Library-Independent Data Race Detection

Ali Jannesari and Walter F. Tichy

Abstract—Data races are a common problem on shared-memory parallel computers, including multicores. Analysis programs called race detectors help find and eliminate them. However, current race detectors are geared for specific concurrency libraries. When programmers use libraries unknown to a given detector, the detector becomes useless or requires extensive reprogramming. We introduce a new synchronization detection mechanism that is independent of concurrency libraries. It dynamically detects synchronization constructs based on a characteristic code pattern. The approach is non-intrusive and applicable to various concurrency libraries. Experimental results confirm that the approach identifies synchronizations and detects data races regardless of the concurrency libraries involved. With this mechanism, race detectors can be written once and need not be adapted to particular libraries.

Index Terms—Parallel programming, parallelization libraries, ad hoc synchronization, synchronization primitives, dynamic analysis, data race detection, debugging, multicore

1 Introduction

Multicore computers have become mainstream, but programming them remains difficult. A particular problem is data races, i.e., situations where several processors simultaneously access the same variable, and at least one of them writes it. Data races may lead to inconsistent program states that are extremely difficult to debug.

To mitigate this problem, automatic detectors have been developed that identify program locations where data races may occur. These detectors find potentially simultaneous accesses to variables and check whether synchronization constructs such as monitors, barriers, or wait conditions serialize the accesses. If serialization is lacking, the detector reports a race. For this to work, the detector must know about the synchronization constructs that programmers and compilers are using. If the detector is uninformed about some of the synchronization constructs, it may overwhelm the programmer with false positives.

The job of race detectors is made difficult by the proliferation of libraries and parallel programming languages. Examples of current parallel programming models include the following: POSIX API (Pthread), Cilk [1], OpenMP [2], Galois [3], OpenCL [4], StreamIt [5], X10 [6], TBB [7], XJava [8], and CUDA [9]. These programming models offer various constructs for parallelism, such as basic threads, pipelines, master/worker, producer/consumer, data parallelism, futures, work stealing, task pools, and more. Most of these implement implicit synchronization. There is also a variety of explicit synchronization primitives, for example monitors, semaphores, barriers (cyclic and non-cyclic), compare-and-swap, atomic data types, transactional memory, and others. No race detector covers all of these constructs, even though a programmer may mix several libraries in a single program, for example OpenMP or OpenCL together with Pthread or TBB. Programmers sometimes even avoid predefined synchronization constructs altogether by programming ad hoc wait loops or implementing their own barriers [10]. In such cases, traditional race detectors have no clue where synchronization occurs and therefore may produce a large number of false positives. Adapting a race detector to a new library, however, is not trivial.

We present an approach that eliminates the dependence of race detectors on libraries. The technique identifies synchronization operations automatically by searching for a low-level code pattern that occurs in locks, barriers, semaphores, etc., in high-level concurrency constructs such as pipelines, and even in user-defined, ad hoc synchronization code. The identification of synchronization operations happens at runtime. A sophisticated, hybrid race detection algorithm determines whether data races may indeed occur.

We implemented our approach in the open source tool Helgrind$^+$ [11], [12].[1] We demonstrate that the tool can identify synchronization constructs from a wide range concurrency libraries, without any information about them. Experimental results show that Helgrind$^+$ identifies data races with high precision, independent of the libraries used.

The remainder of this paper is structured as follows: In Section 2, we present some backgrounds and discuss synchronization operations in state-of-the-art programming models and libraries. Section 2.1 describes our approach. We explain the algorithm that identifies synchronization operations and describe the detector. We briefly cover implementation aspects in Section 3. An

- *A. Jannesari is with the German Research School for Simulation Sciences, Aachen, Germany, and also with RWTH Aachen University, Aachen, Germany. E-mail: ali.jannesari@rwth-aachen.de; jannesari@grs-sim.de.*
- *W.F. Tichy is with Karlsruhe Institute of Technology (KIT), Germany. E-mail: tichy@kit.edu.*

Manuscript received 10 Oct. 2012; revised 10 July 2013; accepted 8 Aug. 2013. Date of publication 20 Aug. 2013; date of current version 17 Sept. 2014. Recommended for accepting by D. Kaeli.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.209

1. Helgrind$^+$ is available at https://svn.ipd.kit.edu/trac/helgrindplus/wiki

1045-9219 © 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

evaluation is provided in Section 4. Section 5 describes related work.

2 Synchronization

Synchronization may be explicit or implicit. Explicit synchronization means that programmers insert calls to synchronization constructs such as lock/unlock or signal/wait into their code. Implicit synchronization, on the other hand, is more or less hidden. It is included in library or language constructs whose main purpose is not synchronization. For instance, parallel loops typically include a barrier synchronization at the end. Fig. 1 illustrates this case for OpenMP. A parallel loop computes a result to be displayed later. To avoid that the main thread runs ahead and displays an incomplete result, an implicit barrier blocks this thread until all worker threads have completed the parallel loop. The barrier is inserted by the compiler. The advantage of implicit synchronization is that programmers can not forget it, thus preventing a source of errors.

The objective of concurrency libraries is to simplify parallel programming by providing high-level constructs such as data parallel loops, pipelines, master/worker, or queues. All of these provide built-in, implicit synchronization. As an example, consider Threading Building Blocks (TBB) [7]. TBB is an open source concurrency library. It encourages the creation of parallel applications through task and pipeline parallelism. A mechanism called task stealing mitigates load imbalance. TBB can be combined with other concurrency libraries such as OpenMP. Its constructs typically provide implicit synchronization. Fig. 2 shows an example of a 3-stage pipeline. The stages are called filters. The first filter reads an input file, while the second capitalizes all letters, and the last stage writes the data to an output file. The code in Fig. 3 implements this pipeline. No explicit synchronization operations are needed.

High-level concurrency constructs abstract away low-level synchronization primitives, so programmers make fewer mistakes [13]. However, a data race detector must be able to identify all locations where synchronization occurs, because otherwise it produces too many false warnings.

2.1 Characteristic Code Pattern

A simple method to help a detector would be to annotate both explicit and implicit synchronization operations. However, this method requires programmer intervention, is intrusive, error prone, and forces recompilation. An automatic identification method is highly preferred.

Prior research [14], [15], [10], and [12] observed that high-level synchronization constructs use synchronization

```
int i, j;

#pragma omp parallel for
for (i=0; i<n; i++)
     for (j=0; i<m; j++)
          a[i][j] = compute(i,j);

display(a);
```

Fig. 1. Implicit *barrier* synchronization used in OpenMP.

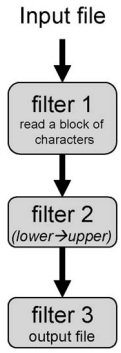

Fig. 2. Example for parallelization: File transformation with a 3-stage pipeline.

primitives provided by threading APIs and the operating systems. For example, the debuggers Helgrind [16] and DRD [17] intercept low-level POSIX calls and thereby capture high-level constructs built from them as well. However, different libraries may use different primitives. For instance, both Helgrind and DRD fail on OpenMP [12]. Moreover, programmers sometimes write explicit wait loops that do nothing but wait for a variable to change. The wait loop uses no synchronization primitive whatsoever, but must still be identified by a race detector as a location where an ordering relation is enforced.

It turns out that synchronization primitives rely, in the end, on a simple, characteristic code pattern. This has been confirmed in publications [18], [19], [20], [14], [15], and [12], which studied the implementations of locks, conditions variables (signal/wait), barriers, and semaphores, as well as ad hoc synchronization. The pattern is as follows:

1. It consists of an *indicator variable*, one or more loops that wait for the variable to assume an *expected value*, and one or more locations in the program that write the expected value into the variable. We call the loops *spinning loops* and the corresponding write operations *counterpart writes*.
2. A spinning loop terminates when it detects the expected value in the indicator variable.

```
// Create the pipeline
tbb::pipeline pipeline;

// produce input filter
// read input file, generate stream
MyInputFilter input_filter( input_file );
pipeline.add_filter( input_filter );

// process filter
// transformation: up casing letters
MyTransformFilter transform_filter;
pipeline.add_filter( transform_filter );

// output filter for output file
MyOutputFilter output_filter( output_file );
pipeline.add_filter( output_filter );

// execute pipeline
pipeline.run( MyInputFilter::n_buffer );
pipeline.clear();
```

Fig. 3. Three-stage pipeline implemented with TBB.

```
lock(int *mutex)
{
  while(!test_and_set
      (mutex,0,1))
    {/*do nothing*/}
}
```

(a)

```
unlock(int *mutex)
{
  *mutex = 0;
}
```

(b)

Fig. 4. Implementation of synchronization primitives $lock()$ and $unlock()$. (a) $Lock()$ implemented as spinning loop and atomic write. (b) Counterpart write operation.

3. A spinning loop either tests or atomically tests and sets the indicator variable (see Section 2.2). The spinning loop does not write the indicator variable otherwise.

Fig. 4 shows how this pattern is used in the implementation of the lock/unlock primitives of the POSIX API. The lock() operation is implemented as a busy wait. For the rest of this paper, it is irrelevant whether spinning loops are implemented as busy waits or give up the processor.

2.2 Detection Algorithm

Helgrind$^+$ identifies spinning loops and counterpart write operations on the fly. Interestingly, this is enough to detect synchronization operations.

2.2.1 Synchronization Primitives

Table 1 shows the protocols observed by several synchronization primitives (the primitives are common and could be provided by various concurrency libraries). Both the primitives lock() and sem_wait() include a spinning loop with an (atomic) test-and-set operation on an indicator variable. The loop changes the indicator variable to a value other than the expected value as soon as the expected value has been encountered (see Fig. 4a for the loop in lock() primitive). This protocol (also implementable with a compare-and-swap instruction) assures that only a single thread can continue past its spinning loop for every write of the expected value. The unlock() operation contains nothing but the counterpart write (see Fig. 4b). Lock() and unlock() pairs on the same indicator variable are customarily used to implement mutual exclusion and monitors. An important observation for later is that a lock() and unlock() pair is completed in sequence by the same thread.

When a thread needs to send a signal to a different, waiting thread, cond_send() and cond_wait() are used. The cond_wait() primitive also uses a spinning loop, but only tests the indicator variable, without resetting it. This protocol is sufficient if cond_wait() is not used repeatedly (as inside a loop), or is reset elsewhere. A cond_send() is simply a counterpart write into the shared indicator variable; it can signal one or several spinning threads.

The semaphore operations sem_wait() and sem_post() can be used for both signaling and mutual exclusion. The sem_wait() operation first enters a spinning loop on an indicator variable to get exclusive access to the semaphore data structure. This data structure includes a counter, which sem_wait() decrements. If the counter variable turns negative, the executing thread is placed on a waiting list.

TABLE 1
Protocols of Low Level Synchronization Primitives

	spinning loop	counterpart write
lock()	test&set	–
unlock()	–	set
cond_wait()	test	–
cond_signal()	–	set
sem_wait()	test&set	set
sem_post()	test&set	set
barrier_wait()$_{i:1..n-1}$	test&set(go$_i$)	set(present$_i$)
barrier_wait()$_n$	test&set(present$_{1..n-1}$)	set(go$_{1..n-1}$)
ad hoc synchronization	test	set

Finally, sem_wait() resets the indicator variable to allow other threads access to the semaphore. Sem_post() also needs exclusive access to the semaphore data structure. Hence, both semaphore operations are wrapped in a spinning loop and a counterpart write on the same indicator variable.

Barriers are somewhat more complicated, but also work with spinning loops. Each thread has two indicator variables: $present_i$ and go_i. When checking into a barrier, thread i writes $present_i$ to indicate that it is inside the barrier and then immediately starts spinning on go_i using test-and-set. A designated thread, say n, spins on all variables $present_i$, one after the other, with test-and-set. When it continues, all threads have checked into the barrier, and all variables in array $present$ have been reset. At this point, thread n releases the waiting threads by writing all the variables go_i. A simplification would be to use a single variable go instead of an array (and no reset), but this implies that the barrier cannot be used in a cyclic fashion. Barrier synchronization can also be implemented with lock() and unlock(), which themselves use spinning loops. For large sets of processors, the loops for waiting and releasing can be accelerated by using fan-in and fan-out trees, but the core is always the spinning loop.

2.2.2 Ad hoc Synchronizations

So far we have discussed synchronization primitives as provided by libraries. However, users may define their own synchronization operations in an ad hoc way. In a study of concurrent programs (Apache, MySQL, Mozilla, and others) Xiong *et al.* [10] found that programs contain a surprising number of ad hoc synchronizations (6-83 ad hoc synchronizations in each program studied). Also in a former study [12] we showed ad hoc synchronizations occur frequently. For instance, we found that eight of the 13 PARSEC benchmarks [21] contain various ad hoc synchronizations. Hence, race detectors must identify ad hoc synchronizations to avoid false positives. Our analysis showed that ad hoc synchronization operations also follow the pattern of spinning loop with counterpart write (for details see [12]). If the race detector can identify this pattern, it can handle ad hoc synchronization in the same manner as standard synchronization primitives and implicit synchronization embedded in high-level language constructs.

The distinguishing properties of a spinning loop are that it repeatedly tests the same location without modifying it, except perhaps in a test-and-set instruction (which changes it only if the loop terminates). In practice, spinning loops

```
int trueSpin(int * mutex)
{
    int offset;
    while(1) {
        offset=0;
        if(mutex[offset]) break;
    }
}
```

Fig. 5. Tricky spinning loop used for ad hoc synchronization.

are short. However, they take on a variety of forms. Fig. 5 shows an example of an ad hoc synchronization taken from the Parsec benchmark suit [22]. At first glance, the loop appears to be infinite. The analysis must identify the terminating break in the conditional statement. A second difficulty is that the variable tested is an array element. To make sure that the same element is tested in every iteration, a dependency analysis is needed. In this case the dependency graph is loop free, therefore the same location is tested in each iteration and the spinning loop property is met.

Helgrind$^+$ identifies spinning loops in machine code on the fly. It uses dynamic data and control dependency graphs spanning multiple blocks (look-ahead technique), including calls. Hence, spinning loops are detected even if spread out over several basic blocks (concatenation of referenced basic blocks). Complex loops with multiple entry and exit points are handled properly by performing an accurate data flow analysis using techniques such as normalization of dependency graphs, distinguishing direct and indirect data dependencies and resolving copy (constant) propagation. The indicator variables belonging to the loops are marked and instrumented. Helgrind$^+$ does not search the code for counterpart writes, but waits for them to occur dynamically. A counterpart write is the *first* write operation on an indicator variable outside the spinning loop. As soon as a counterpart write into an indicator variable occurs, the write operation is intercepted, the spinning loop and counterpart write are paired, and a happens-before edge ($\rightarrow_{\text{hb}}$ edge) is established. The happens-before relation specifies a temporal ordering among code segments and is used in the race detector to rule out potential race conditions.

Since Helgrind$^+$ is a hybrid race detector that works with both the happens-before relation and the lockset algorithm, it must detect lock()/unlock() pairs as well, but in a library-independent manner. Detection is based on the fact that lock()/unlock() pairs are executed by the same thread. Furthermore, the spinning loop of the lock operation must contain a test-and-set (or compare-and-swap) instruction. Thus, when a write occurs to an indicator variable from a spinning loop, Helgrind$^+$ simply records the thread identifier. When a succeeding counterpart write occurs, it checks whether the thread identifiers are identical, and if so, the indicator variable is marked as a lock. This lock protects the variables accessed between the spinning loop and the counterpart write. The race detector uses this information to determine whether the variables are consistently protected throughout the program. One could work without lockset analysis, but our results show that a hybrid approach is more accurate. An example where lockset analysis beats a pure happens-before detector is given in Section 4.2.

2.2.3 Dealing with Semaphores

Semaphores require extra consideration. Since they may be used for either signaling or locking, Helgrind$^+$ must distinguish the two uses (one identifying a $\rightarrow_{\text{hb}}$ edge, the other a lock). Furthermore, Helgrind$^+$ is library independent, so we cannot use the library names of the semaphore operations–they may differ from library to library. Therefore, the detection algorithm has to work on the instruction level. Luckily, the above technique can be extended to work with semaphores. Recall that each semaphore operation is itself wrapped in a spinning loop and a counterpart write, or a lock()/unlock() pair (which is the same). At first, this fact only informs Helgrind$^+$ that the access to the semaphore data itself is properly protected. What we need is a connection between the code segments for sem_wait() and sem_post() bracketing a critical section. As with lock() and unlock(), the two semaphore operations will be executed in sequence by the same thread in the case of a lock. When sem_wait() is executed, its last action will be a counterpart write into the indicator variable that protects the semaphore. At this point, we record the identity of the executing thread. Other threads may also try the sem_wait() operation, but they will be blocked if the semaphore counter was properly initialized for a lock. Eventually, the original thread completes the critical section and executes sem_post(). This means the thread enters another spinning loop and leaves it with a successful test-and-set execution. This write operation is caught by Helgrind$^+$. It checks whether the thread identifier is the same as the one recorded by the previous counterpart write (which occurred when the thread left the previous sem_wait()), and if so, it has detected a lock. In essence, the indicator variable of the semaphore is protecting not only the semaphore data structure itself, but also the critical section between the semaphore operations. If the semaphore is used for signaling instead, the thread identifiers will fail to match and therefore no lock will be recorded (other than for the semaphore data itself). In that case, a $\rightarrow_{\text{hb}}$ edge is added.

Of course, the detection algorithm depends entirely on the presence of spinning loops. If other means of synchronization are used, such as hardware interlocks (preventing race conditions by hardware support), the race detector could be extended to detect these patterns as well. Our benchmark programs did not use techniques other than spinning loops, but in a variety of forms.

3 Implementation

We use our race detector Helgrind$^+$ as the basis of our implementation. Helgrind$^+$ is an open source tool implemented on top of a dynamic binary instrumentation tool called *Valgrind* [24], [23]. Valgrind is a framework for instrumenting binary code. It *disassembles* and *re-synthesizes* code dynamically during just-in-time instrumentation. The framework translates binary code into a platform-independent intermediate representation (IR). Our tool instruments the IR and hands it back to

```
On Event indicator variable spinVar is written by Thread tid
    handleWriteAccess(spinVar, tid)

Function handleWriteAccess(spinVar, tid)
    if inSpinLoop() then
        if (spinVar.lastReleaseWriter ≠ 0) & (!spinVar.wasReleasedBefore) then
            // last write access outside of a loop does not belong to any lock
            spinVar.lastSpinWriter = 0  // no lock
        else
            spinVar.lastSpinWriter = tid  // try to acquire the lock
        end
    else
        spinVar.lastReleaseWriter = tid
        if Thread tid holds Lock spinVar then  // writing thread currently holding the lock
            spinVar.wasReleasedBefore = True
            registerUnlockOperation(tid, spinVar)  // release of the lock
        else // cannot be a lock
            spinVar.wasReleasedBefore = False
        end
    end

On Event Thread tid leaves spin loop
    foreach spinVar ∈ {indicator variables spinVar/spinVar.lastSpinWriter = tid} do
        registerLockOperation(tid, spinVar)  // Thread tid holding now the lock spinVar
    end
```

Algorithm 1: Simplified algorithm for implementing lock primitives detection.

Valgrind, which re-synthesizes machine code from it and then executes it.

3.1 Spinning Loops and Indicator Variables

Roughly speaking, detection of spinning loops and indicator variables works as follows:

- Search the control flow graph for loops that span a maximum of seven basic blocks.[2]
- Track the data dependencies of each variable within the basic blocks and construct a data dependency table.
- A variable is an indicator variable if it is tested but not written inside the loop, except possibly in a test-and-set or a compare-and-swap instruction.

Helgrind$^+$ considers all loops (branches) at the IR level. Function calls are transformed into regular branches and their code is inlined until at most seven blocks have accumulated. Several steps for optimizing and simplifying the dependency graphs are performed. Normalization of dependency graphs and optimization operations such as resolving copy/constant propagation are done on the fly. Distinguishing direct and indirect dependencies and concatenations of referenced basic blocks enable a precise and consistent analysis.

During the instrumentation, the instructions in the spinning loop and the indicator variable are instrumented. During runtime, if a counterpart write occurs on any indicator variable, the write is captured and used for establishing a happens-before relation or updating a lockset. The algorithm distinguishes between signals and locks as discussed earlier.

2. We experimented with different numbers; seven blocks gave the best detection results without undue slowdown.

Algorithm 1 implements a simplified version of the detection of lock primitives. For each indicator variable $spinVar$ we store the following infromation:

- $lastSpinWriter$: the ID of the last thread (tid) writing the indicator inside a spin loop.
- $lastReleasedWriter$: the ID of the last thread writing the indicator outside of a spin loop.
- $wasReleasedBefore$: a boolean value which indicates if the last thread writing the indicator outside of a spin loop held the associated lock. The variable $wasReleasedBefore$ indicates that $spinVar$ is used for lock operation.

We record the $lastSpinWriter$ of the indicator inside the spin loop, if there is no write operation outside of the spin loop on the indicator by another thread. In other words, no write operation on the indicator outside of the loop is allowed, except by the thread currently holding the associated lock. In case of write operation outside of the spin loop the $wasReleasedBefore$ is set and the information is used for the next lock operation again. When leaving the spin loop, the written indicators inside the loop are registered as acquired locks by the thread. If it is not a matter of a lock operation, but a barrier or a condition variable, no write operation within the loop is expected and consequently, the algorithm does not designate the loop as a lock operation. For the sake of simplicity, the algorithm does not depict the distinction between read/write locks and semaphores.

To maintain state information on indicator and other variables, we use shadow memory [25]. Shadow memory stores for each memory location the information needed for runtime analysis, e.g., its state, thread ID, and other information.

TABLE 2
Results on the Test Suite that Contains 131 Programs in Pthread, 15 Programs in OpenMP, and 26 Programs in TBB

Results	library-independent Helgrind$^+$			library-dependent		
	Pthread	OpenMP	TBB	Intel TC Pthread	Helgrind Pthread	DRD Pthread
True Positives (TP)	34	3	6	32	25	20
True Negatives (TN)	73	11	16	65	36	67
False Positives (FP) (unidentified synchronizations)	13	0	2	11	54	20
False Negatives (FN) (unidentified races)	11	0	1	23	16	24
Failed test cases (FP+FN)	24	0	3	34	70	44
Passed test cases (TP+TN)	107	15	23	97	61	87
Sum of all test cases	131	15	26	131	131	131

3.2 Data Race Detection

Helgrind$^+$ is a hybrid race detector, i.e., it uses a combination of lockset algorithm and happens-before analysis ($\rightarrow_{\text{hb}}$). The main idea is to apply the happens-before analysis whenever the lockset algorithm indicates a potential data race. A memory state machine maintains the state of each (non-indicator) variable in shadow memory. Through the state machine, Helgrind$^+$ can distinguish between parallel and ordered accesses to shared variables.

The indicator variables of spinning loops are not subject to the state machine: Accesses to them cause changes in locksets or add $\rightarrow_{\text{hb}}$ edges, which are then used in the state machine for regular variables. See [26] and [11] for more details on the state machine and the race detection algorithm. These earlier papers required information about synchronization primitives to drive the state machine. By detecting spinning loops, counterpart writes, and the distinction between locking and signaling, knowledge about synchronization primitives is no longer needed in the race detector.

4 Evaluation and Results

We evaluated Helgrind$^+$ with a number of benchmarks and compared the results with other race detectors. Helgrind$^+$ correctly identified synchronization operations from various concurrency libraries and detected true data races, while keeping the number of false positives and false negatives low. A true data race affects the behavior and the result of the program.

4.1 Experimental Setup

Our evaluation was conducted on a 2x Intel XEON E5320 Quadcore at 1.86 GHz, 8 GB RAM, running Linux Ubuntu x64. Programs were compiled with gcc 4.2.3. All measurements and reported values are averages over five executions. No source code annotations were used. The 64-bit version of Valgrind 3.4.1 was the basis for Helgrind$^+$. We compared Helgrind$^+$ with other existing race detectors, i.e. the original Helgrind [16], DRD [17] and Intel Thread Checker [27]. Despite the lack of information about synchronization primitives, our results are acceptable and in some cases even better than those of existing race detectors.

4.2 Small Test Programs

Data-race-test [28] is a benchmark suite for race detectors with 120 short programs (called test cases). For each test case, it is specified whether it includes a race or it is a race-free program. The test cases implement various scenarios, including tricky situations that are difficult to analyze. All test cases are implemented in C/C++ using the POSIX API (Pthread) synchronization primitives or some ad hoc synchronization. The programs run with varying numbers of threads. No annotations were provided. We added 11 new and difficult test cases. Furthermore, where possible, we ported test cases to OpenMP and TBB, giving us 15 and 26 extra cases, resp. These test cases are used to check synchronization detection in additional libraries. Overall, we have 131 test cases implemented with Pthread, 15 test cases implemented in OpenMP, and 26 in TBB.

Table 2 shows the results. Helgrind$^+$ does not require any information about concurrency libraries and handles test cases written in Pthread, OpenMP, and TBB. The library-dependent tools Intel TC, original Helgrind, and DRD only handle Pthread test cases.

For Pthread, Helgrind$^+$ fails on only 24 out of 131 cases (false positives and false negatives), while handling 107 cases correctly (true positives and true negatives). Helgrind$^+$ beats all other contenders, although it knows nothing about the Pthread library. Part of the reason is that the library-dependent tools are unaware of ad hoc synchronization or complex constructions such as wrappers, templates, and function calls in combination with Pthread. Helgrind$^+$ handles spinning loops even if they involve templates and function calls. In such situations, the spinning loop spans multiple blocks. We varied the number of blocks analyzed and obtained the best results with seven basic blocks.

Intel TC, a commercial tool, produces only 11 false positives, which is the lowest false positive rate among the tools. However, it misses 23 races, which is a high rate of false negatives. The second library-dependent tool is Helgrind 3.4.1 [16] and the third is DRD 3.4.1 [17], a happens-before race detector.

For OpenMP, Helgrind$^+$ produced no false positives or false negatives, and only three of the TBB cases were handled incorrectly. The other tools, if applied to OpenMP and TBB cases, overwhelm the user with false warnings, as they have no clue about the synchronization primitives present.

A simple case correctly handled by both Helgrind$^+$ and Intel TC is shown in Fig. 6. Pure happens-before race detectors such as DRD may not identify the data race in this example. The reason is that a $\rightarrow_{\text{hb}}$ edge is established from the left `unlock()` to the right `lock()`. This edge implies that

```
y = y+1;
lock(m);
  x = x+1;
unlock(m);
```

```
lock(m);
  x = x+1;
unlock(m);
y = y-1;
```

Fig. 6. Data race on y missed by happens-before analysis but detected by Helgrind$^+$.

the left access to variable y occurs before the right access. However, this is incorrect. The two accesses are actually not ordered. Consider what happens when the right `unlock()` completes first. This is an example where lockset analysis improves detection accuracy. At least three such cases cause false negatives with Helgrind and DRD.

Helgrind$^+$ shows some false positives and false negatives (missed data races). Some of the false positives are caused by tricky and complex forms of ad hoc synchronization. However, most of the false positives produced by other race detectors are removed by Helgrind$^+$ (e.g., 25 false positives produced by Helgrind are removed because of correct handling of ad hoc synchronizations and implicit synchronizations).

An example of a false negative is given in Fig. 7. The loop body applies pointer arithmetic, masking the modification of the loop variable (it writes the second byte of `lock`). However, a loop that modifies its own termination condition is not a spinning loop. Since our algorithm does not handle variable sizes, it mistakenly identifies this loop as a spinning loop and introduces a $\rightarrow_{hb}$ edge where there is none. As a result, the detector misses a data race. Taking variable sizes into account is possible, but causes overhead.

Another example of a false negative is given in Fig. 8. Race detectors are not able to identify the data race in this example. The reason is that `x` is protected by `lock` and `unlock` or a $\rightarrow_{hb}$ edge is established between `unlock` and `lock`. However, this is insufficient. Both threads assign different values to `x` and the two accesses are not actually ordered. This is an example where `order violation` happens. If both threads increment `x` by one, this example does not include a race and race detectors can handle it correctly.

4.3 Real World Applications

The second benchmark suite is PARSEC 2.1 [21]. It consists of thirteen diverse multi-threaded programs from different domains. Some of the benchmarks are available in multiple implementations, using either Pthread, TBB, or OpenMP [29]. Table 3 summarizes the suite (LOC provides lines of code when implemented with Pthread, except in the case of freqmine which is implemented with OpenMP). Eight of the thirteen applications use ad hoc synchronization. `Freqmine` includes 131 ad hoc synchronization constructs [12]. We ran

```
int lock=0;
void falseSpin() {
  do {
    char c=1;
    ((char*)&lock)[1]=c; // 2nd byte written
                         //within loop
  } while(!lock);
}
```

Fig. 7. Problem with pointer arithmetic.

```
lock(m);
  x = 2;
unlock(m);
```

```
lock(m);
  x = 3;
unlock(m);
```

Fig. 8. Problem with order violation.

the benchmarks with two threads per application. Lu*et al.* [30] report that most concurrency bugs manifest themselves with only two threads. Furthermore, Helgrind$^+$ schedules threads in a more fine-grained way than operating systems. Consequently, we assume that many races can be observed with two threads.

Because of the large memory overhead and computational cost, we did not use the native input set. Instead, we used the `simsmall` or `simmedium` inputs and ran each program five times, averaging the results. The authors of the PARSEC benchmarks claim the programs to be race free, but we cannot be absolutely sure that they are. In some cases, we contacted the authors.

Table 4 shows the results of the various race detectors. The numbers reported are distinct program contexts that produce at least one data race warning. For simplicity, we will simply call these contexts warnings, without saying how many warnings each context actually produced.

The first two applications in the table are implemented with three concurrency libraries (Pthread, OpenMP, TBB) each. `Blackscholes` produces no warnings at all, regardless of tool and library. For `bodytrack`, Helgrind$^+$ generates four warnings (in all three variants). The application uses read-write locks and ad hoc synchronization. On close inspection, all four warnings turned out to be false positives. All warnings produced by the other tools are also false positives.

For `canneal`, `dedup`, `facesim`, `ferret`, `fluidanimate`, `raytrace`, and `swaptions`, Helgrind$^+$ generates no warnings. The benchmark `freqmine` is implemented with OpenMP. Helgrind$^+$ catches two (intentional) races and again produces no false warnings. Overall, twelve out of twenty variants produce no warnings. However, the other tools produce hundreds or thousands of false warnings.

The benchmark `vips` uses the library Glib [31] and produces three false warnings under Helgrind$^+$. Glib implements threads and primitives such as mutexes, etc. The thread support in Glib is based upon Pthread or win32 threads; a Pthread version was used here.

TABLE 3
Implementation Variants of the PARSEC 2.1 Benchmarks

Program	LOC	Concurrency library used		
		Pthread	OpenMP	TBB
blackscholes	812	√	√	√
bodytrack	10,279	√	√	√
canneal	4,029	√	-	-
dedup	3,689	√	-	-
facesim	29,310	√	-	-
ferret	9,735	√	-	-
fluidanimate	1,391	√	-	√
raytrace	13,302	√	-	-
swaptions	1,494	√	-	√
freqmine	2,706	-	√	-
streamcluster	1,255	√	-	√
vips	3,228	√	-	-
x264	40,393	√	-	-

TABLE 4
Number of Race Contexts Reported by Tools

Program	Helgrind$^+$ (library-independent)			library-dependent Tools (Pthread-dependent)		
	Pthread	OpenMP	TBB	Intel TC	Helgrind	DRD
blackscholes	0	0	0	0	0	0
bodytrack	4	4	4	13	51	31
canneal	0	-	-	4	1	0
dedup	0	-	-	0	3	0
facesim	0	-	-	0	128	1000
ferret	0	-	-	0	111	246
fluidanimate	0	-	0	0	58	0
raytrace	0	-	-	0	117	1000
swaptions	0	-	0	0	0	0
freqmine	-	2	-	1063	225	1000
streamcluster	2	-	2	2	19	1000
vips	3	-	-	0	69	838
x264	12	-	-	1	734	1000

`Streamcluster` is implemented with two concurrency libraries: Pthread and TBB. Each variant produces two warnings. We can only confirm that one of them is a true positive. A race happens between initializing and using an array. For `x264`, our tool produces twelve false warnings.

The right half of Table 4 lists the number of warnings produced by the library-dependent tools. In case of freqmine, which is implemented in OpenMP, Intel TC produces 1063 false warnings, Helgrind 225, and DRD more than 1000 (DRD shows the first 1000 warnings and suppresses the rest). This is because these tools do not support OpenMP. But even in the Pthread samples, the number of warnings is often significantly higher. For instance, in the case of `raytrace`, DRD and Helgrind can not identify synchronization primitives within a C++ wrapper. ad hoc synchronization is also a reason for false positives.

Three packages, i.e. `dedup`, `raytrace` and `facesim` produce no warnings with Intel TC, but show extremely high memory consumption. In eight cases, Helgrind$^+$ and Intel TC produce identical results. In three cases, i.e., `bodytrack`, `canneal`, and `freqmine`, Helgrind$^+$ generates fewer false positives. `X264` might not have been instrumented properly by Intel TC, since it shows the same execution time and memory consumption before and after instrumentation.

In summary, even though Helgrind$^+$ does not rely on information about synchronization primitives, it is capable of identifying a broad range of synchronization operations in real applications (including ad hoc and implicit synchronizations). Detecting these operations is essential for any race detector, because otherwise a large number of false warnings is produced. Thus, Helgrind$^+$ can continue to be used profitably when switching concurrency and synchronization libraries.

4.4 Performance

We measured memory and runtime requirements of Helgrind$^+$ on the PARSEC benchmark suite. The values reported are averages over five executions with two threads. Because of the large overhead, we used simulation inputs and applied the simsmall for all benchmarks except for streamcluster and swaptions. For the latter two we used simmedium, as the runtime with simsmall was too short. PARSEC inputs use a combination of linear and complex scaling to derive the simulation input sets from native inputs. For this reason, the differences between native inputs and simulation inputs are relatively small and simulation inputs are suitable for performance measurements [21].

Generally, memory consumption is high. The race detection algorithm uses a large amount of shadow memory. We are planning to optimize memory consumption and compress shadow values using techniques suggested by M. Kim*et al.* [32]. For finding synchronization constructs, large code blocks (super blocks) should be disassembled and processed. This process can affect memory consumption, especially if the number of analyzed basic blocks during spinning loop detection increases. Differentiating lock operations from signals does not require much memory.

Fig. 9a depicts average memory consumption We measured the memory usage of instrumented code by Helgrind$^+$ and other tools. Out of 13 applications, twelve use Pthread and one (`freqmine`) uses OpenMP. Intel TC and Helgrind$^+$ typically use more memory than the other tools. In some case, e.g., `facsim` or `raytrace`, Intel TC shows drastically higher memory consumption. However, in most cases the memory overhead of Helgrind$^+$ is less than that of Intel TC, so that applications with significant memory requirements are testable. Helgrind and DRD require little memory due to their simple algorithms.

The normalized time measurements are shown in Fig. 9b (Helgrind$^+$ is considered as the reference with the unit normalized value). The instrumented code slows applications down by a factor of ten to more than 100. The overhead for spinning loop detection is substantial, while distinguishing locks from signals does not cost significantly. There is small overhead of Helgrind$^+$ over Intel TC for some applications. `Facsim`, `canneal` and `raytrace` slow down Helgrind$^+$ significantly. DRD runs slowly on `fluidanimate` and `dedup` and fast on `facsim` (less than 0.1 of the normalized unit). In other cases, the detectors deliver slightly different execution times.

Overall, time and space overheads are bearable for real world applications. For practical uses, overhead reductions are necessary and possible.

5 Related Work

A number of tools for concurrency bug detection have been developed [33], [34], [35], and [36]. These tools

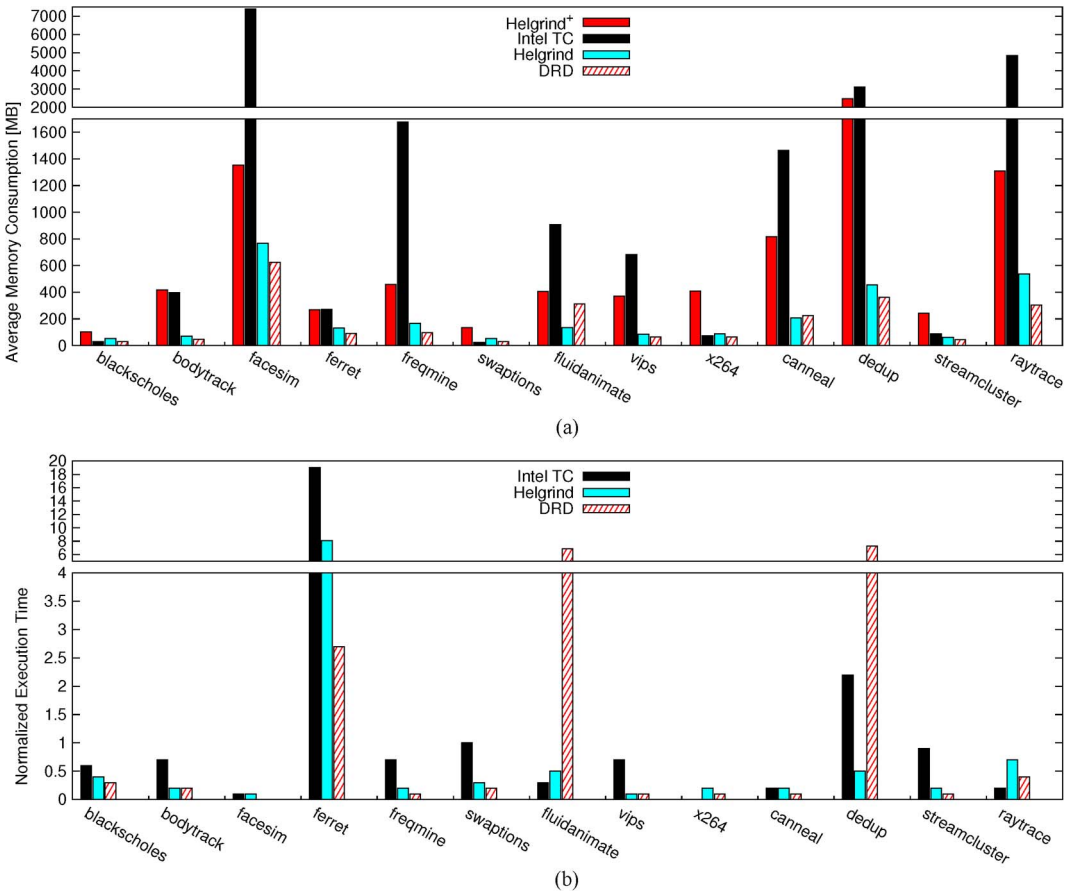

Fig. 9. Memory consumption and execution time on PARSEC. (a) Memory consumption. (b) Normalized execution times having $\mathrm{Helgrind}^{+}$ as the reference.

typically detect the synchronization constructs defined by a single, specific concurrency library. They do not work with other libraries and also fail to detect programmer-defined, ad hoc synchronization constructs. Our approach is not dependent on a specific library and can identify ad hoc synchronization as well as high-level and implicit synchronization constructs.

A few studies discuss high-level synchronization errors, i.e., synchronization anomalies that need a higher abstraction level to detect them. Arthoet *al.* [37] provide a definition for high-level data races and discuss non-atomic protection faults. Reference [38] extends the definition and discusses two types of high-level error scenarios with a static framework for detecting situations where such synchronization anomalies can manifest themselves. Our own research [39] detects data races on correlated variables but has high overhead in space and time.

Some recent work deals with ad hoc synchronization. The approach by Xiong *et al.* [10] requires automatic source code annotation of ad hoc synchronization and uses static analysis. This method suffers from the general drawbacks of static analysis, e.g., state explosion and expensive alias analysis. Code annotations are intrusive and presume availability of source code. Our approach requires no annotations and works on object code. T. Li *et al.* [40] propose special hardware for detecting spinning loops dynamically. To be detectable, the loops have to be simple, i.e., have a single exit that is controlled directly by a synchronization variable. We deal with complex loops spanning multiple blocks. Tian *et al.* [15] identifies spinning loops in software. Tian *et al.* use a simple heuristic: Any loop with a control variable that does not change for three iterations is considered a spinning loop. However, this approach may generate false positives if the spinning loop is executed less than three times or not at all. In ad hoc synchronization, programmers expect their programs to actually wait in these loops rarely. Furthermore, the method may falsely identify loops as spinning. Consequently, ordering relations are falsely assumed to exist which masks races.

6 Conclusion

An approach for detecting data races independent of concurrency libraries was presented. The approach dynamically identifies synchronization constructs based on a characteristic code pattern. By examining diverse benchmarks, among them thirteen full applications, we demonstrated that our approach is able to find data races in programs using a wide range of concurrency libraries. Various synchronization primitives as well as ad hoc synchronization constructs were detected, thereby improving the accuracy of the race detector. With this approach, race detectors do not need to be adapted to evolving concurrency libraries. There is ample opportunity to reduce space and time overhead of the race detector, for

example by encoding shadow memory in a more space-efficient way. Finding races involving correlated variables is another direction for future research.

ACKNOWLEDGMENT

The authors thank C. Schmaltz from Karlsruhe Institute of Technology for his programming support and Felix Wolf from RWTH Aachen University for providing comments.

REFERENCES

[1] M. Frigo, C.E. Leiserson, and K.H. Randall. (1998, May). The Implementation of the Cilk-5 Multithreaded Language. *ACM SIGPLAN Notices* [Online]. *33(5)*, pp. 212-223. Available: http://doi.acm.org/10.1145/277652.277725

[2] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann, 2001.

[3] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L.P. Chew, ''Optimistic Parallelism Requires Abstractions,'' *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 211-222, June 2007.

[4] J.E. Stone, D. Gohara, and G. Shi, ''Opencl: A Parallel Programming Standard for Heterogeneous Computing Systems,'' *Comput. Sci. Eng.*, vol. 12, no. 3, pp. 66-73, May/June 2010.

[5] M.I. Gordon, W. Thies, M. Karczmarek, J. Lin, A.S. Meli, A.A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, ''A Stream Compiler for Communication-Exposed Architectures,'' *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, pp. 291-303, Dec. 2002.

[6] V.A. Saraswat, V. Sarkar, and C. von Praun, ''X10: Concurrent Programming for Modern Architectures,'' in *Proc. 12th ACM SIGPLAN Symp. PPoPP*, 2007, pp. 271-271.

[7] C. Pheatt. (1998, May). Intel Threading Building Blocks. *J. Comput. Sci. Coll.* [Online]. *23(4)*, p. 298. Available: http://dl.acm.org/citation.cfm?id=1352079.1352134

[8] F. Otto, V. Pankratius, and W.F. Tichy, ''High-Level Multicore Programming With XJava,'' in *Proc. 31st ICSE-Companion Vol.*, May 2009.

[9] M. Harris, ''Many-Core GPU Computing with NVIDIA CUDA,'' in *Proc. 22nd Annu. ICS*, 2008, p. 1.

[10] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, ''Ad Hoc Synchronization Considered Harmful,'' in *Proc. 9th USENIX Conf. OSDI*, 2010, pp. 1-8.

[11] A. Jannesari, K. Bao, V. Pankratius, and W.F. Tichy, ''Helgrind+: An Efficient Dynamic Race Detector,'' in *Proc. IEEE IPDPS*, 2009, pp. 1-13.

[12] A. Jannesari and W. Tichy, ''Identifying Ad-Hoc Synchronization for Enhanced Race Detection,'' in *Proc. IEEE IPDPS*, Apr. 2010, pp. 1-10.

[13] S.V. Adve, A.L. Cox, S. Dwarkadas, H. Dwarkadas, and W. Zwaenepoel, ''Replacing Locks by Higher-Level Primitives,'' Dept. Comput. Sci., Rice Univ., Houston, TX, USA, Tech. Rep., 1994.

[14] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar, ''Healing Data Races On-The-Fly,'' in *Proc. ACM Workshop PADTAD*, 2007, pp. 54-64.

[15] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam, ''Dynamic Recognition of Synchronization Operations for Improved Data Race Detection,'' in *Proc. ISSTA*, 2008, pp. 143-154.

[16] *Helgrind: A Data-Race Detector*, Valgrind-Project, 2009. [Online]. Available: http://valgrind.org/docs/manual/hg-manual.html

[17] *Drd: A Thread Error Detector*, Valgrind-Project, 2009. [Online]. Available: http://valgrind.org/docs/manual/drd-manual.html

[18] P. Magnusson, A. Landin, and E. Hagersten, ''Queue Locks on Cache Coherent Multiprocessors,'' in *Proc. 8th Int'l Symp. Parallel Process.*, 1994, pp. 165-171.

[19] 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, Feb. 1991.

[20] R. Gupta, ''The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors,'' *SIGARCH Comput. Architect. News*, vol. 17, no. 2, pp. 54-63, Apr. 1989.

[21] C. Bienia and K. Li, ''Parsec 2.0: A New Benchmark Suite for Chip-Multiprocessors,'' in *Proc. 5th Annu. Workshop Model., Benchmarking Simul.*, June 2009, pp. 1-9.

[22] C. Bienia, S. Kumar, J.P. Singh, and K. Li, ''The Parsec Benchmark Suite: Characterization and Architectural Implications,'' Princeton Univ., Princeton, NJ, USA, Tech. Rep., Jan. 2008.

[23] N. Nethercote and J. Seward, ''Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,'' *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89-100, June 2007.

[24] N. Nethercote, ''Dynamic Binary Analysis and Instrumentation,'' Ph.D. dissertation, Comput. Lab., Univ. Cambridge, Cambridge, U.K., 2004.

[25] N. Nethercote and J. Seward, ''How to Shadow Every Byte of Memory Used by a Program,'' in *Proc. 3rd Int'l ACM SIGPLAN/SIGOPS Conf. VEE*, 2007, pp. 65-74.

[26] A. Jannesari and W.F. Tichy, ''On-The-Fly Race Detection in Multi-Threaded Programs,'' in *Proc. 6th Workshop PADTAD*, 2008, pp. 1-10.

[27] U. Banerjee, B. Bliss, Z. Ma, and P. Petersen, ''Unraveling Data Race Detection in the Intel Thread Checker,'' in *Proc. STMCS*, 2006, pp. 69-78.

[28] *Data-Race-Test: Test Suite for Helgrind, a Data Race Detector*, Valgrind-Project, 2008. [Online]. Available: http://code.google.com/p/data-race-test/

[29] M. Gebhart, J. Hestness, E. Fatehi, P. Gratz, and S.W. Keckler, ''Running Parsec 2.1 on M5,'' Comput. Eng., Univ. Texas Austin, Austin, TX, USA, pp. 1-20. [Online]. Available: http://www.cs.utexas.edu/parsec-m5/TR-09-32.pdf

[30] S. Lu, S. Park, E. Seo, and Y. Zhou, ''Learning From Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics,'' in *Proc. 13th Int'l Conf. ASPLOS*, 2008, pp. 329-339.

[31] *Glib Reference Manual*, G.D. Library, 2008. [Online]. Available: http://library.gnome.org/devel/glib/

[32] M. Kim, H. Kim, and C.-K. Luk, ''SD3: A Scalable Approach to Dynamic Data-Dependence Profiling,'' in *Proc. 43rd Annual IEEE/ACM Int'l Symp. MICRO*, 2010, pp. 535-546.

[33] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, ''Applications of Synchronization Coverage,'' in *Proc. 10th ACM SIGPLAN Symp. PPoPP*, 2005, pp. 206-212.

[34] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu, ''Finding and Reproducing Heisenbugs in Concurrent Programs,'' in *Proc. 8th USENIX Conf. OSDI*, 2008, pp. 267-280.

[35] P. Sack, B.E. Bliss, Z. Ma, P. Petersen, and J. Torrellas, ''Accurate and Efficient Filtering for the Intel Thread Checker Race Detector,'' in *Proc. 1st Workshop ASID*, 2006, pp. 34-41.

[36] C. Flanagan and S.N. Freund, ''Fasttrack: Efficient and Precise Dynamic Race Detection,'' in *Proc. ACM SIGPLAN Conf. PLDI*, 2009, pp. 121-133.

[37] C. Artho, K. Havelund, A. Biere, and A. Biere, ''High-Level Data Races,'' *J. Softw. Test., Verification Rel.*, vol. 13, no. 4, pp. 207-227, Dec. 2003.

[38] S. Raza, S. Franke, and E. Ploedereder, ''Detecting High-Level Synchronization Errors in Parallel Programs,'' in *Proc. Rel. Softw. Technol.—Ada-Europe*, vol. LNCS 6652, A. Romanovsky and T. Vardanega, Eds., 2011, vol. LNCS 6652, pp. 17-30, Springer-Verlag; Berlin, Germany.

[39] A. Jannesari, M. Westphal-Furuya, and W.F. Tichy, ''Dynamic Data Race Detection for Correlated Variables,'' in *Proc. 11th ICA3PP*, 2011, vol. 1, pp. 14-26, Springer-Verlag: Berlin, Germany.

[40] T. Li, A.R. Lebeck, and D.J. Sorin, ''Spin Detection Hardware for Improved Management of Multithreaded Systems,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 6, pp. 508-521, June 2006.

Ali Jannesari received the PhD degree in computer science from Karlsruhe Institute of Technology (formerly University of Karlsruhe), Karlsruhe, Germany. He is the Head of the multicore programming group at the German Research School for Simulation Sciences and RWTH Aachen University in Germany. His research interest is mainly focused on software engineering for multicore systems including automated testing and debugging of parallel programs, parallelism discovery and parallelization methods, and parallel programming models. Performing empirical studies towards the challenges that multicore developers are facing is another major interest of his. He is a member of the IEEE Computer Society, the ACM, and the German Computer Science Society.

Walter F. Tichy received the MS and PhD degrees in computer science from Carnegie Mellon University, in 1976 and 1980, respectively. He was a Professor of Software Engineering at the Karlsruhe Institute of Technology, Germany, since 1986. Previously, he was Senior Scientist at Carnegie Group, Inc., in Pittsburgh, Pennsylvania and served six years on the faculty of Computer Science at Purdue University in West Lafayette, Indiana. His primary research interests are software engineering and parallelism. He is currently concentrating on empirical software engineering, tools and languages for multicore computers, and making programming more accessible by using natural language for programming. He is director at the Forschungszentrum Informatik, a technology transfer institute in Karlsruhe. He is co-founder of ParTec, a company specializing in cluster computing. Dr. Tichy is a fellow of the ACM and a member of GI and the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.