On-the-fly Race Detection in Multi-Threaded Programs

Ali Jannesari and Walter F. Tichy

University of Karlsruhe
76131 Karlsruhe, Germany
{jannesari,tichy}@ipd.uni-karlsruhe.de

ABSTRACT

Multi-core chips enable parallel processing for general purpose applications. Unfortunately, parallel programs may contain synchronization defects. Such defects are difficult to detect due to nondeterministic interleavings of parallel threads. Current tools for detecting these defects produce numerous false alarms, thereby concealing the true defects. This paper describes an extended race detection technique based on a combination of lockset analysis and the happens-before relation. The approach provides more accurate warnings and significantly reduces the number of false positives, while limiting the number of false negatives. The technique is implemented in Helgrind$^+$, an extension of the open source dynamic race detector Helgrind. Experimental results with several applications and benchmarks demonstrate a significant reduction in false alarms at a moderate runtime increase.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming* ; D.2.11 [**Software Engineering**]: Testing and Debugging —*monitors, testing tools.*

General Terms

Experimentation, Performance, Reliability.

Keywords

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

1. INTRODUCTION

Two current developments, the emergence of multi-core chips and stagnating clock rates of processors, are pushing parallel computing out of the niche of numeric applications into the mainstream. Unfortunately, parallel programs may contain synchronization defects, a class of defects not easily detected because of non-deterministic interleavings of concurrent threads. This paper concentrates on the detection of data races, i.e, unsynchronized accesses to shared data. 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[18]. Because multi-threaded programs are schedule-dependent, reproducing data races is often difficult. One may not get the same execution order even with identical inputs. Moreover, programs encountering data races often do not crash immediately, resulting in mysterious and unpredictable behavior. With multi-core computers, the importance of tools that find data races automatically is significantly increased.

We propose a dynamic approach for race detection based on a synthesis of lockset and happens-before analysis. The approach provides a lower rate of false positives and smaller performance overhead. The basic idea is to consult the happens-before relation whenever the lockset algorithm indicates a possible race. The increased precision is due to a more detailed state machine in the analysis (eight possible states vs. four for a pure lockset algorithm). The approach has been implemented in Helgrind$^+$, an extension of the Helgrind tool[28]. Results on several benchmarks demonstrate a significant reduction in false positive rates compared to the original Helgrind tool, which is primarily based on the lockset algorithm.

The paper is organized as follows. Section 2 discusses related work. In Section 3 a short overview of the lockset detection algorithm, happens-before detection, and some hybrid methods are presented. Then the extended race detection approach is explained in Section 4. In Section 5, Helgrind, an open source race detection tool, is introduced. Furthermore, the extension of Helgrind and the memory layout with the shadow value technique are explained. In Section 6 the results of applying the modified tool to several benchmarks and applications are presented. In the last section, we conclude the paper with a short discussion of our results and the focus of our future work.

2. RELATED WORK

There is a substantial amount of prior work regarding detection of potential data races. Proposed solutions can be classified as static (ahead-of-time) and dynamic (on-the-fly) analyses. Static analysis considers the entire program and warns about possible races in all possible executions orders[9]. The main drawback of this approach is that it pro-

PADTAD'08, July 20–21, 2008, Seattle, Washington, USA.

duces too many false positives, as static analysis must conservatively consider all potential thread interleavings, even those that are not feasible. Another issue is that static analysis does not scale well to large programs due to path/state explosions[6]. Furthermore, static analysis has problems with dynamically allocated data. Detecting all feasible data races by static analysis is known to be an NP-hard problem[18].

Dynamic analysis scales better and reports fewer false positives compared to static analysis. However, it detects races only in the actual run, hence there is no guarantee that a program is race-free. Consequently, the program has to be tested with various inputs to cover different execution paths. There are two different methods used by dynamic race detectors: on-the fly and post-mortem. On-the-fly methods record and analyze information as efficiently as possible during program execution. Post-mortem methods record events during program execution and analyzes them later[3], or record only critical events and then replay the program[24]. The post-mortem approach is unsuitable for long-running applications that have extensive interactions with their environment. All dynamic methods add overhead at runtime, which must be traded off against detection accuracy.

Prior dynamic race detectors are based on two different techniques: lockset or happens-before analysis. Lockset analysis checks whether two threads accessing a shared memory location hold a common lock. If this is not the case, the concurrent access is considered a potential data race[26, 29, 19]. The technique is simple, can be implemented with low overhead, and is relatively insensitive to execution order. The main drawback of a pure lockset-based detector is that it produces many false alarms due to the fact that it ignores synchronization primitives other than locks, such as signal/wait, fork/join, and barriers.

Happens-before detectors[5, 24, 4] are based on Lamport's happens-before relation[12]. Happens-before analysis uses program statement order and synchronization events to establish a partial temporal ordering of program statements. A potential race is detected if two threads access a shared memory location and the accesses are temporally unordered. The happens-before technique can be applied to all synchronization primitives, including signal/wait, fork/join, barriers, and others. It does not report false positives in the absence of real data races. However, this approach may miss races, i.e. produce false negatives, as it is sensitive to the order of execution and depends on whether or not the scheduler generates a dangerous schedule. In fact, happens-before detection produces more false negatives than lockset-based detection[20]. Happens-before analysis is also difficult to implement efficiently and does not scale well.

Recent race detectors combine happens-before and lockset-based techniques to get the advantages of both approaches and improve accuracy and performance[30, 22, 20, 28, 25, 11]. The combination was originally suggested by Dinning and Schonberg[7].

Dynamic race detectors differ in how they monitor program execution. Many detectors record load/store instructions and shared-memory references[26, 11]. Other race detectors work with the bytecodes of object-oriented programming languages[30, 20, 5], making them independent of programming language and source code. Some race detectors modify the source code in order to instrument memory accesses and synchronization instructions[22].

3. RACE DETECTION

In this section, we introduce the two main race detection methods, followed by a more detailed description of the algorithms and a discussion of their limitations.

3.1 Lockset-Based Detection

Lockset analysis is based on the observation that each shared memory location accessed by two different threads should be protected by a lock, if at least one access is a write. The detector examines all locations where a shared variable is accessed and checks whether the shared variable is protected by a lock. If the variable is not protected, a warning is issued. The algorithm is simple and easy to implement. Eraser was the first implementation, which worked with programs using the POSIX-Threads library. In this implementation, the basic synchronization primitive is a *mutex*, with methods to acquire and release it. The pseudo code of the basic lockset algorithm or so-called Eraser algorithm[26] is shown in Figure 1. During program execution, the Eraser algorithm maintains for each shared variable d a set of locks C_d that contains the intersection of the sets of locks that were held during all accesses to variable d. The details of the algorithm appear in [26].

Let L_t be the set of locks held by thread t.

For each variable d, initialize C_d to the set of all locks.
On each access to d by thread t,
    set $C_d := C_d \cap L_t$;
    if $C_d = \{\}$, then issue warning.

Figure 1: Basic lockset-based algorithm.

The main drawback of the Eraser algorithm is that it produces too many false alarms, because it can only process lock operations and fails when other synchronization primitives or user-defined synchronizations are used. For example, numeric algorithms often consist of several steps separated by barriers. If memory accesses of two separate steps overlap, Eraser would falsely report races, even though they are prevented by the barriers. An algorithm based on the happens-before relation would not report any false positive in this situation.

A single write operation followed by read-only accesses is a frequent case which lockset detectors must handle. Consider a shared variable that is written once by a main thread and subsequently read by worker threads. It appears that no lock is needed. However, a pure lockset detector would report a race in this case. To handle this situation, Eraser uses the state machine in Figure 2. The idea is to defer error reports until a second thread performs a write operation (and thus reaches the *Shared-Modified* state in the diagram).

However, the state machine in Figure 2 may mask a race (produce a false negative). The program listed in Figure 3 contains a simple undetected data race between parent and worker thread. The parent may write the variable `GLOB` before the worker thread can read it. In this case, the state machine ends up in state *Shared-Read* without issuing a warning. With the opposite order of execution, Eraser would report a race. The basic problem is that there is no synchronization between parent and worker. (As there is no happens-before relation between the parent and worker regarding the read/write operations, a pure happens-before detector would detect the masked race.) This kind of false

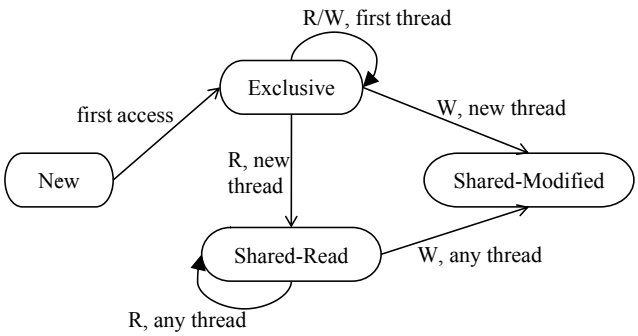

Figure 2: Possible states for a memory location. After allocation, the location is in the state *New*. During the first write, it enters state *Exclusive* and leaves this state only if another thread reads or writes the location. An error is reported if the state *Shared-Modified* is reached and the lockset is empty.

```
int GLOB = 0;

//worker Thread
void *worker(void*){
  printf("GLOB=%d\n", GLOB);
  return NULL;
}

//parent thread
int main(int argc, char **argv){
  pthread_t threadid;
  pthread_create(&threadid, NULL, worker, NULL);
  GLOB = 1;
  pthread_join(threadid, NULL);
}
```

Figure 3: A simple example causes false negatives in Eraser-based race detectors.

negative also exists in other race detectors based on this state diagram [11, 30].

We extend the state machine such that it handles the above case correctly. More details are discussed in section 4.

3.2 Happens-before Detection

The happens-before relation orders events. Lamport introduced the happens-before relation to define a partial ordering of events in distributed systems[12]. According to his definition, an event a happens before an event b if a happens at an earlier time than b.

Based on this relation, a potential race has occurred if we observe two distinct events a and b that access the same memory location, where at least one event is a write, and neither a happens before b nor b happens before a, i.e., there is no temporal ordering of a and b. There are two different ways to implement the happens-before relation: *vector clocks* and *thread segment graphs*. Vector clocks have been widely used in previous works[30, 22, 20]. For details about the computing the happens-before relation with vector clocks refer to [14]. Some other detectors[11, 28] use thread segment graphs.

Compared to lockset-based detection, happens-before analysis has a lower rate of false positives, but causes significant overhead and is difficult to implement. Moreover, it is sensitive to scheduling. 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[30, 22, 20, 11, 28] use the happens-before relation to determine the order imposed by synchronization primitives in order to avoid some false positives of the lockset algorithm. These approaches still report many false positives and even miss some races (see the example in Figure 3 discussed in Section 3.1).

A simple scenario where happens-before analysis could prevent a false warning is the following. A thread allocates a memory location, initializes it by setting it to some value, and then creates a second thread to work on the data. Then the first thread waits for the second thread to join, before it uses the memory location again. Thus, the memory location is shared between threads, but at any point in time only one thread accesses it. A useful way of viewing this sequence of events is that by creating a new thread, the ownership of the memory location is passed to the second thread until it terminates, at which time the first thread receives ownership back. This view can be implemented by introducing the concept of a *thread segment*. A thread segment delineates time in addition to thread identity. By utilizing thread segments or *thread segment identifiers* (TS_{ID}) in the algorithm instead of merely using thread identity, one can distinguish accesses that cannot happen concurrently.

Figure 4 shows a simple scenario illustrating thread segment graphs. The graph represents a partial order for thread segments. All segments on a path are totally ordered with respect to time. If there is no path between two segments, they are considered to be concurrent. When executing fork operations, new thread segments are created. In this manner, events are ordered and as long as memory accesses to shared locations occur in segments with no overlap, there is a happens-before relation and no race is possible. Figure 4 illustrates the happens-before relation between two successive accesses to x. $\mathrm{TS}_{11} \overset{hb}{\rightarrow} TS_{21}$, because of the existing path between segments TS_{11} and TS_{21}. The two accesses to x are temporally ordered.

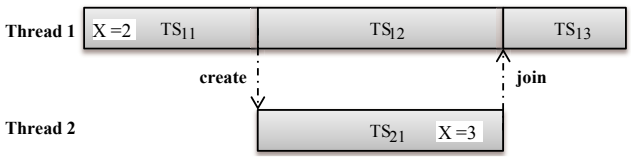

Figure 4: A thread consists of thread segments separated by create and join operations. Memory accesses that occur in non-overlapping thread segments are exclusive even if they belong to different threads.

A pure lockset base detector produces a false alarm in the given scenario, although only a single thread uses the shared location at a time. The happens-before relation is used by race detectors VisualThreads, Helgrind and RaceTrack [11, 28, 30]. The underlying idea is that instead of a thread being owner of a shared variable that is in the exclusive state, the variable is now associated with a *thread segment identifier*.

Whenever another thread accesses the variable, it is checked whether the thread segments overlap or not. If not, the new thread segment becomes the new owner of the variable and its state remains unchanged.

Thread segments can be added to the lockset algorithm as follows:

1. When location d is in state *Exclusive*, associate it with the thread segment identifier (TS_{ID}) of the current thread instead of the thread identifier.
2. If location d is in state *Exclusive* to thread segment TS_i and is accessed by TS_j, and TS_i happens before TS_j in the graph, then instead of marking d with one of the shared states, associate d with TS_j. The state of d remains *Exclusive*.

The additional steps reduce some false positives, but there is still room for improvement. Due to checking the happens-before relation only in state *Exclusive*, the algorithm reverts to a pure lockset algorithm when leaving this state. Figure 5 demonstrates an example where signal and wait produce false positives despite the additional steps. Support for barriers as discussed in section 3.1 is also missing.

```
int COND = 0;
int GLOB = 0;
static pthread_cond_t  CV = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t MU = PTHREAD_MUTEX_INITIALIZER;

//worker thread
void *worker(void*){
  usleep(1000); //worker blocks
  pthread_mutex_lock(&MU);
    COND = 1;
    GLOB = 2;
    pthread_cond_signal(&CV);
  pthread_mutex_unlock(&MU);
  return NULL;
}

//parent thread
int main(){
  pthread_t threadid;
  pthread_create(&threadid, NULL, worker, NULL);

  pthread_mutex_lock(&MU);
  GLOB = 1;
  pthread_mutex_unlock(&MU);

  pthread_mutex_lock(&MU);
    while (COND != 1){
       pthread_cond_wait(&CV, &MU);
    }
  pthread_mutex_unlock(&MU);
  GLOB = 3;
  pthread_join(threadid, NULL);
  return 0;
}
```

Figure 5: An example where false positives regarding`GLOB` are generated, although a happens-before relation exists. The reason is that at the time of last access to `GLOB`, `GLOB` is in the *Shared-Modified* state because the happens-before relation introduced by signal and wait is not considered.

It should be mentioned that race detectors use different techniques to implement the additional steps introduced above. VisualThreads uses thread segment graphs[11], while RaceTrack uses vector clocks[30]. Helgrind uses vector clocks in addition to thread segment graphs to reduce comparison overhead [28]. Another point is that VisualThreads supports only fork/join primitives whereas Helgrind and RaceTrack support some other threading primitives, e.g. condition variables.

Our extended technique avoids the shortcomings of the earlier detectors by combining the happens-before relation and the lockset analysis in an efficient manner.

4. THE EXTENDED TECHNIQUE

In this section, we propose a new hybrid solution that extends the memory state machine used by Eraser and similar tools. In the presented solution, we address the limitations observed in earlier tools by combining locksets and the happens-before relation in a new way. The basic idea is to apply the happens-before relation whenever the locksets indicate a possible race. An extended memory state machine depicted in the section 4.1 reduces false positives while limiting false negatives.

4.1 Memory State Machine

Figure 6 shows the extended memory state machine followed by description of each state. As the implementation is based on Helgrind, thread segment identifiers (TS_{ID}) are used to indicate a happens-before relation between two successive accesses to a memory location. Details about TS_{ID} are discussed in Section 3.2. A similar notation could be used for vector clock values instead of thread segment identifiers.
Following is the notation used in the state diagram:

- d: an arbitrary memory location.
- W: write operation to d.
- R: read operation from d.
- TS_{new}: thread segment identifier of the current access to d.
- TS_{old}: thread segment identifier of the prior access to d.
- $TS_{old} \mid\mid TS_{new}$: thread segments TS_{old} and TS_{new} are concurrent. Any access to d within TS_{old} is concurrent with accesses to d in TS_{new}.
- $TS_{old} \overset{hb}{\rightarrow} TS_{new}$: thread segments TS_{old} happens before TS_{new}. Any access to d within TS_{old} happens before accesses to d in TS_{new}.
- L_t: the current set of locks held by thread t.
- C_d: the current candidate set of locks protecting variable d.

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

Exclusive-Write: Location d is exclusively written by a particular thread. No locksets are needed. Record the thread ID. As long as write accesses occur from the same thread, stay in this state (the happens-before relation holds

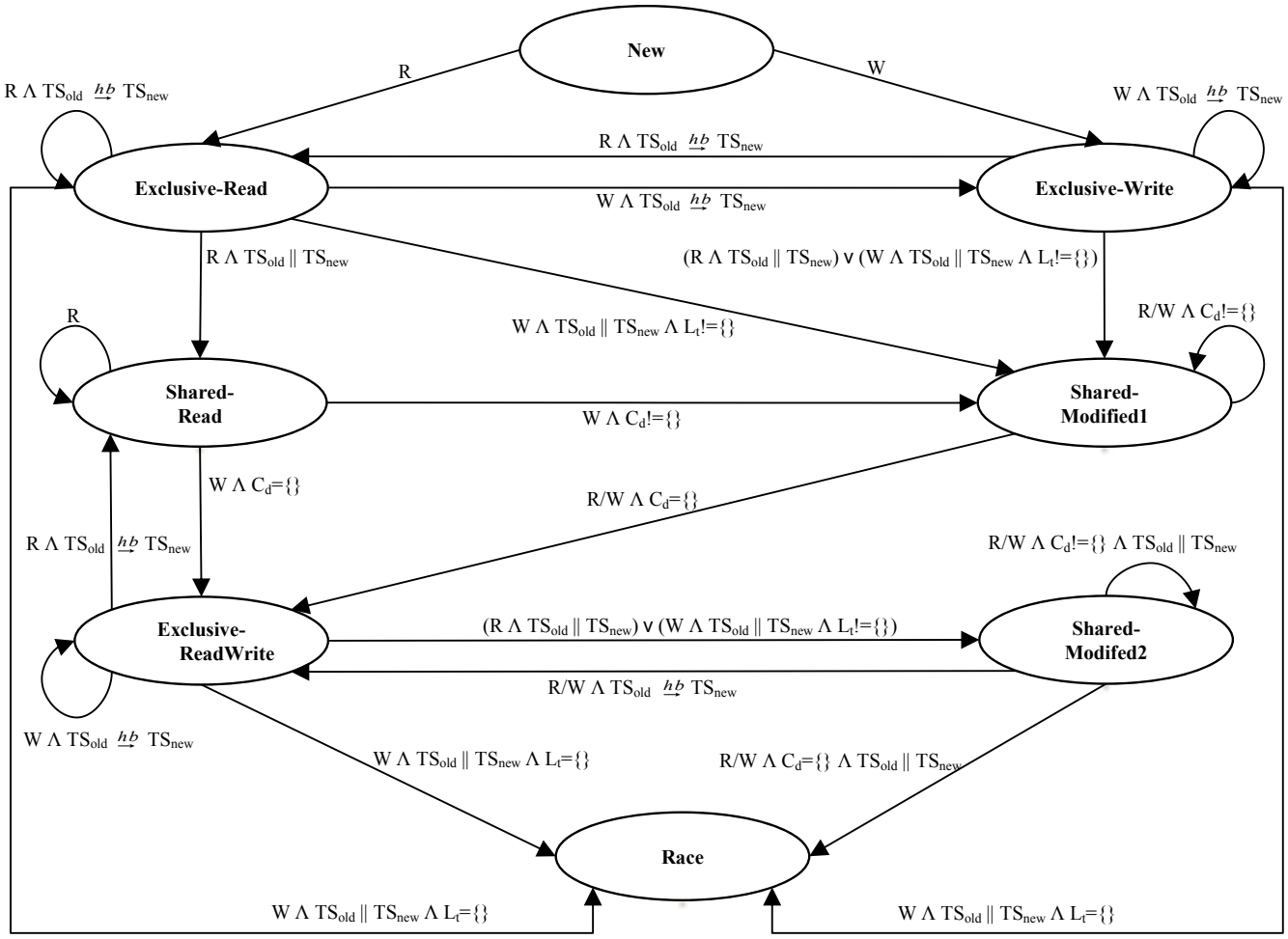

Figure 6: Extended memory state machine.

within the same thread, so there is no concurrent access). When a write access occurs from a different thread, switch to thread segments. Associate d with the thread segment of the new thread. Remain in this state as long as each successive write satisfies $TS_{old} \xrightarrow{hb} TS_{new}$, since there are no concurrent accesses to d. When a write or read occurs with $TS_{old} \mid\mid TS_{new}$, enter *Shared-Modified1* or *Exclusive-Read*.

Exclusive-Read: Location d is exclusively read by a particular thread. Similar to *Exclusive-Write*. When an access occurs that would result in $\mathrm{TS}_{old} \mid\mid TS_{new}$, enter *Shared-Read* or *Shared-Modified1* and discard information about the happens-before relation as it is no longer needed.

Shared-Read: Location d is concurrently accessed by multiple threads, but all accesses are reads. We enter this state from *Exclusive-Read* when a read results in multiple concurrent accesses. In this state, we track only the lockset C_d, which is initialized to L_t. The lockset is updated for every access. If C_d is empty and a write operation occurs, enter *Exclusive-ReadWrite*. If C_d is not empty and a write occurs, enter *Shared-Modified1*. No errors are reported in this state.

Shared-Modified1: Location d is read and written concurrently by multiple threads. This state is entered either from *Exclusive-Write* or *Exclusive-Read*, with C_d initialized to L_t. As in *Shared-Read*, track only the lockset in this state. If C_d becomes empty, enter *Exclusive-ReadWrite*.

Exclusive-ReadWrite: Location d is accessed by multiple threads and the lockset discipline alone is not sufficient. Track the thread segment identifier corresponding to the most recent access to d. Similar to *Exclusive-Read* or *Exclusive-Write*, remain in this state as long as there is a happens-before relation between successive accesses. When there is a write operation and $TS_{old} \mid\mid TS_{new}$, enter *Shared-Modified2*. When a read operation happens and there is a happens-before relation, return to *Shared-Read*.

Shared-Modified2: Location d is concurrently read and written by multiple threads. When entering this state, the lockset C_d is initialized to L_t. Both lockset and thread segments are tracked. If the lockset is empty and d is concurrently accessed, generate an error and enter state *Race*. This is the only state where both the happens-before relation and the lockset is analyzed. But whenever a happens-before relation exists between successive accesses, return to *Exclusive-*

ReadWrite. This speeds up the algorithm because it reduces the overhead for locksets, especially in long-running programs.

Race: A potential race is detected and reported. This state can be reached from *Shared-Modified* when $C_d = \{\}$ and $TS_{old} \mid\mid TS_{new}$, which means that d is concurrently accessed by multiple threads without a common lock. Also it is possible to reach *Race* from all exclusive states, in case a write happens concurrently with another access and $L_t = \{\}$. 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.

4.2 Main Features

The general idea is to avoid entering a shared state until the happens-before analysis shows that there is a concurrent access to a memory location. Lockset tracking is performed only in shared states, that is C_d is updated only in *Shared-Read*, *Shared-Modified1*, and *Shared-Modified2*. No happens-before test is done in the states *Shared-Read* and *Shared-Modified1*. Only state *Shared-Modified2* requires both lockset updates and happens-before analyses (see Figure 6).

Happens-before analysis is deferred until the lockset of a location is empty, leading to performance improvement. Updating both locksets and the thread segment graph for each access during program execution can be quite expensive in both time and space. The idea of deferring the computation of happens-before until necessary was introduced for the first time in [22]. This idea is implemented here by introducing the state *Exclusive-ReadWrite*. Entering the states *Exclusive-Read* or *Exclusive-Write* instead of *Exclusive-ReadWrite* would lead to false negatives in some situations.

Separate *Exclusive-Read* and *Exclusive-Write* states are beneficial for several reasons. As described previously, the Eraser algorithm is vulnerable to scheduling (see the example in Figure 3). By introducing these two exclusive states, the state machine can distinguish a read after a write and a write after a read, so the race is detected regardless of schedule, causing an immediate transition to *Race*. In addition, the edge from state *New* to *Exclusive-Read* makes the detector work more precisely and handle more cases properly. It is often possible for locations to change from *New* directly to *Exclusive-Read*, especially if the application is reading uninitialized variables, or due to compiler optimizations, where the compiler loads a word from memory, part of which is uninitialized, and then does not use the uninitialized part. Another case is when a program has its own memory management routines that initializes memory with zeros before allocation. In this situation, the memory will be *New* but a read from it is quite legal.

With the edges from *Exclusive-Write*, *Exclusive-Read* and *Exclusive-ReadWrite* to *Race* we capture races that happen only once at initialization time. An Eraser-style detector is based on the assumption that the program runs for a long time and if the race happens many times, it will be caught eventually. With the additional edges, the extended memory state machine can catch the race even if it happens at initialization time.

The distinction between *Shared-Modified1* and *Shared-Modified2* is mainly justified by performance reasons. In *Shared-Modified1* only the lockset is maintained, whereas in *Shared-Modified2* both lockset and thread segments are tracked. This optimization helps make the detector practical for real-world applications. Also, there is a difference in the way the empty lockset is handled. In *Shared-Modified1*, a transition to *Race* on an empty lockset would lead to numerous false positives. Thus,a single *Shared-Modified* state would increase the false alarm rate. On the other hand, if the alarm produced in the single *Shared-Modified* is a true positive, it could be caught immediately avoiding a possible false negative. Future experiments will compare both versions choosing the best compromise solution. Based on the current experimental results, the distinction between *Shared-Modified1* and *Shared-Modified2* is beneficial for long-running applications. If an empty lockset in $Shared - modifed1$ is indeed a true positive, it is mostly followed by another parallel read/write leading to the *Race* state.

Finally, the idea of introducing a separate state *Race* is useful, because once the race is reported, the tool does not spend time on this memory location any more.

5. IMPLEMENTATION OF HELGRIND$^+$

We extended the open source dynamic race detector Helgrind 3.3.0 [28] with the new detection technique. The new tool is called Helgrind$^+$. For the sake of brevity, only the main modifications are described.

5.1 Helgrind

Helgrind is a part of the Valgrind tool[17, 15, 16] for race detection in C/C++ and Fortran programs. It uses the Eraser algorithm[26] and the improvements based on the happens-before relation from VisualThreads[8] in order to reduce false positives. Valgrind is a binary instrumentation framework for Linux ELF Binaries and was initially used as a memory checker. It consists of two parts: A core that is responsible for generating intermediate code from an executable binary and interprets the code using a just-in-time compiler to speed up the execution time, and a plug-in that instruments the intermediate code before it is executed and interprets the results. This makes Valgrind a powerful and flexible tool for all kinds of runtime checking.

Helgrind is language independent and does not need source code. It supports the POSIX Threads API for multi-threading (OpenMP is supported as it ultimately uses POSIX threads in the GCC implementation). In order to suppress false warnings of a part of code that is not modifiable (e.g., libraries), one writes a suppression file that contains the report type and call stack patterns of specific false positives, so that such warnings do not appear in future reports.

5.2 State Information and Data Structures

Helgrind uses binary instrumentation in order to observe read and write instructions. Furthermore, a special version of the POSIX Threads library is provided that intercepts calls to thread-management, mutex-handling, etc. This is necessary to maintain thread state information and locksets. The algorithm is unaware of the high level objects or structures in the source code. For every memory location Helgrind maintains a so-called shadow word that contains the state information associated with the memory location. The shadow word used by Helgrind is a 32-bit word. Two bits are used to encode states and the rest to store thread seg-

ment identification (only in Exclusive state) and lockset (in shared states) information. For our extended memory state machine, a 32-bit word for shadow values is not sufficient. There are eight different states and in state *Shared-Modified2* both lockset and thread segment information must be stored. Therefore, Helgrind$^+$ uses a 64-bit shadow word for each memory location. Figure 7 demonstrates the structure of the 64-bit shadow words in different states.

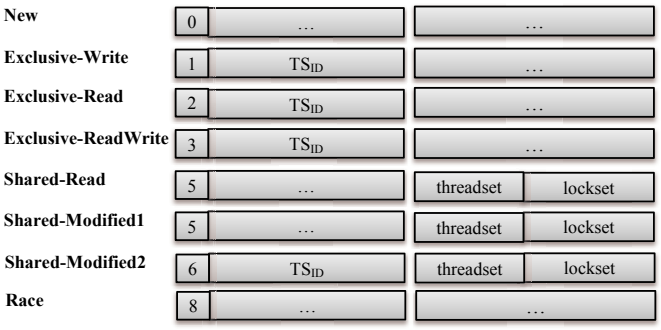

Figure 7: Structure of 64-bit word shadow value and state encoding

The first three most significant bits of the 64-bit shadow value are used to encode the state (0 - 7). The interpretation of the other bits depends on the state. Three bytes are used to store the thread segment identifiers in exclusive states and *Shared-Modified2*. In the exclusive states the second 32-bit word is unused. Lockset information is stored in the four least significant bytes (second 32-bit word) of shared states. The states *New* and *Race* use only three bits to store encoded state. If a memory location is in an exclusive state, the thread segment ID of the last access is stored. The lockset is not initialized unless one of the shared states is reached. Moreover, structures that require dynamic allocation during runtime are provided from a fixed heap to avoid dynamic memory allocation. These structures include condition variable information, mutex information and thread segments. Reference counting is used to determine which segments are free and a simple garbage collection algorithm returns unused segments back to the heap.

Shadow values in Helgrind$^+$ double the space overhead compared to Helgrind. But there are several optimizations that could reduce the memory overhead in Helgrind$^+$, as all 64 bits are needed in only one state.

6. EXPERIMENTS AND RESULTS

In this section, we present our experiences with Helgrind$^+$ and evaluate our approach by applying it to a number of applications and benchmarks. We show that the extended approach significantly reduces the amount of false positives at a reasonable overhead, making Helgrind$^+$ more usable by giving more precise reports. We have used different application from a variety of domains, from small programs to long-running real applications. We focus on false positives.

6.1 False Positives

The first test of Helgrind$^+$ was the analysis of the multithreaded programs collected from graduate students in a lab course at our department during the past two years. The programs are lab assignments done by different teams of students, using POSIX Threads or OpenMP for parallelization. We executed the programs with differing number of threads and chose those programs producing warnings with either tool. The warnings were analyzed to see if they were real races or false positives. Table 1 lists the analyzed programs. The first entry is a parallel implementation of the Single Source Shortest Path algorithm (SSSP)[21]. Another example is bzip2 in several versions (produced by different student teams), a sequential data compressor that uses the Burrows-Wheeler block-sorting text compression algorithm and Huffman coding[27]. For comparison purposes, the number of false positives produced by pbzib2 [10], the parallel version of bzib2 available on Linux Ubuntu 7.10 is also provided. The remaining programs are listed in Table 1. We also used two well-known bug-free SPLASH-2[1] benchmarks FFT and LU. SPLASH-2 has been used in many previous studies[22, 13, 24, 24, 23] to evaluate race detection.We applied Helgrind 3.3.0 and Helgrind$^+$ to the programs. All experiments shown in Table 1 were conducted on a dual core machine with x86 32bit processors running Linux. The programs were compiled with gcc 4.2.1.

Comparing the results in Table 1, false positives are drastically reduced. Also, when increasing the number of threads, the number of false positives tends to increase for Helgrind, while it stays constant for Helgrind$^+$. All known, real races were detected by both tools.

We also applied Helgrind$^+$ to the recently released PARSEC benchmark[2]. PARSEC differs from other benchmark suites, as it is not HPC-focused. It contains twelve divers programs from different areas such as computer vision, video encoding, financial analytics, animation, physics and image processing. Table 2 provides a short summary of the programs in the benchmark.

Program	Application Domain	Parallelization Model	Granularity
blackscholes	Financial Analysis	data-parallel	coarse
bodytrack	Computer Vision	data-parallel	medium
canneal	Engineering	unstructured	fine
dedup	Enterprise Storage	pipeline	medium
facesim	Animation	data-parallel	coarse
ferret	Similarity Search	pipeline	medium
fluidanimate	Animation	data-parallel	fine
freqmine	Data Mining	data-parallel	medium
streamcluster	Data Mining	data-parallel	medium
swaptions	Financial Analysis	data-parallel	coarse
vips	Media Processing	data-parallel	coarse
x264	Media Processing	pipeline	coarse

Table 2: Summary of PARSEC benchmarks.

Both Helgrind and Helgrind$^+$ were applied to all PARSEC programs. Table 3 displays the result of the experi-

Program	Lines	Threads	False Positives		Real Races
			Helgrind	Helgrind$^+$	
SSSP	414	2	19	1	0
SSSP	414	8	16	1	0
p-bzip2 version A	8515	2	7	2	0
p-bzip2 version A	8515	8	9	2	0
p-bzip2 version B	9270	2	3	2	0
p-bzip2 version B	9270	8	8	2	0
p-bzip2 version C	7028	2	2	1	0
p-bzip2 version C	7028	8	2	1	0
p-bzip2 version D	8570	2	0	0	0
p-bzip2 version D	8570	8	0	0	0
pbzib2 Linux	2355	2	2	0	0
pbzib2 Linux	2355	8	2	0	0
Histogram	194	2	0	0	0
Histogram	194	8	2	0	0
Mergesort	525	2	2	2	2
Mergesort	525	8	6	2	2
FFT	1086	2	2050	0	0
FFT	1086	4	3074	0	0
LU	1100	2	366	0	0
LU	1100	4	682	0	0

Table 1: False positive warnings on the selected student programs and the bug-free SPLASH-2 benchmarks. Helgrind$^+$ denotes the modified version of Helgrind.

ments run with two threads per application. All programs were compiled with gcc 4.2.1 without any modification and executed on a machine with 2x Intel XEON E5320 Quadcore at 1.86GHz, 8 GB RAM, Ubuntu 7.10 x64. Because of the large computational cost, we did not perform simulations with the native input set. Instead, we used the `simsmall` 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. All programs used the standard Pthread library for parallelization, except for `freqmine`, which uses OpenMP.

The results appear in table Table 3. Helgrind$^+$ reports no false positives for seven out of twelve programs. For `blackscholes`, which uses only two barriers, no alarm is reported, whereas the original Helgrind reports two false positives, since it does not support barriers. Four of the programs use only locks for synchronization. For two of them, `canneal` and `fluidanimate`, Helgrind$^+$ produces no false alarms. `Swaptions` uses 78 locks, but neither tool reports false positives.

`Freqmine` uses only locks, but the difference with others is that it is implemented in OpenMP. As Helgrind is primarily built as a thread error detector POSIX threads, there are some technical problems when using OpenMP. On Linux, the GNU OpenMP implementation (libgomp.so) constructs its own synchronization primitives and contains its own custom barrier primitive. Because of the way these synchronization primitives are constructed, Helgrind cannot see all the relevant inter-thread dependencies. Thus, the number of false positives for programs using GNU OpenMP is quite high. This limitation is going to be removed in the next versions. Nevertheless, the difference in false alarms is remarkable.

`Streamcluster` and `Bodytrack` use all three kind of synchronization primitives. `Streamcluster` uses a high fraction of barriers compared to the others. Helgrind produces many warnings for this application mostly because it does not support barriers. Helgrind$^+$ produced no warnings for `Dedup`, whereas Helgrind produced thousands. Both tools produce false positives for the remaining applications, but the number of false positives for Helgrind$^+$ is significantly lower.

Condition variables produce many false positives, because they are not easy to handle properly. In many situations it is not easy to detect the happens-before relation introduced by them. Thus, race detectors do not have enough information about inter-thread dependency, causing false alarms. A particularly difficult situation is when a thread sends a signal, but no thread waits for it and the signal is lost. Figure 8 illustrates such a scenario. A similar problem occurs when there are spurious signals, i.e. the signaling thread sends several uncaught signals.

6.2 Overhead

Due to 64-bit shadow values, Helgrind$^+$ uses double the auxiliary memory. Nevertheless, there are many opportunities for optimization. It is also possible to reduce the execution time and memory overhead by reducing the amount of instrumentation.

We measured the execution time of instrumented code versus the actual execution time. Applications typically slow down by a factor of 10 to 50 while using Helgrind. The overhead of Helgrind$^+$ over Helgrind on the PARSEC Benchmark is shown in Figure 9. All measurement are average values of five executions with two threads on a machine with 2x Intel XEON E5320 Quadcore at 1.86GHz, 8 GB RAM,

Program	Lines	Instructions (Billion)		Synchronization Primitives			False Positives	
		Reads	Writes	Locks	Barriers	Conditions	Helgrind	Helgrind$^+$
blackscholes	812	0.092	0.045	0	2	0	2	0
bodytrack	10,279	0.425	0.102	35,849	215	90	2,236,552	1,400,951
canneal	4,029	0.435	0.187	88	0	0	1	0
dedup	3,689	0.658	0.254	18,436	0	3,536	7,752	0
facesim	29,310	9.632	4.191	10,460	0	1,795	341,483	3,390
ferret	9,735	0.005	0.002	6,660	0	10	0	0
fluidanimate	1,391	0.584	0.144	923,750	0	0	15,658	0
freqmine	2,706	0.744	0.283	78	0	0	1,006,210	1,429
streamcluster	1,255	0.428	0.009	266	22,784	74	11,779	0
swaptions	1,494	0.349	0.091	78	0	0	0	0
vips	3,228	0.758	0.199	10,575	0	2,698	614,761	542,642
x264	40,393	0.500	0.204	1,339	0	157	6,145	352

Table 3: Number of warnings as possible data races reported on PARSEC benchmarks. Programs are executed for input set `simsmall` with two threads on an 8-core x86-machine. Numbers for read/write instructions and synchronization primitives are totals across all threads.

Ubuntu 7.10 x64, with the `simsmall` inputs for all simulations. As the figure shows, the time differences between the 32-bit and 64-bit versions are modest. In the average case, the slowdown is about 10%. In the worst case the overhead measured is 32%. The overhead is insignificant compared to the time a user would have to spend checking the extra false positives produced by Helgrind.

7. CONCLUSION AND FUTURE WORK

We have shown that by carefully combining the techniques of lockset-based and happens-before-based detection, a race detector with better accuracy and reasonable overhead can be obtained. Our experimental results suggest that the new combined technique can be applied to a wide variety of applications, resulting in fewer false positives and more accurate warnings, while avoiding false negatives. Execution time and space overhead is moderate enough for practical use in large applications.

Future work includes experiments with additional real-world applications and some optimizations that further improve performance and accuracy. Additional work is needed in automatically detecting and handling synchronization patterns without relying on source code annotations. This work could lead to further reductions in false positives. Techniques for grouping warnings by fault location and classifying races based on the states in which they were detected could be helpful for identifying harmful races. Moreover, applying runtime analysis to exclude variables which are accessed by a single thread only could lead to better performance and fewer warnings. Another direction for future work is applying static analysis to reduce the amount of instrumentation required.

Acknowledgments

We would like to thank Valgrind author Julian Seward for answering our questions regarding the Helgrind codebase.

8. REFERENCES

[1] *The SPLASH-2 Programs: Characterization and Methodological Considerations*, Santa Margherita Ligure, Italy, 1995.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical report, January 2008.

[3] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[4] J.-D. Choi and S. L. Min. Race frontier: reproducing data races in parallel-program debugging. *SIGPLAN Not.*, 26(7):145–154, 1991.

[5] M. Christiaens and K. D. Bosschere. Trade, a topological approach to on-the-fly race detection in java programs. In *JVM'01: Proceedings of the JavaTM Virtual Machine Research and Technology Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 15–15, Berkeley, CA, USA, 2001. USENIX Association.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[7] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. *SIGPLAN Not.*, 26(12):85–96, 1991.

[8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, page 181, New York, NY, USA, 2001. ACM Press.

[9] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003.

[10] J. Gilchrist. Parallel bzip2 (pbzip2):data compression software, 2007.

[11] 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*, pages 331–342, London, UK, 2000. Springer-Verlag.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*,

```
int COND = 0;
int GLOB = 0;
static pthread_cond_t  CV = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t MU = PTHREAD_MUTEX_INITIALIZER;

//worker thread
void *worker(void*){
  GLOB = 1;
  pthread_mutex_lock(&MU);
    COND = 1;
    GLOB = 2;
    pthread_cond_signal(&CV);
  pthread_mutex_unlock(&MU);
  return NULL;
}

//parent thread
int main(){
  pthread_t threadid;
  pthread_create(&threadid, NULL, worker, NULL);
  usleep(1000); //lost signal
  pthread_mutex_lock(&MU);
    while (COND != 1){
       pthread_cond_wait(&CV, &MU);
    }
  pthread_mutex_unlock(&MU);
  GLOB = 3;
  pthread_join(threadid, NULL);
  return 0;
}
```

Figure 8: An example that causes a false positive on `GLOB`, even though a happens-before relation exists. The signal sent by the worker thread is lost and so the race detector cannot establish the happens-before relation. However, there is still an inter-thread dependency, but through the `COND` variable.

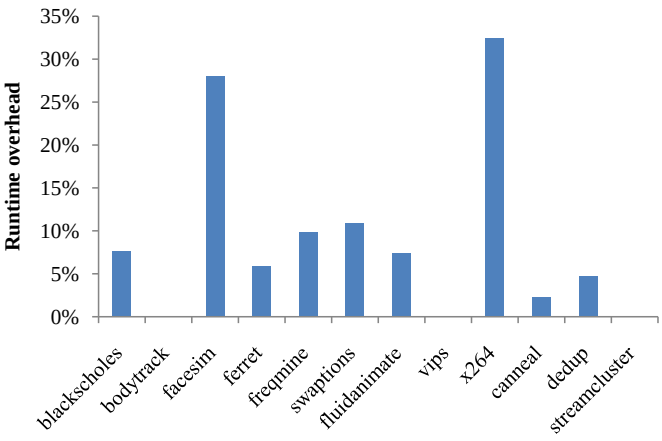

Figure 9: Runtime overhead of modified race checker on PARSEC benchmarks.

21(7):558–565, 1978.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access-interleaving invariants. *IEEE Micro*, 27(1):26–35, 2007.

[14] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*.

[15] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, UK, 2004.

[16] N. Nethercote and J. Seward. Valgrind: A program supervision framework. 2003.

[17] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[18] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[19] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[20] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, 2003.

[21] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 53–60, New York, NY, USA, 2008. ACM.

[22] E. Pozniansky and A. Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, 2007.

[23] B. Richards and J. R. Larus. Protocol-based data-race detection. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 40–47, New York, NY, USA, 1998. ACM.

[24] M. Ronsse and K. D. Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[25] 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*, pages 34–41, New York, NY, USA, 2006. ACM.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[27] J. Seward. bzip2: A free data compressor, 2007.

[28] Valgrind-project. Helgrind: a data-race detector, 2007.

[29] C. von Praun and T. R. Gross. Object race detection. *SIGPLAN Not.*, 36(11):70–82, 2001.

[30] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.