Identifying Ad-hoc Synchronization for Enhanced Race Detection

Ali Jannesari and Walter F. Tichy
Karlsruhe Institute of Technology (KIT)
76131 Karlsruhe, Germany
Email: {jannesari, tichy}@kit.edu

***Abstract*—Parallel programs contain a surprising number of ad-hoc synchronization operations. Ad-hoc synchronization operations are loops that busy-wait on condition variables. Current race detectors produce unnecessary warnings (false positives) when ad-hoc synchronization is used. False positives are also generated when programmers use synchronization primitives that are unknown to race detectors, for instance when programmers switch libraries. These shortcomings may result in an overwhelming number of false positives, dissuading programmers from using race detectors.**

This paper shows that ad-hoc synchronization operations can be detected automatically. The method requires no user intervention such as annotations and has been implemented in the race detector Helgrind$^+$. Evaluation results on various benchmarks confirm that Helgrind$^+$ is aware of all synchronizations in programs, reliably reports true races, and produces few false alarms. A surprising result is that with the new technique, Helgrind$^+$ can analyze synchronization libraries, so special knowledge about these libraries is not needed in the detector.

***Keywords*-data race detection, race conditions, debugging, parallel programs, ad-hoc synchronization, synchronization primitives, dynamic analysis.**

I. Introduction

Programmers tend to implement their own synchronization primitives when available synchronization constructs are too slow. For instance, a programmer may write a spinning loop instead of using a library-supplied wait-operation, if the loop is entered only rarely. Furthermore, libraries may lack certain higher-level synchronization constructs such as barriers or task queues, forcing programmers to implement their own. We call synchronization constructs implemented in application programs *user-defined* or *ad-hoc*.

Ad-hoc synchronization operations occur surprisingly frequently. For instance, we found that eight of the 13 PARSEC benchmarks[1] contain between 32 and 329 ad-hoc synchronizations. For race detectors, ad-hoc synchronization presents a problem, in that race detectors are not aware of these constructs and thus generate an avalanche of false positives for them. For instance, Tian et al [2] observe an average of four million false positives generated for programs containing from 12 to 131 ad-hoc synchronization segments.

The subject of this paper is the reliable detection and correct treatment of ad-hoc synchronization in race detectors, with the aim of eliminating false warnings.

Krena et al[3] and Tian et al[2] identified spin loops that check conditions as the basic pattern in user-defined synchronization. Tian et al use a simple heuristic to identify these spin loops at runtime and suppress warnings associated with them. In essence, their approach treats any loop with a control variable that does not change for three iterations as a spin loop waiting for a signal. However, this approach may still generate false positives if the spin loop is not executed repeatedly. Recall that programmers expect their programs to enter spin loops only rarely.

We provide a dynamic method that detects ad-hoc synchronization constructs reliably, provided they use spin loops that examine condition variables. The method dynamically and automatically identifies these loops by analyzing the object code. The signaling thread cannot be determined through analysis, but it can be found dynamically by instrumenting the code. Our method detects both reads and writes on condition variables and then establishes a happens-before relation between signaling and signaled threads, thus preventing the generation of false warnings. The method has been added to the race detector Helgrind$^+$[4], [5]. The results on substantial benchmark suits confirm that Helgrind$^+$ eliminates false warnings without missing true races.

A side benefit of this approach is that it can also be applied to unknown libraries. Helgrind$^+$ currently uses information about the synchronization constructs of PThreads, but if application programmers use different libraries, then our enhanced Helgrind$^+$ can also detect races reliably, provided the libraries are based on spin loops. Note that even operating system calls such as *wait* that relinquish the processor are typically used inside loops and therefore detectable by Helgrind$^+$. A surprising result is that information about PThreads can be removed entirely from Helgrind$^+$, resulting in only a minor increase in false positives. Thus, Helgrind$^+$ with spin loop detection can be seen as a *universal race detector*.

The structure of the paper is as follows. Section II describes the basics. We deal with the problems of using synchronization operations by presenting various examples and distinguishing true races from false races in Section II-A. Our general method for identifying ad-hoc synchronization and unknown synchronization primitives is presented in Section III. The algorithm to detect the common construct of ad-

978-1-4244-6443-2/10/$26.00

hoc synchronization is discussed in Section III-B. In Section IV, we evaluate our method with respect to accuracy and performance. We describe our experiences with Helgrind$^+$. Related work is discussed in Section V.

II. Synchronization Operations

A data race occurs when at least two threads access the same memory location without any ordering constraints enforced by synchronization operations between accesses, and at least one of the accesses is a write [6]. Dynamic data race detectors typically intercept calls to synchronization primitives in order to find ordering constraints. For instance, if a program calls the synchronization primitive *barrier_wait*, the detector intercepts this function call and records that accesses after the barrier are not concurrent with accesses before the barrier. For a race detector to identify all races and to produce no false positives, it must be aware of ordering effects of all synchronization constructs, including locks, monitors, signal/wait, conditions variables, spurious wakeup calls, barriers, etc. Our previous results in [4], [5] confirmed that by tailoring the detection algorithm for each synchronization primitive, the detector extracts highly accurate ordering information, identifies all races, and keeps the number of false positives low.

But what if synchronization primitives are not supported by the detector? Then the usual detectors are not able to intercept them, know nothing about ordering effects, and therefore may produce numerous false positives. The number of false positives can be so high as to overwhelm the programmer.

The idea of this paper is to identify a basic pattern that occurs in virtually all synchronization primitives and to extend the detection algorithm to handle this pattern. This pattern is the spinning read loop waiting for a condition to change. Once this pattern is handled well, we can in fact remove all code from the race detector dealing with synchronization primitives built upon this loop. Moreover, all synchronization libraries as well as ad-hoc synchronization based on the spinning read loop will be handled automatically, eliminating the need to enhance the detector for every library.

A. True and False Races

Generally, false races can be classified as follows:

- *apparent races*, and
- *synchronization races*

If a detector is not aware of a synchronization construct, it may report races where they are none, because they are actually prevented by the construct. Such cases are called *apparent* races. Since detectors may not support all synchronization primitives, apparent races cause false positives. Figure 1 depicts a simple example that uses the synchronization primitive *barrier_wait(b)*. Assuming the primitive is unknown to the detector, it will report races regarding the variable `DATA`. The detector will consider all read operations after the barrier as races, although there is no concurrent write. If the synchronization primitive were known to the detector, the false races would disappear.

```
Thread 1, 2, ... n:

    lock(l)
      DATA++
    unlock(l)

    barrier_wait(b)

    print DATA
```

Figure 1. Using synchronization primitive *barrier_wait()* from unsupported library causes apparent races on `DATA`.

Figure 2 depicts a simple ad-hoc synchronization in which the second thread waits for condition variable `FLAG`. An uninformed detector would report an apparent race on variable `DATA`.

```
/* Initially FLAG is 0 */

                ...

DATA = 1                    while(FLAG != 1)
                              /* do_nothing */
FLAG = 1
                            print DATA
```

(a) Thread 1 (b) Thread 2

Figure 2. Ad-hoc synchronization causes apparent race on `DATA` and synchronization race on `FLAG`.

The second major reason for false positives are *synchronization races*. Consider `FLAG` in Figure 2. Its accesses are unordered and constitute a true race. However, this race is harmless and, in fact, intentional. The race is necessary for proper synchronization. Intentional races are often known as *synchronization races* [3], [2].

The example in Figure 1 also produces synchronization races. To see why consider the typical implementation of the barrier primitive in Figure 3. The intentional races on variable `counter` are synchronization races and harmless. Synchronization primitives require data races to enable competition for entering critical sections, for locking, for changing condition variables, etc.

```
lock(l)
  counter++
unlock(l)

while(counter != NUMBER_THREADS)
  /* do_nothing */
```

Figure 3. Implementation of synchronization primitive *barrier_wait()* causes synchronization races on `counter`.

Our aim is to refine Helgrind$^+$ in such a way that it does not report apparent or synchronization races, while reporting all other races.

III. Ad-hoc Synchronizations

A good race detector should avoid false positives associated with ad-hoc synchronization and synchronization races. In this section, we propose a dynamic detection method that is based on the fact that the *spinning read loop* is the common pattern of almost all synchronization constructs and a major source of synchronization races. The method identifies spin-loop synchronization correctly even if the spinning read is actually not entered (recall that programmers assume spinning loops are entered rarely). We first discuss the underlying pattern and then present our detection algorithm. The algorithm identifies all spin-loop synchronization operations, including those in libraries.

A. Common Pattern in Ad-hoc Synchronization

The so called *spin-lock synchronization* is the most common and simplest synchronization construct [3]. It employs a *condition* shared among two threads. One thread executes a while loop waiting for the value of the condition to be changed by the other thread. At the moment the value changes, the waiting thread is able to leave the loop and proceed. Figure 4 illustrates the use of the spin-lock. Thread 2 executes a spinning read loop on the shared variable `CONDITION`, until Thread 1 changes the value. The read and write operations are the source of a harmless synchronization race that need not be reported to the user.

```
do_before(X)

set CONDITION to TRUE
```

(a) Thread 1

```
while(!CONDITION){
  /* do_nothing */
}

do_after(X)
```

(b) Thread 2

Figure 4. Spinning read loop pattern. Happens-before relation induced by spin-lock synchronization.

A number of publications analyzed different implementations of synchronization operations [7], [8], [9], [3], [2] and observed that the *spinning read loop* is a common pattern used for implementing synchronization constructs. For example, the barrier implementation in Figure 3 also uses a spinning read loop on a flag and a write ending the spin.

B. Detecting Ad-hoc Synchronizations

In the previous section, we found that the spinning read loop and its counterpart write are the common construct for ad-hoc synchronizations. Helgrind$^+$ identifies spinning read loops just before runtime with binary instrumentation. Later on, runtime analysis establishes the correct happens-before relations between spinning read loops and counterpart writes so that the detector is aware of synchronizations.

The general idea of our method is as follows. Helgrind$^+$ searches the binary code just before execution to find all loops. This is done by building a control flow graph at pre-runtime. Next, it narrows the set to spinning read loops based on the following criteria:

- Evaluating the loop condition involves at least one load instruction from memory.
- The value of the loop condition is not changed inside the loop body.

We instrument each spinning read loop and mark the variables that affect the value of the loop condition (this may be a single flag or several variables, if the condition is an expression). However, at this point we do not know where the counterpart write is. Next, we discuss the instrumentation that finds the write.

Being a runtime race detector, Helgrind$^+$ monitors all read/write accesses. The state of a variable indicates whether it is used by only one thread (state *exclusive*) or several (state *shared*). When entering a spinning loop, the states of the variables affecting the loop condition are set to a special state called *spin*. Two cases are possible. In the first case, the counterpart write does not happen before entering the spin loop. In this case, Helgrind$^+$ waits for the first write operation that affects the loop condition. If the write operation is performed by another thread rather than the spinning thread, then this is the counterpart write. When leaving the loop, Helgrind$^+$ records a happens-before edge between the spinning read loop and its counterpart write.

Consider the example in Figure 4. `CONDITION` is the condition variable in the spinning read loop. Assume the spinning read is entered by Thread 2 before the counterpart write. The happens-before relation is constructed based on the data dependency (*write/read* dependency) on `CONDITION`. Thus, the detector will be aware of this synchronization operation and no race will be reported on `X`. The warning regarding the benign synchronization race on `CONDITION` is also suppressed, since it is marked as being in the special state *spin*.

In the second case the counterpart write happens first: One or more variables affecting the loop condition are written before the loop is entered. Helgrind$^+$ sets the states of the changed variables to *exclusive* and records the location of the write instructions. As soon as the instrumented spinning read loop is entered, the detector notices that variables affecting the loop condition have been changed. The loop itself terminates immediately. Helgrind$^+$ records the happens-before relation as before and sets the states of the changed variables to *spin*. In this case no actual spinning happens–the loop condition is evaluated only once.

The method by [2] fails to establish the happens-before relation in the second case, because this method relies on the loop spinning several times. It also fails to recognize

inter-thread event notifications with *signal()/wait()* in case of lost signals. A signal is lost if a thread sends a signal before any thread is waiting for it. In this case, no spinning read happens at runtime. The *wait()* primitive is not even executed, since the condition variable is already set earlier by the signaling thread and the loop terminates immediately. Failing to take signaling into account may lead to false positives. Another tricky case involves spurious wakeups. These can lead to false negatives (missed races). For proper handling of spurious wakeups, see [5]. Helgrind$^+$ handles all of these cases correctly.

Another problem concerns the heuristic of using a threshold iteration count in order to distinguish spinning read loops from ordinary loops. If the spinning read loop dose not spin long enough to reach the threshold value, the detector misses the spinning read loop and generates false positives. On the other hand, if the threshold value is too low, ordinary loops in the program could be mistaken for spinning read loops, which also results in missed races. Thus, without exploiting the semantic information by dynamic code analysis just before runtime, one may easily miss synchronizations or misinterpret them, since actual spinning reads may not happen at all at runtime or might not reach a preset threshold value.

C. The Algorithm

Conceptually, our method is divided into two phases: instrumentation and runtime. In the instrumentation phase, all loops in the program are recognized and then only the spinning read loops are selected to be instrumented. During the second phase, a runtime data dependency analysis is carried out to construct the happens-before relation between related parts.

Recognizing the loops in the program is performed by means of control flow analysis. We construct a control flow graph on the fly based on the current super block and consider loops with three to seven basic blocks in the graph. We check whether they are spinning read loops or not. In our experiments, we found three to seven basic blocks deliver good results, since the spinning read loops are typically small loops with few instructions. Decreasing this number may result in missing some spinning read loops and producing some false positives. On the other hand, increasing the number of basic blocks causes additional overhead.

Figure 5 provides a high level description of the algorithm for spinning read loop detection. The first step constructs the data dependency table D_l for every loop l. $D_l(condition_l)$ returns all variables that the loop condition $condition_l$ depends on within the loop l. This analysis takes function calls into account. Step 2 examines for all variables v that the $condition_l$ depends on, if v is modified inside the loop. If there is an assignment to any such v, then the loop l is not a spinning read loop. Otherwise, the loop is marked as performing spinning reads only, and the variables of $D_l(condition_l)$ are prepared for instrumentation.

for every loop l:

1) $D_l(condition_l)$: the set of variables, on which the condition $condition_l$ of the loop l depends
2) $\forall\ v \in D_l(condition_l)$:
   if $(D_l(v) \neq \emptyset)$ // v is modified
   return;
3) mark l as spinning read loop
   prepare all $v \in D_l(condition_l)$ for instrumentation

Figure 5. Basic algorithm for detecting spinning read loops.

D. Detecting Unknown Synchronization Primitives

The above method is a general approach that is able to detect synchronization operations executed in the program, which are implemented ultimately by spinning read loops e.g. locks, barriers, etc.

Furthermore, all unknown synchronization primitives that are not supported by our detector will be identified, i.e. synchronization primitives provided by any other library rather than PThreads. In other words, our general approach based on spinning reads detection results in a universal race detector that is aware of all synchronization operations (any library) in the program by identifying them as low level synchronizations. Thus, we overcome the serious limitation of prior works which makes detectors limited to only synchronization primitives of a particular library.

In addition, a race detector could be based only on this general approach to detect synchronization operations based on spinning read loops in a program. Such a race detector is a pure happens-before detector. It cannot make use of lockset algorithm, because it is not aware of locks. In our case, if we turn off the support of Pthreads so that synchronization primitives of Pthreads are not directly intercepted, we will get a pure happens-before detector. Our empirical results show that relying only on this general approach for identifying synchronization operations in the program might become too conservative in some situations. There may be obscure implementation of spinning read loops that are difficult to detect, leading to false positives. We used this approach as complementary method to our hybrid race detection algorithm [4], [5] to identify ad-hoc synchronizations together with synchronization primitives of unsupported libraries to achieve best results.

It should be mentioned, that our method is based on dynamic binary instrumentation. It does not need any programs source code or user interference such as source code annotations and therefore is non-intrusive in the build process. The whole code and semantic analysis are done automatically during just-in-time binary instrumentation.

IV. EXPERIMENTS

In this section, we present the results and evaluate our approach by applying it to a number of benchmarks. We show that by implementing the new method in Helgrind$^+$, we are able to report true races and improve the accuracy by eliminating synchronization races and false alarms. We evaluate the overhead caused by the method. The overhead is reasonable.

A. Experimental Setup

We implement the presented approach into our race detector Helgrind$^+$. Helgrind$^+$ is built on top of a dynamic binary instrumentation tool called *Valgrind* [10], [11], [12]. Valgrind is a *disassemble* and *resynthesize* dynamic just-in-time instrumentation framework. The framework translates binary code into a platform independent *Intermediate Representation* (IR). Helgrind$^+$ instruments the IR and hands it back to the framework which resynthesizes machine code from the instrumented IR.

Loops are converted to conditional branches at low level code. Hence, for the implementation of our algorithm, we consider all conditional branches in the IR code. We search the control flow graph for loops that span a maximum number of three to seven basic blocks. Then, we track the dependencies of each variable within these basic blocks by constructing a data dependency table. The data dependency table is built up with respect to registers at the IR level. All temporaries and addresses in basic blocks are traced to identify the registers they depend on. By means of the dependency table we can now check if the loop condition variable depends on a register that is target of a *load instruction*. We instrument the spinning read loop and insert the required instructions to intercept and analyze it at runtime, if the load addresses stay constant. Helgrind$^+$ is also able to intercept calls to library functions. It instruments direct calls to library functions of PThreads (POSIX Threads) Library for runtime checking.

We also used shadow memory [13] for each memory location to maintain the information needed during runtime, namely state information and vector clocks. A special state called *spin* in shadow memory indicates variables that are used in spinning read loops.

All our 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 8.10.2 x64. All programs were compiled with gcc 4.2.3. No source code annotation was used. We employed Helgrind$^+$ with the new features based on a 64 bit version of Valgrind 3.4.1 for our experiments and measurements.

B. Results

We applied Helgrind$^+$ to programs provided in *data-race-test* [14], a test suite for race detectors. It provides more than 150 short programs (test cases) that implement different scenarios which could occur while running multi-threaded programs. The scenarios represent tricky situations, which are difficult to discover by race detectors. Currently, 120 of these test cases can be classified 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 I shows the result of our experiment on the test suite. All test cases are short programs implemented in C/C++ using PThreads with a varying number of threads and executed without any annotation.

Tools	False Positives	False Negatives	FP+FN	Passed Cases
Helgrind$^+$ lib	32	8	40	80
Helgrind$^+$ lib+spin(7)	8	7	15	105
Helgrind$^+$ nolib+spin(7)	9	7	16	104
DRD	13	20	33	87
Helgrind$^+$ lib+spin(3)	24	7	31	89
Helgrind$^+$ lib+spin(6)	23	7	30	90
Helgrind$^+$ lib+spin(7)	8	7	15	105
Helgrind$^+$ lib+spin(8)	8	7	15	105

Table I
RESULTS OF HELGRIND$^+$ ON THE TEST SUITE *data-race-test* THAT CONTAINS 120 PROGRAMS AS UNIT TEST CASES. FP AND FN DENOTE FALSE POSITIVES AND FALSE NEGATIVES, RESPECTIVELY. THE OPTION *lib* MEANS INTERCEPTION OF PTHREAD LIBRARY AND *spin* STANDS FOR SPINNING READ DETECTION WITH THE NUMBER OF BASIC BLOCKS AS A PARAMETER.

The basic version of Helgrind$^+$ denoted by *lib* option in Table I failed on 40 test cases out of 120. The option *lib* denotes that Helgrind$^+$ intercepts Pthread synchronization primitives calls. It produces 32 false positives and eight false negatives. By enabling the new option *spin(7)* for detecting spinning read loops up to seven basic blocks and identifying ad-hoc synchronizations, 24 false positives and one false negative are removed, i.e., 105 test cases out of 120 pass. The removed false positives are all apparent races or synchronization races that arise from using ad-hoc synchronization. The removed false negative was because of spurious wake ups when using same condition variable between several threads. We consider spinning read loops up to maximum seven basis blocks. All synchronization primitives in programs are used from PThreads which Helgrind$^+$ intercepts directly.

A few failed test cases used ad-hoc synchronization. However, it is not easy to detect them, as they do not follow the standard pattern in ad-hoc synchronization and use a function pointer call to evaluate the loop condition in spinning read loops. This is a limitation in the implementation of our algorithm. We aim to remove this limitation in future

work to further decrease the number of false positives.

If we switch off the support of PThreads library indicated by option *nolib*, synchronization primitives are no longer intercepted directly and therefore unknown to Helgrind$^+$. In this case the detector acts as a pure happens-before detector. We symbolize this situation with *nolib+spin(7)* option in the table above. Only one additional test case fails (one false positive). However, we observe that the best results are achieved when using the new feature as a complementary method to our race detection algorithm (shown as *lib+spin(7)*) .

We compare these results with the results produced by *DRD 3.4.1* [15] a pure happens-before detector. Helgrind$^+$ achieves considerably better results. In particular, the number of false negatives with DRD is more than doubled than the false negatives with *nolib+spin(7)* option. Having false negatives is the main drawback of happens-before detectors.

Second part of the Table I depicts the results when using different number of basic blocks for detecting spinning read loops. By increasing the number of basic blocks, the number of false positives are decreased considerably. We got the best result with seven basic blocks and increasing it further will not improve the results. This is because the test suit uses function templates and complex function calls. Thus, spinning read loops in most test cases contain more than three basic blocks.

The second benchmark suite we used to evaluate our method is PARSEC 2.0 [1]. PARSEC 2.0 contains thirteen diverse multi-threaded programs from different domains. Table II depicts the summary of the programs. Except for `freqmine`, which uses *OpenMP* and `vips` that uses *Glib*[16], all programs use the standard PThreads for parallelization. At least eight applications use ad-hoc synchronizations in addition to standard synchronization primitives (locks, barriers and condition variables). We analyzed the result of executions with two threads per application on Helgrind$^+$. The empirical study in [17] implies that most concurrency bugs manifest themselves with only two 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 two threads.

The result of our experiments on PARSEC is demonstrated in Table III. The presented numbers are distinct program code locations that 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 are totals across all threads. The authors of the PARSEC Benchmarks claim the programs to be race free, however we cannot be absolutely sure that they are. Table III provides the false positives under the assumption that the programs are race free.

The results in Table III are as expected. Compared to basic version with option *lib*, the number of warnings produced by enabling the new feature *spin* is reduced considerably in many programs. In case of `dedup`, `facesim`, `streamcluster`, `vips` and `raytrace` all false warnings are eliminated. Two benchmarks `freqmine` and `vips` use unknown libraries: OpenMP and Glib. The number of warnings decreases to two and zero respectively. In `x264`, `dedup` the basic version of Helgrind$^+$ produces more than 1000 warnings (only a maximum of 1000 warnings is reported by the tools), whereas with the new feature (*lib+spin*) only 19 for `x264` remain. *ferret* generates only two warnings. Nine out of 13 applications do not produce any warnings.

We examine the warnings produced by our race detector with the new feature. All other warnings are benign races that can be counted as false warnings. The reasons for false warnings in some cases are synchronization constructs e.g. a *task queue* defined by the programmer that do not match the spinning loop pattern. For instance, consider `ferret` that uses a task queue and contains two benign races: A variable is used as counter for input packets. A single thread modifies it, while other threads read it without any synchronization. Another benign race is a variable that is used as signal to show if the input is read completely. In both cases, the condition variables are not used in a while loop.

If we switch off the support of PThreads (*nolib+spin*) so that the detector works as a happens-before detector based on identifying only spinning read loops (*nolib+spin*), approximately the same results are achieved. Only in four cases the number of false positives increased slightly. DRD produces more than 1000 warnings for some programs.

Overall, the results on various benchmarks confirm that Helgrind$^+$ with the new complementary method is able to discover ad-hoc synchronization and synchronization operations of unknown libraries without modifying or upgrading the race detector. The programmer is not overwhelmed with too many false alarms and the results appear acceptable for real world applications.

C. Performance Evaluation

We measured the runtime behavior and the memory requirements of our detector in different features on the basis of the PARSEC benchmark. Firstly, we measured the memory usage of instrumented code run by the detector. 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 6(a) depicts the average memory consumption. The memory consumption of Helgrind$^+$ is approximately constant across different

Program	Thread modell	LOC	Synchronization Method			
			Locks	Barriers	CVs	Ad-hoc
blackscholes	POSIX	812	-	√	-	-
bodytrack	POSIX	10,279	√	√	√	√
canneal	POSIX	4,029	√	-	-	-
dedup	POSIX	3,689	√	-	√	√
facesim	POSIX	29,310	√	-	√	√
ferret	POSIX	9,735	√	-	√	√
fluidanimate	POSIX	1,391	√	-	-	-
freqmine	OpenMP	2,706	-	-	-	-
streamcluster	POSIX	1,255	√	√	√	√
swaptions	POSIX	1,494	-	-	-	-
vips	Glib	3,228	√	-	√	√
x264	POSIX	40,393	√	-	√	√
raytrace	POSIX	13,302	√	-	√	√

Table II
SUMMARY OF PARSEC 2.0 BENCHMARKS.

Program	Instructions (10^9)		Racy Contexts			
	Reads	Writes	Helgrind$^+$ lib	Helgrind$^+$ lib+spin	Helgrind$^+$ nolib+spin	DRD
blackscholes	0.092	0.045	0	0	0	0
bodytrack	0.425	0.102	36.8	3.6	32.4	34.6
canneal	0.435	0.187	0	0	0	0
dedup	0.658	0.254	1000	0	2	0
facesim	9.632	4.191	113.8	0	0	1000
ferret	0.005	0.002	111	2	47	214.6
fluidanimate	0.584	0.144	0	0	0	0
freqmine	0.744	0.283	153.4	2	2	1000
streamcluster	1.795	0.033	4	0	1	1000
swaptions	1.414	0.365	0	0	0	0
vips	0.758	0.199	50.8	0	0	858.6
x264	0.500	0.204	1000	19	28	1000
raytrace	19.260	13,746	106.4	0	0	1000

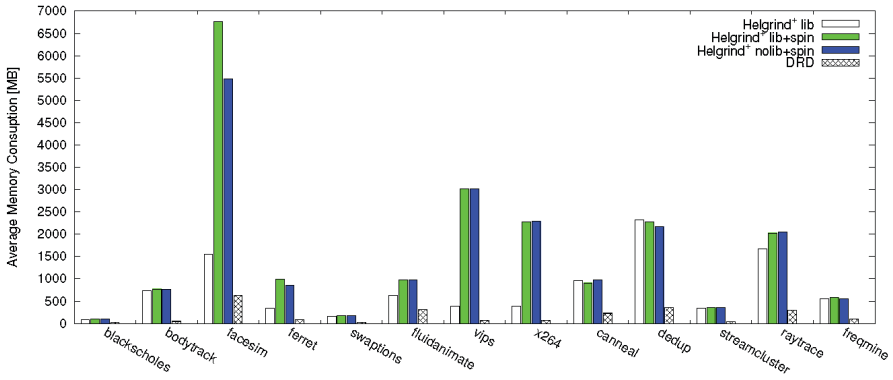
Table III
NUMBER OF RACY CONTEXTS REPORTED ON PARSEC 2.0 BENCHMARKS. ALL PROGRAMS ARE EXECUTED FOR INPUT SET `SIMSMALL` EXCEPT `SWAPTIONS` AND `STREAMCLUSTER` THAT IS FOR `SIMMEDIUM`.

modes. There is some overhead caused by the new features implemented for the ad-hoc feature. Compared to DRD the memory overhead is higher. However, the memory overhead in Helgrind$^+$ is small enough that real world applications with higher memory requirements are still testable. Optimizing our implementation could help reduce the memory overhead, which we intend to do as a future work.

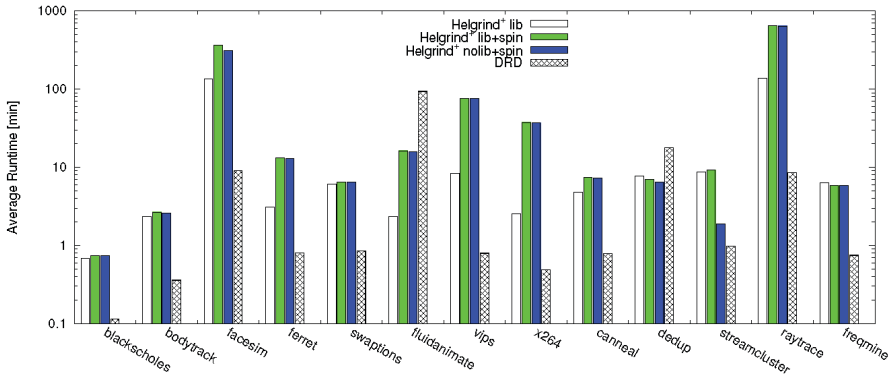
The execution time of instrumented code versus the actual execution time is typically slowed down by a factor of 10 to 50 on Helgrind$^+$. We measured the execution time of instrumented code on different modes. The measurements are shown in Figure 6(b). There is also some overhead of Helgrind$^+$ over the basic mode (*lib* option). In the worst cases, `x264` and `vips` on Helgrind$^+$ with *lib+spin* and *nolib+spin* options increase the execution time significantly. In most cases, Helgrind$^+$ delivers approximately equal execution times in all modes. As the figure shows, compared to DRD, there is an execution overhead. But in case of `dedup` and `fluidanimate` DRD's execution time is much higher than the Helgrind$^+$. This is because many locks are used in these two benchmarks compared to the other programs. On average, the slowdown factor is reasonable to apply for different kind of applications and get accurate results.

V. RELATED WORK

Prior work can be divided into two categories with respect to the data race detection approach: static and dynamic. Static approaches are not accurate, produce many false positives and false negatives. They consider all potential thread interleavings including those that are not feasible [18], [19]. Dynamic approaches report only races that actually occur during the program execution. They are either based on the lockset algorithm [20] or happens-before analysis [21]. The lockset algorithm produces too many false positives but it is simple and can be implemented with low overhead. On the other hand, happens-before analysis is difficult to implement. It may miss races, i.e. produce false negatives, as it is sensitive to the order of execution.



(a) Memory consumption on PARSEC 2.0



(b) Execution time on PARSEC 2.0

Figure 6. Memory consumption and execution time on PARSEC 2.0 by different tools.

Recent dynamic approaches [22], [23], [24], [25], [26], [27] have combined these two methods into a hybrid method with the strengths of each. But no one really succeeded and they still produce many false positives and even miss races.

Basically this is because they suffer from two serious limitations. Firstly, they are not able to detect ad-hoc synchronizations implemented in user code itself. Secondly, the detectors are restricted to synchronization primitives of a specific library and any synchronization primitive used from another library in the program is simply ignored. Thus, they are not able to produce accurate reports and applying these detectors to real applications overwhelms the user with too many number of false alarms. Our work removes these limitations by introducing a general approach for detecting ad-hoc synchronizations and unknown synchronization primitives. We are able to eliminate false positives including benign synchronization races and possible false negatives caused due to missed or incorrect synchronizations.

In our previous works[4], [5], we proposed a dynamic hybrid approach that employed heuristics to combine lockset algorithm and happens-before. We presented two memory state models optimized for *short-running* and *long-running* applications with an automatic technique that is able to detect synchronization caused by inter-thread event notifications (condition variables). The current work is a complement to our previous works. It identifies ad-hoc synchronizations and unknown synchronization primitives that are hidden to race detectors. This work could be used as complementary method to any other race detector.

Also, the method presented in this work is general and could be used as a complete race detection approach in a race detector. The resulted race detector is a universal happens-before race detector. Compared to other happens-before race detectors such as DRD [15], this method induces also happens-before edges when using ad-hoc synchronization or unknown synchronizations primitives, resulting in substantial accuracy.

A dynamic technique in [2] is used to identify synchronization operations in programs. The method is only able to partially suppress false positives caused by apparent races and benign synchronization races. It is merely based on actual spinning reads occur at runtime and set a threshold value for the number of spinning reads to identify them during execution. The value of the threshold is set heuristically (they set the number of spin reads to three). If the spinning read does not happen, the detector will miss the synchronization. As discussed earlier, this could happen when using condition variables or in ad-hoc synchronization

that causes false alarms. Furthermore, the method could detect by mistake loops in the program as spinning reads and interpret them as synchronization operation. This could lead to misinterpretation of synchronizations for the detector and causes false negatives (missed races).

VI. Conclusion

In this work, we have shown that the knowledge of all synchronization taking place in the program is crucial for accurate data race detection. We demonstrated that missing ad-hoc synchronization causes a lot of false positives. The presented dynamic method in this paper is able to identify ad-hoc synchronizations. It is also able to detect synchronization primitives of unknown libraries eliminating the need of upgrading the detectors. Our empirical results confirm that our method could be used as a complementary method to achieve the optimum results removing false alarms with almost no false negatives. Furthermore, the evaluation shows that the overhead caused by the new method in our race detector is moderate enough to apply in practical applications.

Using the method alone as a complete race detection approach results in a universal happens-before race detector that is able to detect all different synchronization operations with minor increase in false positives. A direction for our future work is improving the accuracy of the universal race detector by identifying lock operations, enabling lockset analysis. Helgrind$^+$ is open source and available at http://svn.ipd.uni-karlsruhe.de/trac/helgrindplus.

Acknowledgments
We would like to thank Kaibin Bao for his support during our work and Markus Westphal for reviewing this paper.

References

[1] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," June 2009.

[2] 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.

[3] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar, "Healing data races on-the-fly," pp. 54–64, 2007.

[4] 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.

[5] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–13, 2009.

[6] 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.

[7] P. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," pp. 165–171, 1994.

[8] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[9] R. Gupta, "The fuzzy barrier: a mechanism for high speed synchronization of processors," *SIGARCH Comput. Archit. News*, vol. 17, no. 2, pp. 54–63, 1989.

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

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

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

[13] ——, "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

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

[15] ——, "Drd: a thread error detector," 2009. [Online]. Available: http://valgrind.org/docs/manual/drd-manual.html

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

[17] 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.

[18] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 237–252, 2003.

[19] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data racesallusing replay analysis," *SIGPLAN Not.*, vol. 42, no. 6, pp. 22–31, 2007.

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

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

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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