();
13     $info.owner$ := t;
14     $info.pos$ := $tail$;
15     $info.xact$ := Is-In-Transaction ($t$);
16     $info.ls$ := $\{t\}$;
17     for each ($t \in Tid$)
18         if ($ReadInfo(o, d, t) \neq$ null)
19             Check-Happens-Before ($ReadInfo(o, d, t)$, $info$);
20     if ($WriteInfo(o, d) \neq$ null)
21         Check-Happens-Before ($WriteInfo(o, d)$, $info$);
22     $WriteInfo(o, d) := info$
23     for each ($t \in Tid$) $ReadInfo(o, d, t)$ := null

24 } else if ($\alpha\ = commit(R, W)$) {
25     Enqueue-Synch-Event ($t$, $\alpha$);
26     for each $\tilde{\alpha}$ in ($W \cup R$)
27         Handle-Action ($t$, $\tilde{\alpha}$); // Check race freedom as regular access
28     }
29 }
```

Figure 8. Implementation of the Goldilocks algorithm

that the access comes after. The *ls*field contains the lockset of the variable just after the access. The fields $alock$ and $xact$ will be explained below while discussing the short-circuit checks and the transactions.

After an access α_1 to a variable, a new $Info$ instance, say $info_1$, that represents the current access is created. $info_1.pos$ is simply set to the current tail of the list (see lines 6,14 of *Handle-Action*). At this point, according to the variable access rule in Figure 5, the lockset of the variable should contain only the id of the currently

[2] An implementation of our algorithm in the just-in-time compilation mode is straightforward but requires a lot more code.

accessor thread. $info_1.ls$ is set after the access in order to reflect this (see lines 8 and 16 of *Handle-Action*).

We perform the lockset computation and race-freedom check lazily, only when an access to a variable happens. At a subsequent access α_2 to (o,d), represented by, say, $info_2$, the lockset $LS(o, d)$ of (o,d) after α_2 is computed and implicitly represented by applying to the lockset of the last access, $info_1.ls$, the lockset update rules for the events in the synchronization list between $info_1.pos$ and the current tail of the synchronization list. A race is reported if the lockset after α_2 does not contain the id of the current accessor thread.

The *Check-Happens-Before* procedure determines whether there is a happens before relationship between two accesses represented by two $info$ data structures. Before applying the lockset update rules of Figure 5, *Check-Happens-Before* first applies three cheaper checks that are sufficient conditions for race-freedom between the two accesses. These "short-circuit checks" are described in the next section. Our experimental results indicate that the short-circuit checks succeed most of the time, and the lockset update rules are only applied in the case of more elaborate ownership transfer scenarios. If all the short-circuit checks fail to prove the happens-before edge, the lockset of the variable is computed lazily in *Apply-Lockset-Rules* as described above.

To distinguish between read and write accesses, the implementation maintains the *Info* records for the last write access to each shared data variable (o, d) ($WriteInfo(o, d)$) and the last read access to (o, d) by thread t ($ReadInfo(o, d, t)$) if this access came after the last write access. These maps are updated after each access to (o,d). Note that, for each data variable, there are potentially as many pointers into the synchronization event list from $ReadInfo$ as the number of active threads, although, in practice, we rarely encounter this situation. In this case, instead of checking the happens-before edge between any two accesses to (o,d), our algorithm checks 1) for each read access whether it happened before $WriteInfo(o, d)$, and 2) for each write access whether it happened before $ReadInfo(o, d, t)$ for each thread t.

5.1 Short circuit checks

The first constant-time short-circuit check is effective when two consecutive accesses to a shared variable are performed by the same thread. In this case, the happens-before relationship is guaranteed by program order. This is detected in constant time by checking whether the $thread$ field of the $info$ data structure is the same as the thread performing the current access (see line 3 of *Check-Happens-Before*).

The second constant-time short-circuit check requires checking whether a random element of $LS(o, d)$ at the last access, kept in the $alock$ field of *Info*, is also held by the current thread. If these two locks happen to be the same, then the current access is race free (see line 4 of *Check-Happens-Before*).

The last short-circuit involves the *Apply-Lockset-Rules* subroutine and consists of considering only the subset of synchronization events executed by the current and last accessing thread when examining the portion of the synchronization event list between $info_1$ and $info_2$. This check is not constant time, but we found that it saves on the runtime of *Apply-Lockset-Rules* when the happens-before edge between $info_1$ and $info_2$ is immediate, i.e., is accomplished by a direct transfer of ownership from one thread to another.

5.2 Sound static race analysis to reduce overhead

As is apparent from the implementation pseudocode, the runtime overhead of race detection is directly related to the number of data variable accesses checked and the synchronization events that occur. In the worst-case, this overhead is proportional to the product of the following two quantities: (i) the sum of the number of synchronization events and shared variable accesses, (ii) the number of data variables in the execution. This overhead is amortized over the total number of accesses. In practice, we do not encounter this worst-case behavior and see a constant-time overhead per access. To reduce the number of accesses checked at runtime, we use existing static analysis methods at compile time to determine accesses or access pairs that are guaranteed to be race-free.

We worked with two static analysis tools for this purpose: a newer version of the Chord tool [17] and the RccJava static race detection tool [1]. The output of RccJava is a list of fields that may be involved in a race condition. The output of Chord is a list $\mathcal{R}$ of pairs of accesses (line numbers in the source code) that *may* be involved in a race, i.e., if any execution of the program ever produces a race, the pair of accesses (α_1, α_2) is guaranteed to be in $\mathcal{R}$. Our runtime makes use of $\mathcal{R}$ by parsing Chord's output and inferring from it the sets of object fields ($\mathcal{F}$) and methods ($\mathcal{M}$) that are guaranteed to never be involved in a race. It then annotates the Java class files using the reserved bits of the access flags of classes, fields and methods to enable/disable race checking on the particular class, field or method.

5.3 Transactions

To detect races at runtime, our approach requires a transaction manager to provide or make possible for the runtime to collect for each transaction $commit(R, W)$ the sets R and W and the place of commit point of the transaction in the global synchronization order. Note that transaction implementations need to ensure such an order to implement the required semantics correctly, thus to provide the latter information, transaction implementations do not need to perform any additional synchronization. For the transaction implementations in LibSTM [12] and LibCMT [13], this information is readily available to the runtime, and can be collected easily in SXM [14] at runtime.

In our implementation (Figure 8), when a transaction commit action is encountered, we first insert the commit action α (that contains the list of read and write accesses R and W) as a synchronization action into the synchronization event list. Then, all shared variable reads and writes within the transaction (i.e., in R and W) are checked for race-freedom (lines 24-28 of *Handle-Action*) in the same way as accesses outside transactions. Note that the collection of R and W is done in a distributed fashion by each transaction and does not introduce extra synchronization between transactions. We use the $xact$ field of *Info* in order to indicate whether an access happened inside a transaction or not (see lines 7,15 of *Handle-Action*). $xact$ is used in another short-circuit check (line 1 of *Check-Happens-Before*); as long as the accesses to (o,d) are inside transactions, the lockset computation is skipped.

This way of handling the happens-before relationships induced by transactions avoids processing the synchronization operations performed by a transactions manager that has already been proven correct. It infers exactly the desired set of happens-before edges, and accomplishes this at less of a computational cost than treating transactions implementations as part of the application program. This approach can be viewed as a modular way of checking race-freedom and thus sequential consistency. The implementor of software transactions guarantees the happens-before edges between transactions as described in Section 3 and race-freedom for accesses within transactions. Our Java runtime performs the race-freedom check for the rest of the accesses using the happens-before edges provided by the transactions implementation.

5.4 Garbage collection of synchronization events and partially- eager evaluation

The synchronization events list is periodically garbage collected when there are entries in the beginning of the list that are not

			Uninstrumented Runtime		Without static information		With Chord outputs		With RccJava outputs		Short-circuit checks (%)	
Benchmarks	# Lines	# Threads	Just-in-time	Interpreted	Runtime	Slowdown	Runtime	Slowdown	Runtime	Slowdown	Chord	RccJava
colt	-	10	6.3	6.8	13.2	1.9	7.8	1.2	-	-	66.67	-
hedc	2.5K	10	3.1	7.5	14.5	1.9	11.0	1.5	-	-	31.94	-
lufact	1K	10	0.2	0.5	2.0	4.1	0.9	1.8	-	-	12.73	-
moldyn	650	5	21.6	135.2	730.2	5.4	712.9	5.3	217.8	1.6	99.53	99.99
montecarlo	3K	5	43.8	44.1	97.2	2.2	47.0	1.1	44.8	1.0	99.93	99.98
philo	86	8	2.9	2.6	2.7	1.0	2.7	1.0	-	-	6.62	-
raytracer	1.2K	5	0.4	3.4	61.7	17.9	39.0	11.4	7.4	2.1	51.01	51.01
series	380	10	87.9	88.4	94.1	1.0	93.3	1.0	-	-	10.17	-
sor	220	5	31.5	32.8	44.1	1.3	31.5	1.0	-	-	16.97	-
sor2	252	10	0.6	11.2	70.8	6.3	25.8	2.3	12.4	1.1	0.00	99.00
tsp	700	10	1.6	2.5	5.5	2.2	3.4	1.3	3.1	1.2	64.54	37.28

Table 1. Results of experiments with the race-aware Kaffe runtime

relevant for the lockset computation of any variable. This is the case when an entry in the list is not reachable from the lockset pointer (pos) for any $info$ data structure. We keep track of this information by maintaining a reference count in every $cell$ data structure. We periodically discard prefix of the list up to the first cell with non-zero reference count.

While running Goldilocks on long executions, sometimes the synchronization event list gets too long. It is not possible to garbage-collect starting from the head of the list when a variable is accessed early in an execution but is not used afterwards. In this case, a long prefix of the list is reachable from the $info$ data structure representing this early access, which prevents garbage collection of the entire rest of the list. This phenomenon is a side effect of the initial Goldilocks implementation performing "completely lazy" evaluation. Below, we describe a technique, "partially-eager" lockset evaluation, that we used to address this problem. For simplicity, partially-eager evaluation is explained only for write locksets. The procedure is the same for each read lockset of each thread for each shared data variable.

Suppose that the initial $cell_0$ has a non-zero reference count, followed by a long sequence of entries with zero reference counts. Let $\mathit{InfoSet} \subseteq \mathit{WriteInfo}$ be the set of Info records whose pos fields refer to $cell_0$. Let $cell_1$ be a later cell in the synchronization event list. For each variable (o, d) such that $\mathit{WriteInfo}(o, d) \in \mathit{InfoSet}$, no later write access to (o, d) has occurred (otherwise the $\mathit{WriteInfo}$ data structure would point to a later list entry). Thus, for each such (o, d), we 1) perform the lockset computation up to $cell_1$, and store this intermediate result in $\mathit{WriteInfo}(o, d).ls$, and 2) reset $\mathit{WriteInfo}(o, d).pos$ to point to $cell_1$. If there is a later write access to (o, d), the lockset computation starts with this intermediate lockset (instead of $\{\mathit{WriteInfo}(o, d).owner\}$) and at $cell_1$. We perform this partial computation of locksets for all $info$ data structures that refer to $cell_0$. After the reference count of $cell_0$ reaches 0, we garbage-collect the prefix of the list up to $cell_1$. We repeat this prefix-trimming operation starting with the new head of the list until the list size is down to a pre-specified number. This partially-eager lockset evaluation scheme provides a running time-memory trade off and is necessary for continuously-running programs which may have very long synchronization event lists otherwise.

Currently we trigger the partially-eager lockset computation scheme to trim the first 10% of the entries in the synchronization event list when garbage collection is triggered in the Kaffe JVM. We explicitly trigger a garbage collection when the event list grows longer than one million entries.

6. Experiments

We ran on our race- and transaction-aware JVM a set of benchmarks well known in the literature. A Linux machine with Intel Pentium-4 2.80GHz CPU was used. The initial and the maximum Java heap space were set to 16 MB and 512 MB, respectively. These limits triggered a moderate number of garbage collection runs which were necessary for proper trimming of the synchronization event list by partially-eager lockset propagation. The experiments were run on the interpreter mode of the Kaffe Virtual Machine version 1.1.6. We collected the runtime statistics using the PAPI interface to the hardware performance counters [2]. Arrays were checked by treating each array element as a separate variable.

To provide a reasonable idea of race checking overhead in our experiments, when a race was detected on a variable, race checking for that variable was turned off during the rest of the execution. Checks for all the indices of an array were disabled when a race is detected on any index of the array. The Java benchmarks we worked on do not have handlers for `DataRaceExceptions`. It is common in these benchmarks for many more races to occur on a variable once one race has occurred. Turning off race checking after the first detected race provides a more reasonable idea of overhead in these cases because, in normal operation, we do not expect a program to continuously detect and recover from races on the same variable.

We used eleven benchmark programs to evaluate the performance of our implementation in Kaffe. Six of our benchmarks are from the Java Grande Forum Benchmark Suite. We reduced the size of inputs to the Grande benchmarks because of the long running time of the applications. The remaining four benchmarks are from [23].

We were able to apply the Chord static race analysis tool to all of the benchmarks and used the resulting race-prone access pairs as described in Section 5. We only had access to RccJava output for the following benchmarks: `moldyn`, `montecarlo`, `raytracer`, `sor2`, and `tsp`. In most of these cases, the checks that RccJava was able to eliminate subsumed those that Chord was able to.

Table 1 presents the performance statistics of the Goldilocks algorithm in different settings of the benchmark programs. The column titled "Uninstrumented" reports the total runtime of the programs in the JIT-compilation and the interpreter modes of JVM with the race detection feature disabled. The other columns present the running times and the slowdown ratios of the programs on JVM with race checking enabled. The runtime measurements are given in seconds. The slowdown ratio is the ratio of the runtime with the race checking enabled to the uninstrumented runtime of the program (both in interpreted mode). For the results in the columns titled "with Chord outputs", "with RccJava outputs", and "short-circuit checks", the standard libraries were instrumented as well. For the results in the columns titled "without static information" the libraries were not instrumented. For these experiments, instrumenting libraries at most doubles overhead.

Table 1 also reports the percentages of succeeding short-circuit checks when the outputs of Chord or RccJava are used. The rest of the accesses require full lockset computations by traversal of the

	Variables checked (%)		Accessed checked (%)	
Benchmarks	Chord	RccJava	Chord	RccJava
colt	0.1	-	0.0	-
hedc	0.0	-	0.0	-
lufact	1.1	-	2.1	-
moldyn	84.1	83.5	56.6	49.1
montecarlo	15.4	13.7	13.0	39.9
philo	0.2	-	0.1	-
raytracer	80.5	35.3	5.2	1.3
series	0.3	-	2.0	-
sor	0.3	-	0.9	-
sor2	1.3	0.0	36.9	0.0
tsp	60.1	80.0	42.0	23.7

Table 2. Statistics on experiments with static analyses

#Threads	Uninstrumented	Goldilocks with transactions		#Accesses	#Transactions
	Runtime	Runtime	Slowdown	x1000	x1000
5	0.35	0.51	1.46	215	21
10	0.66	0.80	1.21	381	45
20	1.11	1.60	1.44	660	87
50	2.89	3.55	1.23	1460	206
100	5.57	8.19	1.47	2819	407
200	12.15	15.30	1.26	5493	802
500	31.78	40.84	1.29	13648	2006

Table 3. Performance of checking races for transactional Multiset

synchronization event list. These results clearly indicate that for some programs, most of the happens-before edges can be captured by using cheaper short-circuit checks, which significantly reduces overhead. The high percentage of lockset computations for `sor2` explains the high overhead incurred for this benchmark. In `lufact`, `philo`, `series` and `sor` the percentage of successful short-circuit checks is high, which led to low runtime overhead. The short-circuit checks do not help to reduce the overhead with Chord outputs for `moldyn` and `raytracer`. This is because `moldyn` and `raytracer` use barrier synchronization which is not captured by Chord, and thus, each volatile variable read and write used to implement barriers is processed separately in the lockset computations. The overhead for these examples is much lower when RccJava outputs are used, because RccJava eliminates checks for most of the variables whose accesses are synchronized using barriers. Note that Eraser-based algorithms do not handle barrier synchronization and would have declared false alarms for `moldyn` and `raytracer`.

Table 2 reports the percentage values for the variables checked among all the variables created, and the variable accesses checked among all the accesses that take place during the execution. Using the output of Chord reduces the slowdown to a small value between 1 and 2 for most of the benchmarks. Using RccJava outputs decreases the slowdown for `moldyn` and `raytracer` to similar levels, whereas overhead remained high with the outputs of Chord for these examples. These experiments demonstrate that, with proper prior static analysis, the overhead of precise race checking at runtime can be reduced significantly. These results also indicate that, to be practical, precise race detection at runtime must be preceded by sound static techniques for determining race-free variables.

6.1 Experiments with transactions

To measure the performance of our implementation of a transaction-aware race checker, we mimicked the transactions implementation by source-to-source translation of Grossman et. al. [15] for a concurrently-accessed data structure. We did this as we did not have access to the source code for a Java implementation of software transactions.

The implementation of transactions in [15] uses Java's object locks for synchronization. All shared variable reads and writes in an atomic transaction are protected by the object locks for the objects accessed. All shared variable reads and writes that are part of a transaction take place between the first lock acquire and the first lock release associated with the transaction. The first lock release also constitutes the commit point of the transaction.

The data structure we worked on is a `Multiset`, based on the benchmark with the same name in [10]. The test program for `Multiset` consists of a number of threads accessing a multiset of integers concurrently by performing insertions, deletions and queries. The representation for a multiset is an array `elements` of objects where each object potentially stores an element of the multiset. The `Insert(int[] a)` operation attempts to first allocate space for the `a.length` entries in `elements` using a transaction for each allocation. If this allocation is successful for all `a.length` entries, then all of the new multiset elements are made visible to other threads in an atomic transaction. If the allocation is unsuccessful because of space contention with concurrent threads, the space allocated in `elements` is freed in a single atomic transaction. This mimics transaction rollback. To mimic the use of transactions mixed with other synchronization primitives, the arrays used as arguments to `Insert` were generated by a factory object shared among threads. This object and the array objects were manipulated outside transactions.

To imitate what our JVM expects of a transactions implementation, shared variable reads and writes between the first lock acquire and the first lock release in each transaction were recorded by instrumentation code in the JVM to form the sets R and W associated with a $commit(R, W)$ action. This commit action was inserted in the synchronization event list where the first lock release in a transaction would have been inserted if we were not explicitly considering transactions.

We measured, for different numbers of threads sharing a multiset of size 10, the runtime with race checking enabled for the transactions as described in Sections 4 and 5 (Table 3). The table also reports the number of shared variable accesses and the number of transactions in the executions. The results indicate that the runtime overhead of our approach (which includes the overhead of keeping track of the read and write sets of transactions) when explicitly handling transactions is moderate. When we analyze `Multiset` executions without taking transactions into account we incur slowdown factors of more than ten[3]. This indicates that treating software transactions as high-level synchronization primitives may reduce the runtime overhead of race checking.

7. Related work

There are two approaches to dynamic data-race detection, one based on locksets and the other based on the happens-before relation. Eraser [20] 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.

The other approach to dynamic data-race detection is based on computing the happens-before relation [7, 19, 21] using vector clocks [16]. Hybrid techniques [18, 25] combine lockset and happens-before analysis. For example, RaceTrack [25] uses a basic

[3] This overhead is not representative as Multiset executions consist entirely of shared variable accesses each involving a separate lock acquire and release.

vector-clock algorithm to capture thread-local accesses to objects thereby eliminating unnecessary and imprecise applications of the Eraser algorithm. Our technique, for the first time, computes a precise happens-before relation using an implementation that makes use of only locksets.

There is also prior work that used the result of static analysis to eliminate unnecessary runtime checks. Choi et al. [6] present an unsound runtime algorithm for data-race detection. They used a static race reporting algorithm [5] to eliminate unnecessary checks for well-protected variables.

The work on software transactional memory (STM) is orthogonal to our work; our algorithm can be integrated with all the implementations we are aware of. Our definition of data-races in the presence of transactions was influenced in part by a study of the interaction between the synchronization induced by software transactions and weaker memory models [11].

8. Conclusion

We present the first formulation of data-races in the presence of software transactions and a race- and transaction-aware runtime for Java. We have designed and implemented a precise and efficient algorithm for detecting data races in dynamic executions. Our algorithm uniformly supports a variety of synchronization mechanisms including software transactions. Through a combination of static analysis and an efficient implementation of our data-race detection algorithm, we have demonstrated that the runtime overhead of precise data-race detection required for supporting a `DataRaceException` can be made reasonable. With improvement of static analysis techniques and further optimizations in the implementation, we believe that the runtime overhead can be reduced enough to be acceptable for continuous monitoring of program executions during debugging and during deployment for critical programs.

References

[1] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, 2006.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The Intl. Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[3] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting Data Races in Cilk Programs That Use Locks. In *SPAA 98: Annual ACM Symposium on Parallel Algorithms and Architectures*. pages 298309, Puerto Vallarta, Mexico, June 28-July 2, 1998.

[4] James R. Larus and Ravi Rajwar. Transactional Memory. *Synthesis Lectures on Computer Architecture*. 1(1):1–226. 2006.

[5] Jong-Deok Choi, Alexey Loginov, and Vivek Sarkar. Static Datarace Analysis for Multithreaded Object-oriented Programs. Technical Report, RC22146. IBM Research. 2001.

[6] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In *PLDI '02: Proc. ACM SIGPLAN 2002 Conf. on Programming Language Design and Implementation*, pp. 258–269, New York, NY, USA, ACM 2002.

[7] Mark Christiaens and Koenraad De Bosschere. TRaDe: Data race detection for Java. In *Proc. of the Intl. Conference on Computational Science - ICCS2001*, number 2074, pp. 761–770, San Francisco, May 2001. Springer Verlag.

[8] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently Computing the Happens-before Relation Using Locksets. In *Proc. of the Workshop on Formal Approaches to Testing and Runtime Verification (FATES/RV 2006).* 2006.

[9] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently Computing the Happens-before Relation Using Locksets. Technical Report MSR-TR-2006-163, Microsoft Research, 2006.

[10] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer. Vyrd: Verifying Concurrent Programs by Runtime Refinement-Violation Detection. In *PLDI '05: Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 27–37, New York, NY, USA, ACM 2005.

[11] Dan Grossman, Jeremy Manson, and William Pugh. What Do High-level Memory Models Mean for Transactions? In *MSPC'06: Proc. of the 2006 Workshop on Memory System Performance and Correctness*, pp. 62–69, New York, NY, USA, 2006. ACM Press.

[12] Tim Harris and Keir Fraser. Language Support for Lightweight Transactions. In *OOPSLA '03: Proc. of the 18th annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, pp. 388–402, New York, NY, USA, ACM 2003.

[13] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *PPoPP '05: Proc. of the 10th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 48–60, New York, NY, USA, ACM 2005.

[14] Maurice Herlihy. SXM1.1: Software Transactional Memory Package for C#. Technical Report, Brown University & Microsoft Research, May, 2005.

[15] Benjamin Hindman and Dan Grossman. Atomicity via Source-to-source Translation. In *MSPC'06: Proc. of the 2006 Workshop on Memory System Performance and Correctness*, pp. 82–91, New York, NY, USA, 2006. ACM Press.

[16] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Intl. Workshop on Parallel and Distributed Algorithms*. 1988.

[17] Mayur Naik, Alex Aiken, and John Whaley. Effective Static Race Detection for Java. In *PLDI'06: Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 308–319, New York, NY, USA, ACM 2006.

[18] Eli Pozniansky and Assaf Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *PPoPP'03: Proc. of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 179–190, New York, NY, USA, ACM 2003.

[19] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[20] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[21] Edmond Schonberg. On-the-fly Detection of Access Anomalies. In *Proc. of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, volume 24, pp. 285–297, Portland, OR, June 1989.

[22] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pp. 204–213, ACM 1995.

[23] Christoph von Praun and Thomas R. Gross. Object race detection. In *OOPSLA'01: Proc. of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 70–82, New York, NY, USA, ACM 2001.

[24] Tim Wilkinson. Kaffe: A JIT and Interpreting Virtual Machine to Run Java Code. *http://www.transvirtual.com/*, 1998.

[25] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP'05: Proc. of the 20th ACM symposium on Operating systems principles*, pp. 221–234, New York, NY, USA, ACM 2005.