Helgrind$^+$: An Efficient Dynamic Race Detector

Ali Jannesari, Kaibin Bao, Victor Pankratius and Walter F. Tichy
University of Karlsruhe
76131 Karlsruhe, Germany
Email: {jannesari, bao, pankratius, tichy}@ipd.uni-karlsruhe.de

Abstract

Finding synchronization defects is difficult due to non-deterministic orderings of parallel threads. Current tools for detecting synchronization defects tend to miss many data races or produce an overwhelming number of false alarms. In this paper, we describe Helgrind$^+$, a dynamic race detection tool that incorporates correct handling of condition variables and a combination of the lockset algorithm and happens-before relation. We compare our techniques with Intel Thread Checker and the original Helgrind tool on two substantial benchmark suites. Helgrind$^+$ reduces the number of both false negatives (missed races) and false positives. The additional accuracy incurs almost no performance overhead.

Index Terms

Race detection, race conditions, debugging, parallel programs, multi-threaded programming, dynamic analysis, happens-before, lockset.

1. Introduction

As parallel computing enters the mainstream, synchronization defects such as deadlocks and data races become more prevalent. This paper concentrates on techniques for automated data race detection. A data race occurs when at least two threads access the same memory location with no ordering constraints enforced between the accesses, and at least one of the accesses is a write [1]. Race detectors can easily miss races or produce false alarms.

We developed a new method for correct handling inter-thread event notifications automatically and without relying on source code annotation. This method accurately establishes happens-before relations implied by condition variables and thus eliminates almost all cases of false alarms and missed races caused by wrong or missed detection of inter-thread event notifications.

We also propose a new memory state model, which takes full advantage of the high accuracy in detecting happens-before relations. The detection approach is based on combining the lockset algorithm with happens-before analysis. The new memory state model is optimized for short-running programs.

Our techniques have been implemented in Helgrind$^+$, an extension of the race detector Helgrind [2]. The user can choose which memory state model he or she wants to use and switch between happens-before analysis accuracy. Thus, the user can select the sensitivity depending on his or her situation. Helgrind$^+$ has been evaluated on the data race test suite for Helgrind [3] and on the PARSEC [4] benchmark. Results show that it is significantly more accurate than the original Helgrind as well as Intel's Thread Checker, with negligible increase in overhead.

The paper is organized as follows. Section 2 discusses motivation for this work. In Section 3, we define some of the terms and basics used in this paper. A a short overview of the lockset and happens-before detection is presented. We explain our method and the new features in Section 4 and 5. In Section 6, Valgrind [5], [6], [7], the kernel of Helgrind is introduced and then the implementation of Helgrind$^+$ is discussed. In Section 7, we evaluate the accuracy and the performance of Helgrind$^+$ compared to other detectors. Related works are discussed in Section 8 and in the last section we conclude our paper.

2. Motivation

2.1. More Sensitive Race Detection

In [8], we proposed a race detector that significantly reduces the false alarms. This approach is suitable for analyzing long-running applications without overwhelming the user with false alarms. In long-running applications, a data race pattern is likely to be repeated. Based on this

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

assumption, the race detector in [8] defers certain race reports until the race reoccurs, thus reducing false alarms. Conversely, it does not detect those deferred races if they occur only once.

But what happens if the program runs briefly? Then, races may not occur several times. This situation could happen especially during unit testing. For this reason, we introduce the option to discover races even if they occur only once.

2.2. Lost Signals

A higher sensitivity usually also means a higher rate of false alarms. To avoid this, the detector has to distinguish more accurately between real data races and harmless accesses. Only parallel accesses can lead to data races and the detector has to find out how accesses are ordered.

```
DATA++

lock(l)
  FLAG = 1
  signal(CV)
unlock(l)
```

(a) Thread 1

```
lock(l)
  while(FLAG != 1)
    wait(CV)
unlock(l)

DATA--
```

(b) Thread 2

Figure 1. Using synchronization primitives `signal()` and `wait()`.

In some cases, it is extremely difficult to reconstruct the implicit ordering imposed by synchronization primitives, see for example Figure 1. Thread 1 operates on `DATA` and then signals Thread 2 that it can take over the data for further processing. The threads are properly synchronized, but there is an ordering in which the happens-before relation caused by `signal()` and `wait()` is not visible to the race detector that would issue a false warning on `DATA`. This situation is as follows.

If Thread 1 finished first, Thread 2 would not call `wait()`. Consequently, the signal sent by Thread 1 is lost. Any instrumentation of `signal()` and `wait()` thus does not detect the proper ordering of the two threads. Thread 2 carries on and as soon as it accesses DATA, a data race is reported, even though there is none. The proper ordering is enforced by the condition variable `FLAG`, but noticed by neither lockset nor happens-before detectors.

In Section 5 we show how to solve this problem, without any source code annotation.

3. Definitions

Before we describe the components of our race detector in detail, let us first define some of the terms used later on.

3.1. Thread Segments

The instruction sequence of a thread can be sliced into subsequent pieces, called *thread segments*. Synchronization with other threads (or thread segments) happens at the start or at the end of each thread segment. Of course, all thread segments belong to a specific thread. Within a thread segment, all operations are totally ordered. The thread segments of each thread are also totally ordered. Synchronization defines a partial order of thread segments. If two thread segments are *not* ordered, they may execute in parallel.

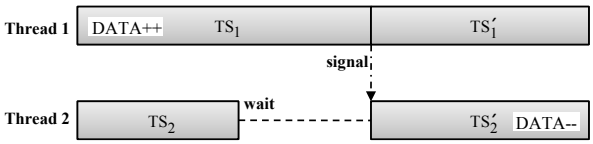


Figure 2. A thread consists of thread segments separated by synchronization operations.

Figure 2 shows the thread segment graph of a potential execution of the program depicted in Figure 1. Thread 1 sends a signal to Thread 2. Thus the first part of Thread 1 TS_1 happens before the second part of Thread 2, TS_2'. Both Thread 1 and 2 are accessing variable `DATA`. Because of the ordering, there is no race here.

3.2. Happens-Before Relation

For further discussion, it is useful to define a concise notation for the ordering of thread segments. Lamport's happens-before relation $\overset{hb}{\rightarrow}$ express exactly this [9]. When a thread segment TS_1 is executed before another thread segment TS_2, we say $TS_1 \overset{hb}{\rightarrow} TS_2$.

We define the relation $\overset{hb}{\rightarrow}$ to be reflexive and transitive. The relation is defined to be reflexive regarding thread segments, i.e. $TS_1 \overset{hb}{\rightarrow} TS_1$ is possible. This is because execution within a thread segment is strictly ordered and throughout our algorithm, we always compare the present point of execution with a past point of execution which could be in the same segment. Transitivity allows us to traverse through the thread segment graph and check if two segments are parallel: Two thread segments TS_1 and TS_2 are parallel iff there is no path of $\overset{hb}{\rightarrow}$-relations between them. This situation is denoted as $TS_1 \mid\mid TS_2$.

3.3. Lockset algorithm

The Lockset algorithm [10] is used to determine whether a variable is protected by a specific lock or a

set of locks. For each shared variable d a set of candidate locks C_d is maintained, which is refined every time the variable is accessed. The lockset algorithm reports a race whenever a candidate lockset C_d becomes empty.

This algorithm is insensitive to the program schedule, but has a high rate of false alarms, because it does not consider other synchronization primitives such as *Fork/Join*. Combining the happens-before relation with lockset analysis results in a hybrid solution with a trade-off between accuracy and runtime overhead. Recent race detectors [11], [12], [13], [14], [2] use the happens-before relation to take into consideration other synchronization primitives. These approaches still report many false positives and even miss races [8]. Additionally, many of them support only a subset of synchronization primitives.

4. The Race Detector

Our detection algorithm combines the happens-before relation and the lockset analysis in a new and efficient way. Basically, both the lockset algorithm and the happens-before analysis are performed. Although the lockset algorithm is a reliable method to detect correct synchronization using critical regions, the happens-before edges between locking and unlocking of a region are ignored, similar to previous works [11], [12], [13], [14], [15].

Our algorithm is based on the following preconditions:

1) The program uses only the following types of synchronization methods:
   - locks
   - fork/join
   - condition variables
   - barriers
2) Throughout the program, each shared variable can be protected by different types of the aforementioned methods. For example, a variable `x` at the beginning of the program could be protected by a lock, whereas later on, a barrier could be employed.

Helgrind$^+$ uses dynamic instrumentation to track the program execution. Section 4.1 describes in detail which operations are instrumented and how the current state of program and its threads are maintained.

To detect races, each variable has an associated state. This state indicates whether the variable is shared or exclusively accessed by which thread segments. Every access to a variable is instrumented to track the associated state according to a finite state machine. This state machine is a fundamental part of the race detector. Details appear in Section 4.2.

4.1. Instrumentation

4.1.1. Locks. For the Lockset algorithm, we need to know which locks are being held by each thread at any time. The locks held by Thread t are stored in the lockset L_t. When Thread t acquires or releases a lock, we have to update L_t in the following way:

After t executes Lock(l) :
$L_t \leftarrow L_t \cup \{l\}$

After t executes Unlock(l) :
$L_t \leftarrow L_t \backslash \{l\}$

4.1.2. Happens-before. For the happens-before analysis, Helgrind$^+$ maintains thread segments and the happens-before relations between them. For convenience, we define the function $NewSegment(TS_1, TS_2, ..., TS_n)$ that performs the following actions. It returns a new thread segment TS_{new} and adds new happens-before relations such that $\forall i : TS_i \overset{hb}{\rightarrow} TS_{new}$. At any point in time, each thread t is in one of thread segments. The current thread segment of thread t is called TS_t. When a thread executes a synchronization primitive described in this section, the current thread segment ends and a new one is created.

$Fork()$ / $Join()$ are used for creation and termination of threads. When a thread t creates a new thread u, everything u does happens after t's past operations before t created u. Thread u cannot hold any locks at that moment, so L_u is set to empty.

When thread t calls $Join(u)$, it will wait for thread u to terminate. That means everything thread u has done happens before any operation t will do after the joining. Additionally, on a $Join()$ operation, we will scan through all shared variables to see if some of them are not shared anymore. Each shared variable d is accessed by a set of threads called S_d. If S_d becomes singleton after the terminated thread u was excluded from the set, variable d can be reseted to non-shared or "exclusive" state.

Before t executes Fork(u) :
$L_u \leftarrow \emptyset$
$TS_u \leftarrow NewSegment(TS_u, TS_t)$
$TS_t \leftarrow NewSegment(TS_t)$

After t executes Join(u) :
foreach shared variable d:
$S_d \leftarrow S_d \backslash \{u\}$
if S_d is singleton:
reset d to exclusive state
$TS_t \leftarrow NewSegment(TS_t)$

$Signal()$ / $Wait()$ are the primitives for *inter-thread event notifications*. A thread t sends a signal while another thread u blocks until a signal was received. Operations of thread t before sending the signals happens before operations of thread u after receiving it. The thread segment of the signaler has to be stored so that the waiting thread can create a happens-before relation to it. As different signals can be sent depending on which condition variable cv is used, each condition variable can hold a thread segment TS_{cv}.

Before t executes Signal(cv) :
$TS_{cv} \leftarrow TS_t$
$TS_t \leftarrow NewSegment(TS_t)$

After u executes Wait(cv) :
$TS_u \leftarrow NewSegment(TS_u, TS_{cv})$

When using the $Barrier()$ primitive, each thread is allowed to leave the barrier only after all participating threads have reached it. Thus, each thread segment after the barrier happens after all other thread segments before the barrier. A Barrier stores an immediate thread segment TS_b. After all participating threads have reached the barrier, TS_b happens after all thread segments. By leaving the barrier, each thread segment synchronizes with TS_b.

Before t executes Barrier(b) :
$TS_b \leftarrow NewSegment(TS_b, TS_t)$

After t executes Barrier(b) :
$TS_t \leftarrow NewSegment(TS_t, TS_b)$

4.2. New Memory State Machine

The effect of a memory state machine on the outcome of a detector is crucial. With Helgrind$^+$ one can choose between two different memory state machines. Based on our empirical studies, the memory state machines are tailored and carefully tuned for two different categories of applications: long-running and short-running applications. Compared to the memory state machine of Eraser [10] and similar tools, our memory state machines are more complex and accurate. We address the limitations observed in earlier memory states by making the required lockset and threadset refinements carefully.

We provide both versions in Helgrind$^+$ to have a complete solution for different kind of applications. The user is able to choose the memory state machine depending on the application type.

4.2.1. Memory State Machine for Long-running Applications. We proposed in [8] a memory state machine mainly for long-running applications. It is based on the assumption that a past data race access pattern is likely to be repeated in the future. We refer to this state machine as MSM-long. MSM-long has eight different states and it defers the happens-before analysis until the lockset analysis proves enough insufficiencies. Our empirical result with MSM-long showed a significant reduction of false positives[8], making the tools practical for long-running applications.

4.2.2. Memory State Machine for Short-running Applications. We propose a new state machine which is more suitable for short-running applications. The new state machine concentrates on accurately detecting data races and prevents false negatives while avoiding false positives as demonstrated in Section 7. The new state machine is called MSM-short and compared to MSM-long, has two states less.

4.2.3. States of MSM-short. Figure 3 depicts MSM-short followed by description of each state and the required steps used in the detection algorithm. Thread segments (e.g. TS_t) are used to indicate a happens-before relation between two successive accesses to a memory location. The function $threadof(TS_t)$ returns the thread to which the thread segment TS_t belongs to. We use this function when updating threadset S_d. The following notation is used in the state diagram:

d	an arbitrary memory location.
$write$	write access to d.
$read$	read access to d.
TS_{new}	$:= TS_t$, the thread segment of the current thread t accessing d is called TS_{new} in the diagram.
TS_{old}	$:= TS_d$, the thread segment of the prior access to d is called TS_{old} in the diagram.
L_t	current set of locks held by thread t.
C_d	current candidate set of locks protecting variable d.

New: Newly allocated memory location that is not yet accessed. No locksets are needed. On the first write/read access, enter state *Exclusive-Write/Exclusive-Read*.

When t executes Read(d) $\vee$ Write(d) :
$TS_d \leftarrow TS_t$
set state to *Exclusive-Read / Exclusive-Write*

Exclusive-Write: Location d is synchronized with happens-before relations and the last access was a write by a particular thread segment. No locksets are needed.

Remain in exclusive state as long as successive accesses are ordered by $\overset{hb}{\rightarrow}$-relation, since there are no concurrent accesses to d. When a write or read occurs which is parallel to previous access, enter *Shared-Modified*. Otherwise, switch to *Exclusive-Write* or *Exclusive-Read* corresponding to the type of current operation.

It is possible to reach *Race* from exclusive states, in case an access happens concurrently with another access and $L_t = \emptyset$. Transitions on empty L_t prevent false negatives in many situations. L_t is the set of locks currently held by a thread during program execution and tracking it involves hardly any overhead.

When t executes Read(d) $\vee$ Write(d) :
if $TS_d \overset{hb}{\rightarrow} TS_t$
$TS_d \leftarrow TS_t$
set state to *Exclusive-Read / Exclusive-Write*
elseif $TS_d \mid\mid TS_t \wedge L_t \neq \emptyset$
$TS_d \leftarrow TS_t$
$C_d \leftarrow L_t$
$S_d \leftarrow \{t,\ threadof(TS_d)\}$
set state to *Shared-Modified*
else
set state to *Race*

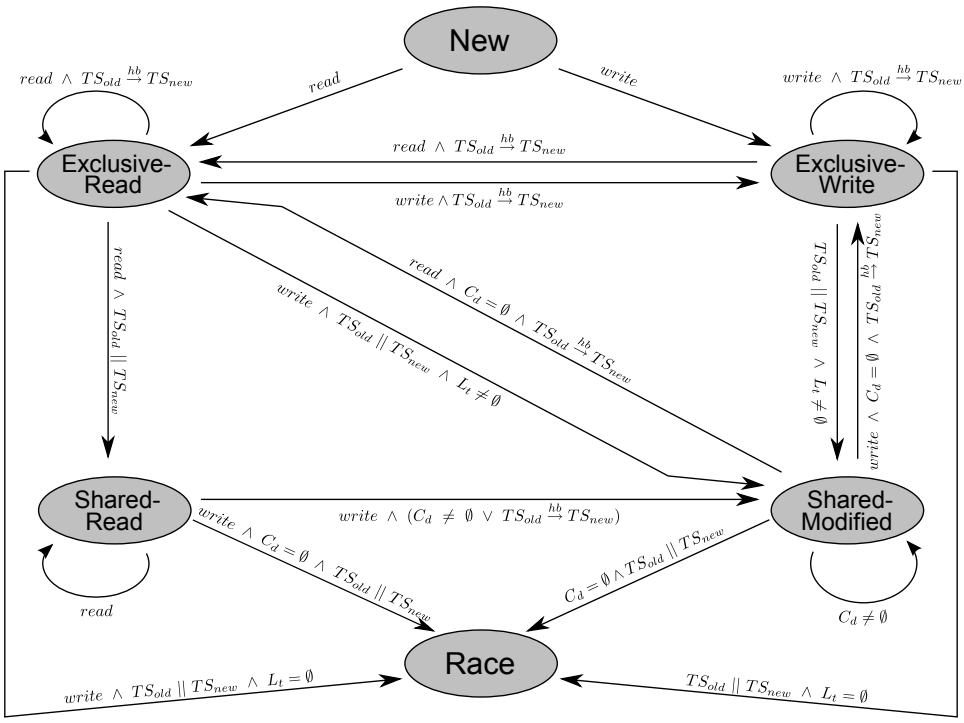


Figure 3. Extended memory state machine.

Exclusive-Read: Similar to *Exclusive-Write* except that the last access was a read operation. We presume that location d is synchronized with happens-before relations and no locksets are needed.

When a parallel access occurs, this is potentially a race except in the following cases:

- On a read operation, we enter *Shared-Read*, because parallel reads are not considered a data race. At this moment the happens-before chain is broken. Thread segment TS_d is kept to be used in *Shared-Read*.
- On a write operation, if the thread does hold any lock, we assume that from now on, variable d is protected by locks and we enter *Shared-Modified*.

In all other cases, we report a race.

When t executes Read(d) :
if $TS_d \mid\mid TS_t$
keep TS_d
$C_d \leftarrow L_t$
$S_d \leftarrow \{t,\ threadof(TS_d)\}$
set state to *Shared-Read*
else
set state to *Exclusive-Read*

When t executes Write(d) :
if $TS_d \xrightarrow{hb} TS_t$
$TS_d \leftarrow TS_t$
set state to *Exclusive-Write*
elseif $TS_d \mid\mid TS_t \wedge L_t \neq \emptyset$
$TS_d \leftarrow TS_t$
$C_d \leftarrow L_t$
$S_d \leftarrow \{t,\ threadof(TS_d)\}$
set state to *Shared-Modified*
else
set state to *Race*

Shared-Read: Location d is concurrently accessed by multiple threads, but all accesses are reads. *Shared-Read* allows parallel reads. We enter this state from *Exclusive-Read* when a read results in multiple concurrent accesses.

In this state, thread segments are not updated. We track only the lockset C_d, which is initialized to L_t and the threadset to see if a variable is shared between threads. The lockset is updated for every access. If a write occurs, enter *Shared-Modified*, except when this write is in parallel with the TS_d stored in *Exclusive-Read* and no lock protecting it. This is the only case of reporting a race in *Shared-Read*. Since we do not update the thread segment in *Shared-Read*, the thread segment in *Exclusive-Read* is stored at the point where the happens-before chain is broken. Then, by the write operation causing to leave *Shared-Read*, the thread segment of the writing thread and the stored TS_d are compared. If there are parallel accesses and the lockset is empty, we enter *Race*. This increases the chance of detecting races raised in *Shared-Read*, unlike many Eraser-style tools that lack the ability to detect races for shared-read data.

When t executes Read(d) :
$C_d \leftarrow C_d \cap L_t$
$S_d \leftarrow S_d \cup \{t\}$
set state to *Shared-Read*

When t executes Write(d) :
if $TS_d \xrightarrow{hb} TS_t$
$C_d \leftarrow C_d \cap L_t$
$S_d \leftarrow S_d \cup \{t\}$
set state to *Shared-Modified*
else
set state to *Race*

Shared-Modified: Location d is concurrently read and written by multiple threads. We presume that variable d is protected by the locks in C_d. If it is entered from an exclusive state, the lockset C_d is initialized to L_t. If this state is entered from *Shared-Read*, the lockset is passed over from this state. In addition to threadset, both lockset and thread segments are tracked.

If the lockset C_d is empty, then d is obviously not correctly synchronized with locks. So we check if other synchronization patterns impose any happens-before relations. If not, we generate an error and enter the *Race* state. If there are $\stackrel{hb}{\rightarrow}$ relations, we return to an exclusive state. This speeds up the algorithm because it reduces the overhead for locksets analysis for all accesses during program execution.

Note that this is the only state where both the happens-before relations and the locksets are analyzed.

When t executes Read(d) $\vee$ Write(d) :
  $C_d \leftarrow C_d \cap L_t$
  if $C_d = \emptyset$
    if $TS_d \stackrel{hb}{\rightarrow} TS_t$
      $TS_d \leftarrow TS_t$
      set state to *Exclusive-Read / Exclusive-Write*
    else
      set state to *Race*
  else
    set state to *Shared-Modified*

Race: A potential race is detected and reported. Introducing this separate state is useful, because once the race is reported, the tool does not spend time on this memory location any more.

4.2.4. Discussion of MSM-short. As Figure 3 depicts, the main idea is to avoid entering a shared state until the happens-before analysis shows that there are concurrent accesses to a memory location. Threadset and lockset tracking are performed only in shared states. No thread segment tracking is performed in *Shared-Read*. Only state *Shared-Modified* requires both lockset updates and happens-before analysis. Tracking both locksets and thread segments for each access during program execution can be quite expensive in both time and space. For this reason, the happens-before analysis in *Shared-Modified* is deferred until the lockset of a location is empty. That is, we do not track the thread segments until the lockset would report a race, leading to performance improvement. If there is a happens-before relation, we return to one of the exclusive states.

Separate *Exclusive-Read* and *Exclusive-Write* states are beneficial for several reasons. The state machine can distinguish a read after a write or a write after a read. We have more information about the accesses in the past, making the detector work more precisely. In addition, this distinction helps the detector to handle races that could happen only once during initialization time[8], unlike the shortcoming in Eraser-style detectors.

Compared to MSM-long [8], which has two different *Shared-Modified* states, MSM-short is more sensitive. The two distinct states in MSM-long were introduced to defer race warnings. It is assumed that in long-running applications, races on a memory location happened several times. So, in cases when MSM-long is not sure whether the observed potential race is a real race, it waits until it is repeated. In opposite, MSM-short will warn about races immediately by the first indication of incorrect synchronization.

4.2.5. Limitations. Our memory state model is a compromise between detecting all races and reporting too many false positives. There are some special cases in which a false negative occurs. One scenario is when a variable `X` in *New* state is initialized by a thread unprotectedly. There is race condition when a second thread writes to `X` concurrently. If the second thread holds any unrelated lock, this race is missed. Note, that by handling inter-thread notification events, the false negatives are significantly reduced. In addition, reducing the false positive rate even further is a continuing challenge.

5. Precise Happens-Before Detection

Reconstructing the precise happens-before relations is a crucial task for our race detector. If happens-before relations between thread segments are missed, the detector considers them to be parallel although they were correctly synchronized, thus producing false positives. On the other hand, if we add false relations to the happens-before graph, the detector loses accuracy, causing false negatives.

We discovered two problems which led to inaccuracies in the happens-before graph. One is caused by lost signals and the second one is caused by spurious wake ups. Spurious wake-ups result from the usage of only one condition variable to signal several different conditions at the same time. These issues were not handled in other approaches, e.g. [13], that extend data race detection for condition variables.

5.1. Lost Signal Detector

The standard technique for happens-before detection by intercepting library calls works fine with Fork/Join and Barriers, as synchronization primitives are called explicitly. That is not always the case when using condition variables. In Section 2.2, we discussed a typical synchronization pattern using condition variables. The problem is that condition variables are stateless and signals are not accumulated. If the waiting thread is scheduled after the

```
X++
lock(l1)
  FLAG1 = 1
  signal(CV)
unlock(l1)
```
(a) S_1

```
Y++
lock(l2)
  FLAG2 = 1
  signal(CV)
unlock(l2)
```
(b) S_2

```
lock(l1)
  while(FLAG1!=1)
    wait(CV)
unlock(l1)

X--
```
(c) W_1

```
lock(l2)
  while(FLAG2!=1)
    wait(CV)
unlock(l2)

Y--
```
(d) W_2

Figure 4. Several signaling and waiting threads.

signaling thread, the signal will be *lost* and the waiting thread will never call the corresponding `wait()`. Thus, the detector *does not know* that synchronization has taken place.

Our detector searches for while-loops in the binary program code which contain a call to the `wait()` library function. All these loops are instrumented so that, whether a call to `wait()` is executed or not, does not matter anymore. This way, the race detector will not miss any signals. Details on how we find those while-loops can be found in Section 6.3.

This approach is working very well, since the code to correctly wait for a signal (as depicted in Figure 1(b)) always follows specific rules:

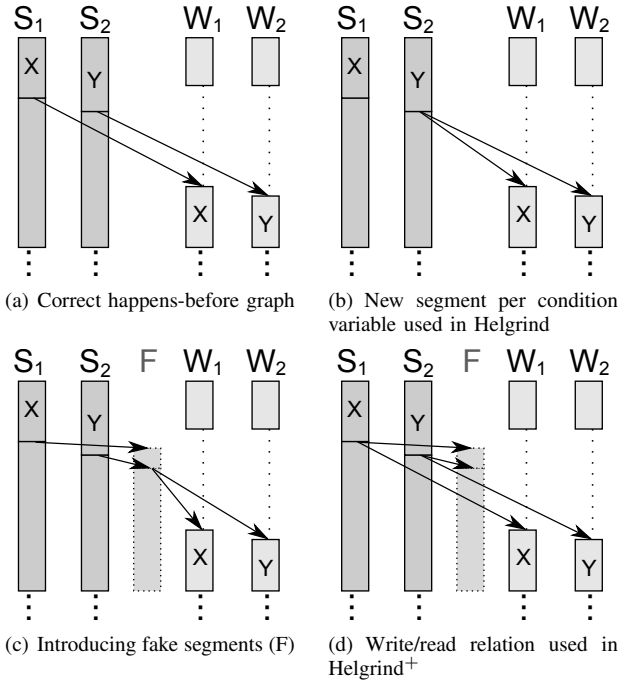
- The `wait()` library function is called within a loop, when a specific condition was not met
- It is assured that at the time the program leaves the loop, the signal was definitely sent
- Locks are used to protect the condition variables

A similar workaround to address this problem is source code annotation [16]. The major drawback of this method is the inconvenience of recompiling the source code and of course, the source code must be available. Our approach uses dynamic instrumentation, which means we do not need the source code to handle these situations.

On the other hand, our approach can detect *where* a signal should be waited for. But it does not know exactly, *which* signal to wait for, i.e. in the lost signal case, when the loop body is not executed, it is hard to find out, what the parameters of the `wait()` function are. The correct parameters are also crucial to set up the happens-before relation correctly. Our approach does a stack walk to extract the parameter set.

5.2. Spurious Wake ups

Figure 4 shows a situation with two pairs of signaling and waiting threads. Each pair accesses a different set of data. But only one condition variable was used for conceptually two different signals. This way, a signal would wake up both pairs which means one signal would spuriously wake up a wrong thread. The correct happens-before relations are shown in Figure 5(a). There are no



(a) Correct happens-before graph

(b) New segment per condition variable used in Helgrind

(c) Introducing fake segments (F)

(d) Write/read relation used in Helgrind$^+$

Figure 5. Happens-before graphs generated by different tools. X and Y denote variable accesses in a thread segment.

races since accesses to identical data are strictly ordered. The algorithm depicted in Section 4.1 is not able to handle this situation correctly, as it is only capable of storing one thread segment per condition variable. This would result in a happens-before graph like in Figure 5(b). By contrast, our original algorithm would report a race for variable X due to the false happens-before graph.

In [16] *fake thread segments* are introduced to compensate for this kind of situation. This results in a graph that is still incorrect (Figure 5(c)). By introducing the fake segments, happens-before edges are redirected to and from fake segments. According to this graph, waiting threads W_1 and W_2 are synchronized after both signaling threads S_1 and S_2. But if by a mistake W_1 accesses Y instead of X, the race between S_2 and W_1 on Y is not detected and we would have a false negative based on this schema.

5.3. Write/Read Relations

You can also see in Figure 4 that the difference between the pairs lies in the condition they check. This condition can be indirectly derived from the variables used in the condition, `FLAG1` and `FLAG2`. These variables are protected by locks.

We say that there is a write/read-relation between thread segments, if there is a variable which is modified by one thread segment shortly before sending a signal and the other thread segment reads this variable shortly before receiving a signal. That means that there is a write/read-relation between signaling thread S_1 and waiting thread W_1 on variable `FLAG1`. These write/read-relations describe the happens-before relations of threads more accurately. Our race detector uses the happens-before relations imposed by write/read relations whenever they can be detected. Otherwise, fake thread segments are still used to back off. Figure 5(d) shows the happens-before graph constructed by Helgrind$^+$.

To extract the variables potentially used in a write/read-relation, we record all variables modified between `lock()` and `signal()` and bind them to the current thread segment at signaler side. On the waiter side, all read accesses between `lock()` and `while(expression)` (including the expression) are observed and compared with the recorded accesses. This comparison marks the thread segment of the real signaler. The thread segments of the signaler and the waiter are bound in a happens-before relation based on this write/read relation.

This extension in our race detection algorithm is depicted in Figure 6.

Whenever thread t enters a lock-protected region with `lock()`, we start to record all following read operations into R_t and all write operations to W_t. R_t and W_t are reset when we leave a lock-protected region with `unlock()`. In a lock-region, a call to `signal()` causes the detector to map all modified variables since the beginning of the lock-region to the current thread segment. This is done by $DS(d)$ in the algorithm, i.e. $DS : d \mapsto ts$ where ts denotes the thread segment of last signaler which modified d. Additionally, a fake thread segment is created which is ordered after every signaler segment. When a `wait()` is called or annotated, all recorded read operations R_t are checked if there is a mapping $DS(d)$ indicating any write/read relation on d. If it exists, a happens-before relation between waiter and the real signaler is constructed. Otherwise, we have to back off with the fake segment, which was created by our `signal()`-handler previously. As far as we know, the solution presented to proper handling of condition variables without source code annotation is beyond the current state of the art.

After t executes Lock(l) :
$R_t \leftarrow \emptyset$
$W_t \leftarrow \emptyset$

After t executes Unlock(l) :
$R_t \leftarrow \emptyset$
$W_t \leftarrow \emptyset$

After t executes read(d) :
$R_t \leftarrow R_t \cup \{d\}$

After t executes write(d) :
$W_t \leftarrow W_t \cup \{d\}$

Before t executes Signal(cv) :
foreach d in W_t :
$DS(d) \leftarrow TS_t$
$TS_{cv} \leftarrow NewSegment(TS_t, TS_{cv})$
$TS_t \leftarrow NewSegment(TS_t)$

After t executes Wait(cv) :
foreach d in R_t :
if $DS(d)$ exists:
$TS_t \leftarrow NewSegment(TS_t, DS(d))$
if $\forall x \in R_t : DS(d)$ not exists:
$TS_t \leftarrow NewSegment(TS_t, TS_{cv})$

Figure 6. Basic detection algorithm and lock-set update rules with write/read relation.

6. Implementation

6.1. Valgrind

Our race tracker is a modification of Helgrind, which is a tool based on Valgrind [5], [6], [7]. Valgrind is a *disassemble and resynthesize* dynamic instrumentation framework for Linux executables. The framework translates a binary into a platform independent intermediate representation (IR). Helgrind$^+$ instruments this intermediate representation and hands it back to the framework which resynthesizes machine code from the instrumented IR. Instrumentation is done just-in-time and does not need the source code of the program. Valgrind is also able to intercept calls to library functions through dynamic library preloading. This makes the tools based on the Valgrind framework powerful and flexible for all kinds of runtime checking.

Helgrind$^+$ supports the POSIX Threads API for multithreading. The GCC implementation of OpenMP is indirectly supported as it ultimately uses POSIX threads.

6.2. Shadow memory

Helgrind must maintain state information not only for each thread, but also for each data used by the program. As Helgrind is unaware of high level data structures, it can only operate on memory locations with granularity at byte level. State information about memory locations can be stored with shadow memory [17], that is, for every byte

of memory used in the program, a shadow word is stored. Valgrind provides an extra memory pool for the shadow memory and other data structures such as mutex or thread segment information so that Helgrind's data will not mix with the data of program. Original Helgrind 3.3.1 used 32-bit shadow words, which proved to be too small for our race checker, so we use 64-bit shadow values.

As Figure 7 shows, we use the 64 bits differently depending on in which state the memory location is. The state is stored in the first three bits. Not all information are relevant in every state. Threadsets for example are not tracked in exclusive states, so we do not need to store them in these states. An identifier of the thread segment is stored in the remaining bits of the first 32-bit word and the candidate lockset and threadset are stored in the second 32-bit word.

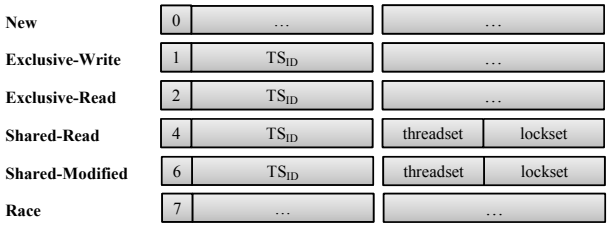


Figure 7. Structure of 64-bit word shadow value and state encoding for MSM-short

6.3. Instrumenting while-loops in Intermediate Representation (IR)

As described earlier, we can catch all lost signals by instrumenting the while-loops which enclose a call to the `wait()` library function. Loops are represented by conditional jumps in machine code, which is visible in Valgrind's IR. There are many conditional jumps in a program, so we had to focus on a specific patterns, which were typical for `wait()`-enclosing while loops. The loop condition is usually not very complex (e.g. just evaluate one flag) and the loop body often just contains a call to `wait()`. That means that the jumping distance is very small: We only considered jumping distances between 12 to 100 instruction bytes. The `wait()`-enclosing while loops of programs in our experiments were all smaller than 100 instruction bytes, except only in one case.

The next step is to look inside the loop body and detemine whether there is a call to `wait()`. Helgrind$^+$ does that by searching for jumps to the `wait()` function address in the IR of the loop body. The function address can be found in the relocation tables of the Linux ELF file [18]. We had to modify Valgrind so that Helgrind$^+$ was able to request arbitrary machine code blocks to be translated into IR. Additionally, Valgrind had to read out the relocation tables of the Linux ELF file [18]. We implemented this function only for the x86 and amd64 ELF type.

Finally, when a `wait()` was found in the loop body, the parameters are of interest to determine which signal should be received. We simulate a portion of the program stack to determine the parameters, which does not work well for few cases. When we are not able to get the parameters, we back off by letting the thread synchronize to signals which has been recently sent.

The presented method cannot cope with wrapper functions, that encapsulate the call to `wait()`. Fortunately, the compiler inlines the wrapper function during code optimization, which can be enforced by the `-O2` compiler switch. In our experiments, these wrapper functions were all inlined and opposed no problem to the lost signal detector.

7. Experiments and Evaluation

In the following section, we present our experiences with the extended Helgrind$^+$ and evaluate our approach by applying it to a number of applications. We show that with the extended techniques, we are able to detect races more precisely while reducing false positives. The overhead caused by Helgrind$^+$ is reasonable, so that it can be used by applications from a variety of domains to produce more accurate reports.

All experiments and measurements in this section were conducted on a machine with 2x Intel XEON E5320 Quadcore at 1.86GHz, 8 GB RAM, running Linux Ubuntu 8.04.1 x64. All programs were compiled with gcc 4.2.3. No source code annotation was used. We employed different detectors for our experiments and measurements. *Intel TC* denotes the commercial tool Intel Thread Checker 3.1 [19]. *HG-64* denotes a 64 bit version of recent development version from Helgrind repository[2], revision 8520. It is the improved version of the recently released *Helgrind 3.3.1*. *HG-64-precise* denotes the initial 64 bit version with our extension for lost signal detection and write/read relation (precise mode). *HG$^+$-long* symbolizes Helgrind$^+$ based on MSM-long and *HG$^+$-long-precise* is in precise mode. Similarly, *HG$^+$-short* is based based on MSM-short.

7.1. Test Suite — data-race-test

We applied Helgrind$^+$ to unit test cases provided in *data-race-test* [3], a test suite for Helgrind. The test suite aims to create a framework to check a race detector and evaluate the effect of each test case on the tool. This suite provides more than one hundred test cases that implement different scenarios which could occur while running

multi-threaded programs. Most of these scenarios represent tricky situations, which are difficult to discover by race detectors. Currently, 79 of these test cases are categorized into two main categories: "racy" cases that involve at least one data race, and race-free cases. We examine and analyze the effect of each test case on Helgrind$^+$. Table 1 shows the result of our experiment on the test suite. All test cases are short programs implemented in C/C++ with a varying number of threads. All test cases are executed without any annotation at source code level for all detectors.

Tools	FP Cases	FN Cases	Failed Cases	Passed Cases
Intel TC	3	21	24	55
HG 64	39	8	47	32
HG 64 precise	37	9	46	33
HG$^+$ long	8	16	24	55
HG$^+$ long precise	3	15	18	61
HG$^+$ short	17	5	22	57
HG$^+$ short precise	11	4	15	64

Table 1. Comparing the results of Helgrind$^+$ with other race detectors on the test suite. FP and FN denote False Positives and False Negatives, respectively. The sum of both are failed test cases.

Notice that the result of Helgrind$^+$ based on MSM-short with the enabled features of lost signal detection and write/read relation (HG$^+$-short-precise) is significantly improved. Since each test case is considered to be a short program, the results are as expected. For HG$^+$-short, 22 test cases out of 79 failed (17 false positives and 5 false negatives). By enabling the precise mode (HG$^+$-short-precise), 6 false positives and 1 false negative are removed, i.e., 64 cases of 79 cases pass. Two test cases provided in the test suite are similar to the example provided in Figure 4. These tests pass only in HG$^+$ in the precise mode, while all other detectors fail. Also, all cases regarding lost signal detection passed in HG$^+$ precise mode. Most cases that failed had user defined synchronization; they are difficult to detect, since they do not follow the standard pattern given by synchronization primitives. A few test cases produced benign races.

When using HG$^+$-long, the false positives are considerably reduced to 8 but on the other hand, the false negatives increase to 16. This indicates that HG$^+$-long is not adequate to find the masked races in short programs in the test suite. Applying the precise mode increases the accuracy of HG$^+$-long and reduces both false positives and false negatives.

Comparing the result of Helgrind$^+$ to the initial version of Helgrind (HG-64), a significant improvement is the number of passed cases, which doubled in the best case. Applying HG-64 in precise mode hardly improved the result. This is because of the Eraser-like memory state model used in HG-64 is rather simple.

In addition, we compared the behavior of Helgrind$^+$ with the commercial tool Intel TC for the test suite. The false positives by Intel TC are as few as in HG$^+$-long. But on the other hand, Intel TC did not detect races in 21 test cases, whereas Helgrind$^+$ masked only 4 racy cases for HG$^+$-short-precise.

The results on unit test confirm that Helgrind$^+$ is able to discover masked races more accurately compared to other race detectors mentioned here. Especially when using HG$^+$-short-precise, the fault detection ratio is promising for small and short-running applications.

7.2. PARSEC Benchmark

We evaluate our detector with the recently released PARSEC 1.0 benchmark [4]. PARSEC contains twelve diverse multi-threaded programs from different areas such as computer vision, video encoding or financial analytics.

Table 2 depicts the result of the experiments on different tools. We analyzed the result of executions with two threads per application. The empirical study in [20] implies that most concurrency bugs manifest themselves with only 2 threads. Also, Valgrind schedules threads in a more fine-grained way than the operating system would do. Consequently, we assume that many races can already be observed with 2 threads. The presented numbers are distinct program code locations which produced at least one potential data race, which are called *racy contexts*. Because of the large memory consumption and computational cost, we did not perform simulations with the native input set. Instead, we used the `simsmall` or `simmedium` inputs for all simulations and ran each program five times, averaging the results. All numbers for read/write instructions and synchronization primitives are totals across all threads. Numbers for synchronization primitives include primitives in system libraries. *Locks* are all lock-based synchronizations including *Read-Write locks* (*rwlocks*). Barriers are barrier-based synchronizations, *Conditions* are waits on condition variables. Except for `freqmine`, which uses OpenMP, all programs used the standard Pthread library for parallelization.

It should be mentioned that the authors of the PARSEC Benchmarks claim the programs to be race free, however we cannot be absolutely sure that they are. Table 2 provides the false positives under the assumption that the programs are race free. Future work will be to analyze the warnings to see whether some of them are true positives.

In Table 2, the number of warnings produced by HG$^+$-long-precise is reduced considerably. That is because HG$^+$-long is based on the assumption that a race access pattern will be repeated again. Since the PARSEC

Program	LOC	Instructions (10^9)		Synchronization Primitives			Racy Contexts				
		Reads	Writes	Locks	Barr.	CVs	Intel TC	HG 64	HG 64 precise	HG$^+$ long precise	HG$^+$ short precise
blackscholes	812	0.092	0.045	0	2	0	0	0	0	0	0
bodytrack	10,279	0.425	0.102	35,849	215	90	1	51	51	0	2
canneal	4,029	0.435	0.187	88	0	0	6	1	1	0	1
dedup	3,689	0.658	0.254	18,436	0	3,536	-	3	0	0	0
facesim	29,310	9.632	4.191	10,460	0	1,795	-	128	131	37	115
ferret	9,735	0.005	0.002	6,660	0	10	0	111	13	1	6
fluidanimate	1,391	0.584	0.144	923,750	0	0	0	56	56	0	0
freqmine	2,706	0.744	0.283	78	0	0	1,011	228	224	28	159
streamcluster	1,255	1.795	0.033	146	12,998	34	2	19	20	0	0
swaptions	1,494	1.414	0.365	78	0	0	0	0	0	0	0
vips	3,228	0.758	0.199	10,575	0	2,698	0	105	103	41	49
x264	40,393	0.500	0.204	1,339	0	157	-	734	702	173	44

Table 2. Number of racy contexts reported on PARSEC 1.0 benchmarks. All programs are executed for input set `simsmall` except `swaptions` and `streamcluster` that is for `simmedium`.

benchmarks are long-running applications, the results are acceptable compared to other detectors. HG$^+$-short-precise is more accurate and targeted for short-running applications, therefore producing more warnings than expected. Comparing the result of Helgrind$^+$ to the initial version of Helgrind (HG-64), a remarkable reduction in the number of false warnings can be observed. Running HG-64 in precise mode did not improve the results. Only in a few cases minor improvements are achieved. As mentioned before, this is due to the simple memory model used in HG-64.

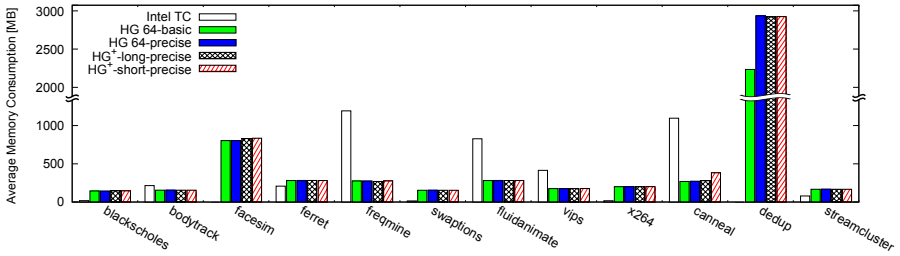
We also apply Intel TC to PARSEC. Two packages of PARSEC, i.e. `facesim` and `dedup`, crashed while running with ITC due to high memory consumption. Also `x264` may not be instrumented properly by Intel TC, since the execution time of instrumented code was equal to non-instrumented code using the same memory consumption. For this reason, we excluded `x264`, along with `facesim` and `dedup` from our discussion when using Intel TC. But the results of Intel TC on other packages are approximately comparable with HG$^+$-long, producing a small number of warnings. It should be noted that `vips` uses the `GLIB` [21] library that is not supported by Valgrind. That is why HG$^+$ produced many alarms, since it is not able to intercept all function calls inside `vips`. The outcome on PARSEC shows that Helgrind$^+$ with the new features reports races only in cases where they actually occurred. However, if any correct synchronization is implied, no race is reported. This reduces false positives and makes real-world applications easier to handle.

7.3. Performance

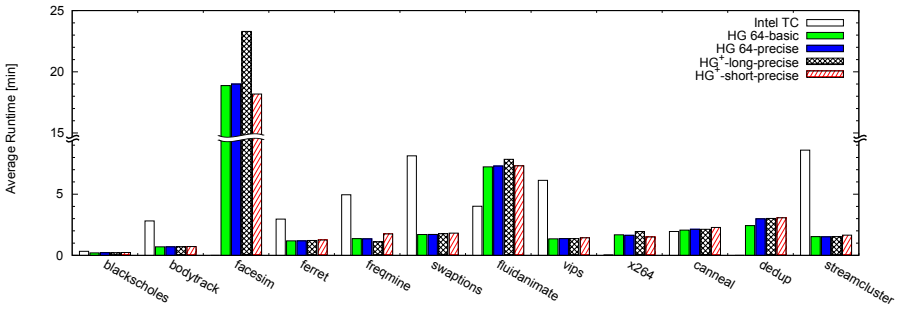
To validate our method, we compared the runtime behavior and the memory requirements of detectors on the basis of PARSEC benchmark. Firstly, we measured the memory usage of instrumented code running by different detectors. All measurements are average values of five executions with two threads using the `simsmall` or `simmedium` inputs for all simulations. We used `simmedium` inputs for `streamcluster` and `swaptions`, as the runtime with `simsmall` was too short. Figure 8(a) depicts the average memory consumption on PARSEC benchmarks by different detectors. The memory consumption of Helgrind$^+$ in different modes is approximately constant. There is little overhead caused by the extended memory state machines or the new features implemented for the precise mode. Compared to the basic version of Helgrind (HG-64), the overhead by Helgrind$^+$ is only significant in the case of `dedup`. Intel TC caused fairly large memory overhead, especially in the case of memory-intensive programs, e.g. `facesim` or `dedup` that crashed while with Intel TC. Memory overhead in Helgrind$^+$ is small enough that applications with higher memory requirements are still testable.

The execution time of instrumented code versus the actual execution time is typically slowed down by a factor of 10 to 60 on Helgrind$^+$. We measured the execution time of instrumented code on different detectors. The measurements are shown in Figure 8(b) for the PARSEC benchmark. There is minor overhead of Helgrind$^+$ over basic Helgrind. In the worst case, `facesim` on HG$^+$-long-precise increases the execution time significantly. In other cases, different version of Helgrind$^+$ in different modes have approximately equal execution times. As the figure shows, compared to Helgrind$^+$, Intel TC increased the execution time remarkably. On average, the slowdown is a factor of 2 compared to Helgrind$^+$.

Overall, the results confirm that the methods presented in this work cause a modest overhead. They are fast enough and need less memory for different kind of applications. The fault detection ratio is comparable to other tools, and reports are more accurate for the case of short-running as well as long-running applications.



(a) Memory consumption on PARSEC 1.0



(b) Execution time on PARSEC 1.0

Figure 8. Memory consumption and execution time on PARSEC 1.0 by different detectors.

8. Related Work

Many recent approaches have combined the happens-before relation and lockset algorithm [11], [12], [13], [2], [22], [14], but they still produce many false alarms or miss races. Our dynamic detection approach is also based on the lockset algorithm and happens-before analysis, however, the employed heuristics and the combination of these methods differentiate our detector from other approaches. We propose a new memory state model that is optimized for short-running programs. Compared to the memory state model presented in our previous work [8], the current model is more sensitive to potential races and is adequate when a race pattern occurs only once, as in short-running applications or during development. The previous state model, on the other hand, is based on the assumption that a race pattern is likely to be repeated as in the case of long-running applications or integration testing. It is therefore somewhat less sensitive, but produces fewer false positives.

Detectors in [13], [2], [16] produce a lot of false warnings and miss races, since the ordering induced by inter-thread event notifications (via condition variables) is not taken into account. Our work eliminates almost all false positives in this situation, while detecting all races (false positives and false negatives caused by lost signals and spurious wake-ups are detected, see Section 5). The write/read relation is a new method used to proper handle condition variables. In [23], a dynamic software technique is used to identify the user-defined synchronizations exercised during program execution. This technique is not able to handle condition variables, while our method is more general; it can be applied to identify the user-defined synchronizations and other synchronization constructs, e.g. spin-locks.

9. Conclusion and Future Work

An adequate memory state model is essential for accurate race detection. In this paper, we combined the lockset algorithm with a happens-before analysis in a new and more efficient way. Our approach achieves a higher accuracy by making the race detector adaptable to short-running and long-running applications.

Several other improvements over state-of-the art were achieved. Our analysis was extended to work more precisely in situations where certain synchronization patterns were caused by condition variables. By taking lost signals and the write/read relation into account, we reduced the number of false positives and false negatives. Our empirical data substantiates that our improvements lead to better results, with almost no increase in execution time and memory overhead.

There are many opportunities for future work to improve Helgrind$^+$. For example, program analysis might be used to automatically select the appropriate memory state machine. Extending the write/read relation to identify the user-defined synchronizations could improve the accuracy of the happens-before detection. Classification techniques

for warnings might draw upon the state machine's history. The identification of benign races is another important issue. Applying a runtime analysis that excludes variables that are only accessed by a single thread could improve performance as well.

Acknowledgments
We would like to thank Valgrind author Julian Seward for answering our questions regarding the Valgrind/Helgrind codebase, and Konstantin S. Serebryany for creating the data-race-test project, a test suite for Helgrind and collecting test programs. We also appreciate the support of the excellence initiative at the University of Karlsruhe.

References

[1] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, 1992.

[2] Valgrind-project., "Helgrind: a data-race detector," 2007. [Online]. Available: http://valgrind.org/docs/manual/hg-manual.html

[3] ——, "Data-race-test:test suite for helgrind, a data race detector," 2008. [Online]. Available: http://code.google.com/p/data-race-test/

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," Tech. Rep., January 2008.

[5] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, 2007.

[6] N. Nethercote, "Dynamic binary analysis and instrumentation," Ph.D. dissertation, University of Cambridge, UK, 2004.

[7] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," 2003. [Online]. Available: http://valgrind.org/

[8] A. Jannesari and W. F. Tichy, "On-the-fly race detection in multi-threaded programs," in *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems*. New York, NY, USA: ACM, 2008, pp. 1–10.

[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[11] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 221–234, 2005.

[12] E. Pozniansky and A. Schuster, "Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 3, pp. 327–340, 2007.

[13] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," *SIGPLAN Not.*, vol. 38, no. 10, pp. 167–178, 2003.

[14] J. J. Harrow, "Runtime checking of multithreaded applications with visual threads," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. London, UK: Springer-Verlag, 2000, pp. 331–342. [Online]. Available: citeseer.ist.psu.edu/harrow00runtime.html

[15] A. Dinning and E. Schonberg, "Detecting access anomalies in programs with critical sections," *SIGPLAN Not.*, vol. 26, no. 12, pp. 85–96, 1991.

[16] A. Muehlenfeld, "Runtime race detection in multi-threaded programs," Ph.D. dissertation, IST - Institut fuer Softwaretechnologie, Technische Universitaet Graz, May 2007.

[17] N. Nethercote and J. Seward, "How to shadow every byte of memory used by a program," *Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2007)*, 2007. [Online]. Available: http://valgrind.org/docs/shadow-memory2007.pdf

[18] T. Committee, "Tool interface standard (tis) executable and linking format (elf) specification version 1.2," 1995. [Online]. Available: http://refspecs.freestandards.org/elf/elf.pdf

[19] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, "Unraveling data race detection in the intel thread checker," 2005. [Online]. Available: http://www.isi.edu/ kintali/stmcs06/UnravelingDataRace.pdf

[20] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2008, pp. 329–339.

[21] G. D. Library, "Glib reference manual," 2008. [Online]. Available: http://library.gnome.org/devel/glib/

[22] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas, "Accurate and efficient filtering for the intel thread checker race detector," in *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. New York, NY, USA: ACM, 2006, pp. 34–41.

[23] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam, "Dynamic recognition of synchronization operations for improved data race detection," in *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2008, pp. 143–154.