Int J Parallel Prog (2015) 43:656–678
DOI 10.1007/s10766-014-0313-x

Detection of High-Level Synchronization Anomalies in Parallel Programs

Ali Jannesari

Received: 20 December 2013 / Accepted: 22 May 2014 / Published online: 31 May 2014

Abstract In parallel programs concurrency bugs are often caused by unsynchronized accesses to shared memory locations, which are called *data races*. In order to support programmers in writing correct parallel programs, it is therefore highly desired to have tools on hand that automatically detect such data races. Today, most of these tools only consider unsynchronized read and write operations on a single memory location. Concurrency bugs that involve multiple accesses on a set of correlated variables may be completely missed. Tools may overwhelm programmers with data races on various memory locations, without noticing that the locations are correlated. In this paper, we propose a novel approach to data race detection that automatically infers sets of correlated variables and logical operations by analyzing data and control dependencies. We develop an algorithm that is inspired by lockset analysis and combine it with happens-before analysis to provide the first *hybrid, dynamic race detector for correlated variables*. We implemented our approach on top of the Valgrind, a framework for dynamic binary instrumentation. Our evaluation confirmed that we can catch data races missed by existing detectors and provide additional information for effective bug fixing.

Keywords Data race detection · Parallel programs · Dynamic analysis · Correlated variables · High-level data races

A. Jannesari (✉)
German Research School for Simulation Sciences, Aachen, Germany
e-mail: a.jannesari@grs-sim.de

A. Jannesari
RWTH Aachen University, Aachen, Germany

1 Introduction

As multi-core processors have become more and more ubiquitous in recent years, programmers are faced with the challenge of writing parallel programs to leverage this computing power. Yet, writing parallel programs is inherently harder than sequential ones: among other difficulties, concurrency related bugs, such as deadlocks, atomicity and order violations [1], tend to appear randomly and are troublesome to reproduce and fix—especially if several variables are involved. How can we support programmers in this tedious work and improve existing tools?

1.1 Traditional dynamic data race detection

A popular approach is to automatically detect so-called *data races*, since data races often accompany and cause concurrency bugs. A data race occurs

> when at least two threads access the same memory location with no synchronization between them[1] and at least one of these accesses is a write [2].

Take, for example, Fig. 1: two threads increment a shared variable i in parallel; because no locking enforces the required mutual exclusion, we can observe a data race on i, which indicates an atomicity violation.

Generally, approaches to data race detection can be divided into static and dynamic analysis. Each brings its own advantages and disadvantages [3]. Among dynamic methods, which we concentrate on, two main techniques have evolved over time: The lockset algorithm [4] and happens-before analysis [5]. The lockset algorithm checks if every access to a shared memory location follows a certain locking discipline. The happens-before analysis is used to distinguish parallel thread segments from ones that have a certain order enforced between them (see Sects. 3.2 and 3.4).

While the lockset algorithm is relatively independent from scheduling, the happens-before analysis is not; however, a pure lockset algorithm suffers from many false positives, because it does not support many synchronization operations. Therefore, tools like Helgrind$^+$ [6–9] and others combine these two approaches [10,11]. Such tools are called *dynamic hybrid race detectors*.

1.2 Problem Description

The 'traditional' definition of data races does not cover concurrency bugs that involve more than a single memory location or multiple read/write-operations: consider the function *scaleVector* shown in Fig. 2, where every access to the shared tuple (x, y) is protected by lock m. Although *scaleVector* is clearly intended as an atomic operation on (x, y), another thread could change x or y during the computation of max. If the other thread also protects x and y with Lock m, no data race is detected, yet *scaleVector* obviously suffers from an atomicity violation.

[1] For example by mutual exclusion using locks or by enforcement of a specific order through signal/wait.

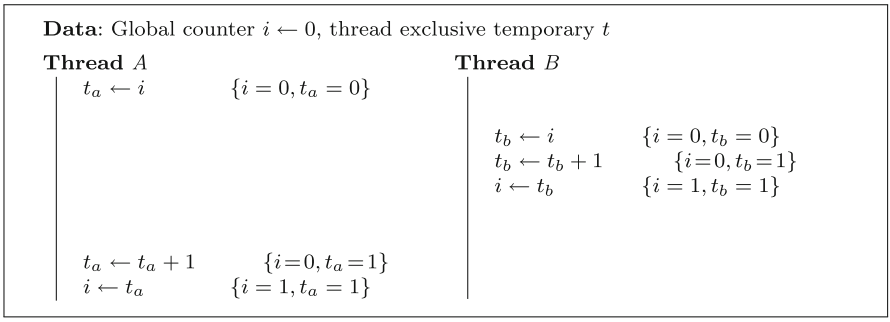


Fig. 1 Two parallel increments causing a data race

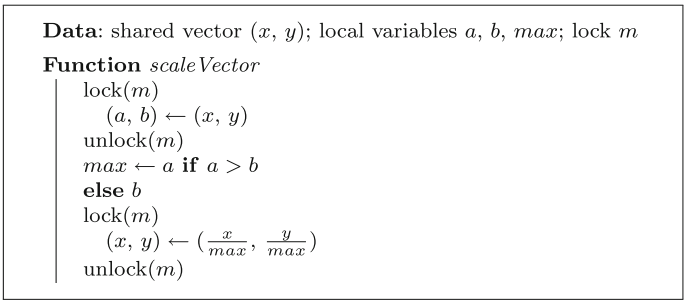


Fig. 2 Function with concurrency bug

An extensive study of concurrency bugs in [1] has revealed that a significant number (34 %) of the examined non-deadlock bugs fall into this category and are therefore not adequately addressed by existing tools. The detection of such bugs is challenging since correlations among variables usually escape traditional race detectors which are not aware of logical relationships.

In this paper, we present a new approach to data race detection that will help close this gap. Roughly speaking, we developed a new algorithm by adapting the lockset algorithm and the happens-before analysis to build a *dynamic hybrid data race detector for correlated variables and logical operations*. Our race detector is an end-to-end implementation of integrating the correlation detection algorithm into a hybrid race detection algorithm.

First and foremost, we must extend the definition of data races to capture scenarios like the one we just described. Therefore, two aspects of data races need reconsideration:

- Spatial aspect: Instead of single memory locations, we must monitor sets of correlated variables that share a semantic *consistency property*. We call such sets *correlated sets*. In the example above $s = \{x, y\}$ is one such correlated set.
- Temporal aspect: A logical operation on a correlated set that preserves its consistency property may consist of several elementary reads or writes. We call such operations *computational units*. In our example, the computational unit $u = scaleVector$ operates on s.

Using these terms, we come up with the following new definition of extended data races:

> Accesses of two parallel computational units u_1 and u_2 to the same correlated set s are called *extended data race*, if s is modified, and u_1 and u_2 are not synchronized in a manner that enforces mutual exclusion or a specific order.

Our work is based on this definition. We can divide our approach into the following three steps:

1. Automatically and dynamically infer correlated sets and computational units.
2. Adapt the lockset analysis and the happens-before analysis to provide a new algorithm for detection of extended data-races.
3. Build a *dynamic, hybrid race-detector for correlated variables*.

Note that, beside this name, concurrent logical operations on single variables without proper synchronization will also be reported.

The remainder of this paper is structured as follows: in Sect. 2, we describe other approaches that tackle similar problems. In Sect. 3, we present a technique to automatically detect correlated sets and computational units, and we develop our detector. We briefly discuss its implementation in Sect. 4 and then its evaluation in Sect. 5.

2 Related Work

There is a lot of prior work dealing with data race detection for single memory locations [8,10,12]. However, the problem of dealing with concurrency bugs involving multiple, correlated variables has only been addressed by few authors. We briefly describe three such publications:

(a) The papers [13,14] are tailored towards object oriented environments, in particular Java. It is assumed that annotated fields of a class (per instance) form an *atomic set*, while methods of the same class are *units of work* on these sets. Atomic sets are comparable to our correlated sets and units of work to computational units.

But instead of relying on synchronization operations, the authors apply the concept of *serializability* [15] as correctness criterion: for parallel units of work, they track actual sequences of interleaved read and write operations on a shared atomic set. If such a sequence is *not* equivalent to a *serial* execution of these units of work, an error is reported. This is efficiently checked by comparing the tracked sequences with a list of patterns that provide a complete characterization of *atomic set serializability*.

(b) MUVI [16] detects correlated variables by applying data mining techniques during static analysis. Although the authors' main focus is to detect inconsistent update bugs, a basic variant of the dynamic lockset algorithm is developed. It works as follows: First the lockset for every shared variable v is determined using the standard lockset algorithm. Then, it is checked whether correlated variables are protected by a common lock. Bugs caused by using different locks for correlated variables can be detected this way. However, due to the lack of a concept for logical operations, this approach fails to detect bugs like the one shown in Fig. 2.

(c) The Serializability Violation Detector (SVD) [17] uses dynamically derived control and data dependencies to detect computational units on the fly. When compu-

tational units terminate, information about them is discarded and correlations are built "from scratch"—that is, no persistent information, as with correlated sets or atomic sets, is stored. Actually, our own detection of correlated sets and computational units is based on this work. We will describe it in greater detail in Sect. 3.1. As criterion for bug detection, again, serializability is used (albeit with a more basic variant than in [13]).

Looking at the related works, it seems clear that concepts similar to correlated sets and computational units are necessary for the detection of multi-variable concurrency bugs. However, the methods to infer such constructs vary significantly, as do criteria for actual bug detection.

While using serializability can prevent benign races in some cases [13], it is inherently dependent on a concrete schedule. Also, order violation bugs may be overlooked: an example is the *use after initialization* pattern, when thread t_1 writes an initial value to v, while thread t_2 reads v—these operations are obviously serializable, but can still lead to program crashes when executed in the wrong order, e.g. if v is a pointer type.

Therefore, it is promising and desirable to bring the benefits of hybrid race detection to the domain of multi-variable concurrency bugs.

3 Race Detection for Correlated Variables

3.1 Inferring Correlated Sets and Computational Units

A prerequisite for detecting extended data races is to dynamically infer correlated sets and computational units. In related work, we have seen several solutions to this problem. Since we aim to develop a method without user intervention, we do not rely on source annotations. Instead, we infer correlated sets and computational units automatically. Our approach is therefore based on the *region hypothesis* [17] for computational units:

- All operations of a computational unit are related through either true data dependencies (read after write) or control dependencies.
- Computational units follow the 'read compute write' pattern: a program state is first read from shared memory, the new state is computed using thread exclusive memory and finally written back to shared memory. Therefore, there are no true data dependencies on *shared* memory locations *within* computational units.

Additionally, we infer correlated sets using the same heuristic: *All memory locations read or written within a computational unit, form a correlated set.*

Based on these criteria, both computational units and correlated sets can be computed fully automatically using the following online algorithm:

1. Initially, each dynamic operation (instruction) forms its own computational unit and each memory location its own correlated set.
2. When an operation op_1 is executed:
   - We *merge* the computational units of all dynamic operations that op_1 depends on through a control dependency.

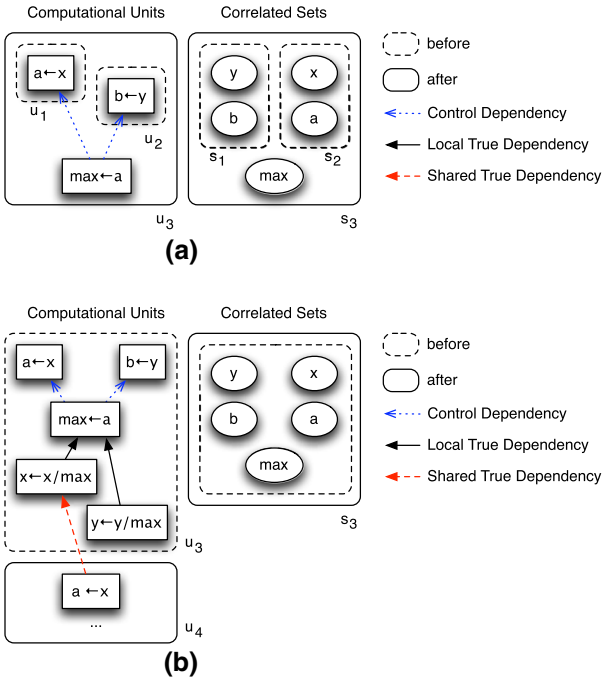


Fig. 3 Region hypothesis applied to the function *scaleVector* of Fig. 2. **a** Merging computational units and correlated sets due to control dependency. **b** Ending a computational unit due to shared true dependency

- We *merge* the computational units of all dynamic operations that op_1 depends on through a true data dependency.[2] However, if op_1 is true data dependent on op_2 through a *shared* memory location, op_2's computational unit is *not* merged, but instead marked as *closed*.
- We *merge* the correlated sets of all memory locations that op_1 reads, and the correlated sets of all memory locations that op_1 is control dependent on. The merged correlated set is also assigned to the variable written by op_1 (eventually overwriting its old correlated set).

In Fig. 3 this algorithm is applied to the function *scaleVector*. Subfigure (a) shows the situation before and after executing $op = max \leftarrow a$: first, the assignments $a \leftarrow x$ and $b \leftarrow y$ form their own computational units u_1 and u_2, and two correlated sets $s_1 = \{a, x\}$ and $s_2 = \{b, y\}$ could be inferred during u_1 and u_2, respectively. After executing op, because of op's control dependencies, all operations are merged into a single computational unit u_3 and all memory locations to a single correlated set s_3. In Subfigure (b), we can see the situation before and after executing *scaleVector* a second time: all operations within this function are related through either control dependencies or true data dependencies; therefore *scaleVector* is recognized as computational unit

[2] An operation op_1 has a true data dependency on an operation op_2, if op_1 reads a value that was last written by op_2.

u_3. Furthermore, when executed a second time, a shared true dependency on x is observed, ending u_3 and starting u_4.

As one can see, it is possible for correlated sets and computational units to contain both shared and exclusive parts alike. While it is mostly the shared parts, which finally matter for data race detection, we must also track thread exclusive computations and memory locations for two main reasons: first, because resources that are now considered exclusive may become shared later on. Second, because correlations between shared resources are often established through exclusive intermediate values—as we've seen in the example above.

For the *scaleVector* function, the region hypothesis obviously led to correct results. However, because of its heuristic nature, this must not always be the case: in fact, experiments in [17] showed that the region hypothesis holds on the most common paths of 14 examined atomic regions but fails on some rare paths. One common source of errors are shared true dependencies within atomic regions. In these cases, the region hypothesis cuts computational units too early. This limitation could be mitigated by exploiting information about program structure: shared true dependencies are allowed within computational units, if both operations occur within the same function body (similar to the criterion used for units of work in [14] and [13]).

On the other hand, an 'early `if`' could cause the whole program to be interpreted as a single computational unit. We therefore limited the influence of control dependencies on merging to function scope. In [17] control dependencies were completely ignored.

We will further discuss the impact of detecting computational units and correlated sets and show alternatives to the region hypothesis in the conclusion of this paper.

One final aspect that has yet to be clarified is how exactly one can detect control dependencies during dynamic analysis. To do so, we use the idea of *reconvergence points* introduced in [18]. When encountering a conditional jump, the jump target is probed to determine the type of control flow construct: For example, if the target is preceded by an unconditional forward jump, we have encountered an `if-else` construct; the reconvergence point is the target of the unconditional jump. On the other hand, if there is *no* jump, we have encountered an `if` construct. Figure 4 illustrates these two cases.

However, in contrast to [18] and [17] that are limited to `if` and `if-else` constructs, we're also able to identify loops. This is possible, because of using our loop

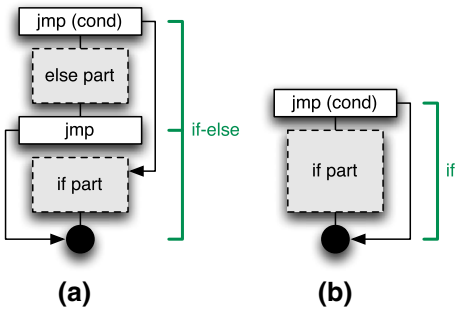


Fig. 4 Finding the reconvergence point (*black filled circle*). **a** `if-else` construct. **b** `if` construct

detection patterns introduced in [7,8]. Furthermore we support non-local jumps caused by `break`, `continue` or `return`.

3.2 Adapting the Lockset Algorithm

As mentioned, the original lockset algorithm checks if every access to a shared resource obeys a certain locking discipline that ensures mutual exclusion. Let us briefly review the original algorithm for better understanding before we discuss our extensions.

The lockset algorithm enforces that every shared memory location v is protected by a non empty set of locks in the sense that all of these locks are held whenever a thread accesses v. Since it is at first unclear which memory location is protected by which locks, we dynamically gather this information during the program's execution: For each Thread t we store L_t, the set of all locks currently held by t. We call L_t the *lockset* of t. Furthermore, we maintain a *candidate set* of locks C_v for each memory location v. Initially, C_v is assumed to consist of all locks and than successively refined on each access to v. The complete algorithm is shown in Fig. 5.

Obviously, this approach considers neither the spatial nor the temporal aspects which we earlier captured in form of correlated sets and computational units. Therefore, to extend this algorithm for our needs, we must somehow substitute L_t and C_v with equivalents for computational units and correlated sets. Let us look at how we can redefine locksets first.

Generally spoken, a computational unit u consists of three parts (see Fig. 6):

– $excl_1(u)$: u accesses only exclusive variables

Function *initialize*
  ∀ variables $v : C_v \leftarrow \{\text{all locks}\}$
  ∀ threads $t : L_t \leftarrow \emptyset$

On Event *Thread t acquires lock m*
  $L_t \leftarrow L_t \cup \{m\}$

On Event *Thread t releases lock m*
  $L_t \leftarrow L_t \setminus \{m\}$

On Event *Thread t accesses v*
  $C_v \leftarrow C_v \cap L_t$ `{refine` C_v`}`
  if $C_v = \emptyset$ **then** `{`v `unprotected}`
    issue warning
  end

Fig. 5 Basic lockset algorithm

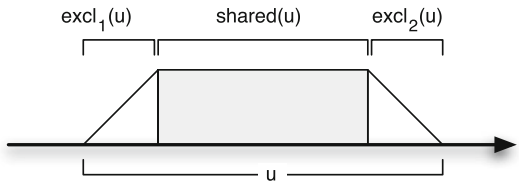


Fig. 6 Different parts of a computational unit u

```
On Event Computational Unit u accesses v ∈ s
    C_s ← C_s ∩ L_u                          {refine C_s}
    if C_s = ∅ then                                        {s unprotected}
        issue warning
    end
```

Fig. 7 Adapted lockset algorithm

– $shared(u)$: u accesses shared and exclusive variables alike
– $excl_2(u)$: u accesses only exclusive variables

Each of these parts can also be empty. Based on this observation we define:

$$L_u := \begin{cases} held_u, & shared(u) \neq \emptyset \\ all\_locks, & \text{otherwise} \end{cases}$$

where

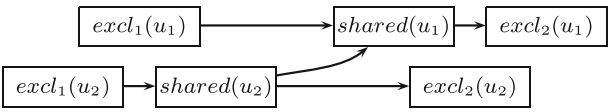
$$held_u := \{\text{Lock } m \mid m \text{ held throughout } shared(u)\}$$

This means: L_u consists of the locks held throughout $shared(u)$, if $shared(u)$ is not empty (denoted by $held_u$ above); otherwise L_u equals the set of all locks.

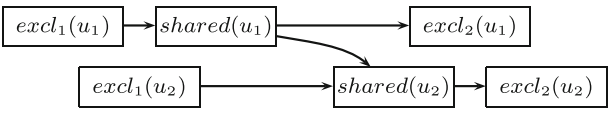
However, because of our tool's intended dynamic nature, we cannot know in advance exactly when $shared(u)$ starts and ends—actually computing L_u is therefore a problem itself, which we discuss in Sect. 3.3. For the moment, we assume that we have all required knowledge available in advance. To complete the adapted lockset algorithm, we can now simply replace L_t with L_u and C_v with C_s (C_s denotes the candidate set of a correlated set s and is computed analogously to C_v), yielding the algorithm shown in Fig. 7.

For the discussion of our locking policy, we assume that a correlated set s is accessed by u_1 and u_2 in parallel, with $L_{u_1} \cap L_{u_2} \neq \emptyset$. Obviously $shared(u_1)$ and $shared(u_2)$ cannot overlap, so that only the following two general cases of interleaving are possible:

– Either $shared(u_2)$ precedes $shared(u_1)$:

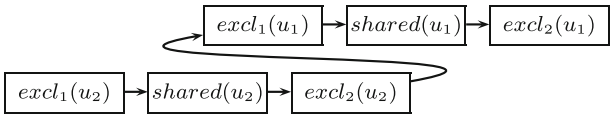


– Or $shared(u_1)$ precedes $shared(u_2)$:

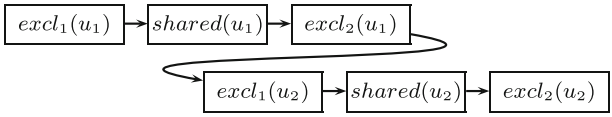


In the diagrams above, arrows denote the temporal ordering between individual parts (we will get to know this ordering as happens-before relation in Sect. 3.4). Now,

since all exclusive parts solely operate on exclusive variables and do not interfere with parallel computations, the first case is always equivalent to:



while the second is always equivalent to:



Therefore, our initial assumption implies that all possible interleavings of u_1 and u_2 must be equivalent to either $u_2 \rightarrow u_1$ or $u_1 \rightarrow u_2$ and are thus *serializable*. If we generalize this observation for s with $C_s \neq \emptyset$ and an arbitrary number of u_i accessing s, all u_i are serializable. This guarantees that there is no data race on s.

Note that there are other possibilities to define L_u. In particular, we could have defined L_u to consist of all locks that are held from the very beginning to the very end of u. However, this definition would yield many false positives, since $excl_1(u)$ and $excl_2(u)$ do not need to be protected by locks.

3.3 Calculating a Computational Unit's Lockset

In the previous section, we assumed that we are endowed with sufficient a priori knowledge to compute L_u. That is, knowledge about which memory accesses constitute $shared(u)$. In dynamic program analysis, however, $shared(u)$ can repeatedly change for various reasons:

1. After the last access to a shared memory location, we must assume that all further exclusive read/write-operations are part of $excl_2(u)$. This assumption must be revised, if another access to a shared memory location follows.
2. Upon merging two computational units u_1 and u_2 to u, their shared parts must be combined, yielding $shared(u)$. $shared(u)$ may now contain accesses that are neither part of $shared(u_1)$ nor of $shared(u_2)$ (Fig. 8a).
3. A variable v that was formerly considered exclusive, may later turn out to be actually shared. The shared part of the computational unit that accessed v earlier, must then be extended accordingly (Fig. 8b) .

We can solve these problems by introducing a new concept called *lock vector*, which is inspired by vector clocks. For a thread t, the lock vector function $\mathbf{l}_t$ is defined as follows:

$$\mathbf{l}_t : Lock \rightarrow \mathbb{N} \times \mathbb{N},\ m \mapsto (\mathbf{l}_t(m)_{\text{acq}}, \mathbf{l}_t(m)_{\text{rel}}).$$

$\mathbf{l}_t$ maps a lock m to its number of acquisitions $\mathbf{l}_t(m)_{\text{acq}}$ and releases $\mathbf{l}_t(m)_{\text{rel}}$ by thread t so far. Note that $\mathbf{l}_t(m)_{\text{acq}}$ is in general *not* equal to the number of calls to $m.lock()$

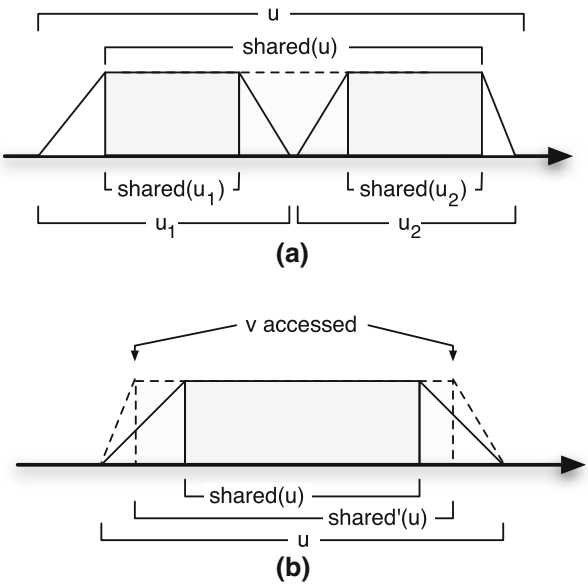


Fig. 8 Reasons for $shared(u)$ to change. **a** Merging of computational units. **b** Exclusive variable becomes shared

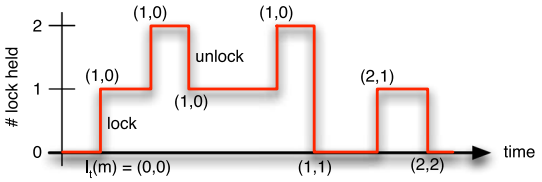


Fig. 9 Various lock operations and resulting $\mathbf{l}_t$

by t: for example, in the case of a recursive lock, $\mathbf{l}_t(m)_{\text{acq}}$ isn't further increased, if t already is in possession of m. Figure 9 shows how $\mathbf{l}_t$ changes for various calls to $lock()$ and $unlock()$.

Furthermore, we store two local copies of the lock vector $\mathbf{l}_t$ at the beginning and end of $shared(u)$, called $\mathbf{l}_{\text{fst},u}$ and $\mathbf{l}_{\text{lst},u}$. Then, we can compute L_u as follows:

$$L_u = \left\{ \text{Lock } m \,|\, \overbrace{\mathbf{l}_{\text{fst},u}(m)_{\text{acq}} - \mathbf{l}_{\text{fst},u}(m)_{\text{rel}} = 1}^{m \text{ held at the beginning of } shared(u)} \right.$$

$$\left. \wedge \underbrace{\mathbf{l}_{\text{lst},u}(m)_{\text{rel}} - \mathbf{l}_{\text{fst},u}(m)_{\text{rel}} = 0}_{m \text{ not released until the end of } shard(u)} \right\}$$

Table 1 Lockset algorithm applied to *scaleVector* (new values are marked bold)

Statement executed	$\mathbf{l}_t(m)$	Part of u	$\mathbf{l}_{\text{fst},u}(m)$	$\mathbf{l}_{\text{lst},u}(m)$	L_u	C_s
initially	(0, 0)	$excl_1$	–	–	*all*	*all*
lock(m)	**(1, 0)**	$excl_1$	–	–	*all*	*all*
$(a, b) \leftarrow (x, y)$	(1, 0)	**shared**	**(1, 0)**	**(1, 0)**	{**m**}	{**m**}
unlock(m)	**(1, 1)**	*shared*	(1, 0)	(1, 0)	{m}	{m}
$max \leftarrow a$ **if** $a > b$ **else** b	(1, 1)	**excl$_2$**	(1, 0)	(1, 0)	{m}	{m}
lock(m)	**(2, 1)**	$excl_2$	(1, 0)	(1, 0)	{m}	{m}
$(x, y) \leftarrow \left(\frac{x}{max}, \frac{y}{max}\right)$	(2, 1)	**shared**	(1, 0)	**(2, 1)**	$\emptyset$	$\emptyset$

If another shared variable is accessed by u, we can easily update $\mathbf{l}_{\text{lst},u}$ and L_u; if two computational units u_1 and u_2 are merged to u, we set

$$\mathbf{l}_{\text{fst},u} = \min(\mathbf{l}_{\text{fst},u_1}, \mathbf{l}_{\text{fst},u_2}) \quad \text{and}$$
$$\mathbf{l}_{\text{lst},u} = \max(\mathbf{l}_{\text{lst},u_1}, \mathbf{l}_{\text{lst},u_2})$$

using element-wise comparison and recompute L_u. The problems caused by the above points 1) and 2) are therefore solved, yet problem 3) still remains. To solve it as well, we must store additional copies $\mathbf{l}_{\text{fst},v}$ and $\mathbf{l}_{\text{lst},v}$ of $\mathbf{l}_t$ for an exclusive variable v the first and last time it is accessed by u. If v later becomes shared, u's lock vectors are then updated as follows:

$$\mathbf{l}_{\text{fst},u} = \min(\mathbf{l}_{\text{fst},v}, \mathbf{l}_{\text{fst},u}) \quad \text{and}$$
$$\mathbf{l}_{\text{lst},u} = \max(\mathbf{l}_{\text{lst},v}, \mathbf{l}_{\text{lst},u})$$

and L_u is recomputed.

This concludes our description of the adapted lockset algorithm. Before we continue to explain how to integrate temporal ordering, we will exemplarily apply it to the function *scaleVector* in Fig. 2. The result is shown in Table 1: initially, there are neither acquisitions nor releases of lock m, while u is in $excl_1$ and L_u, therefore, contains all locks by definition. As we encounter the first acquisition of m, we increase $\mathbf{l}_t$ for m to (1, 0). With the assignment in the third row of Table 1, u accesses the shared resources x and y: u switches to its *shared* part and makes local copies of $\mathbf{l}_t$. When writing to max, we can assume that u has reached $excl_2$. However, this assumption must be revised on the next access to x and y. Since $\mathbf{l}_t(m)$ has changed to (2, 1) in between, $\mathbf{l}_{\text{lst},u}(m)$ also takes this new value, causing L_u and finally C_s to become empty. The detection of an extended data race will be reported at this point.

3.4 Happens-Before Analysis: Hybrid Race Detector

A pure lockset based race detector fails to recognize synchronizations like signal/wait, fork/join or barriers, and will therefore produce many false positives. As a hybrid race

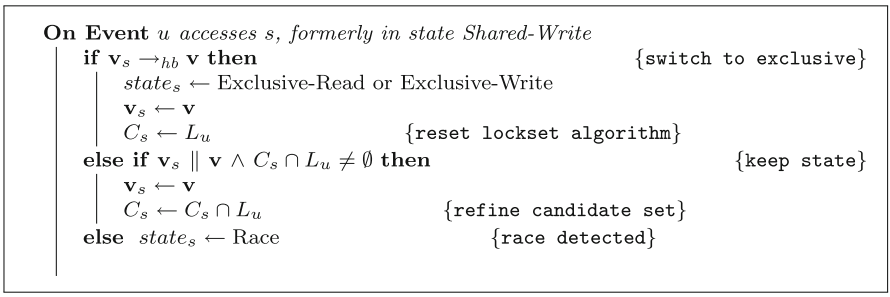


Fig. 10 Steps performed in shared-write state

detector, our approach therefore combines the lockset algorithm with the *happens-before relation* $\rightarrow_{\text{hb}}$ that tracks the temporal and causal ordering of events. For two such events e_1 and e_2 we define:

$$e_1 \rightarrow_{\text{hb}} e_2 \;:\Leftrightarrow\; e_1 \text{ precedes } e_2 \text{ temporally/causally}$$
$$e_1 \| e_2 \;:\Leftrightarrow\; e_1 \not\rightarrow_{\text{hb}} e_2 \wedge e_2 \not\rightarrow_{\text{hb}} e_1$$

The happens-before relation itself is implemented by *vector clocks*: a global timestamp vector, called $\mathbf{v}$, is tracked using thread local event counters that are exchanged on synchronization events [19]. For two events e_1 and e_2 that take place at times $\mathbf{v}_1$ and $\mathbf{v}_2$, we then have $e_1 \rightarrow_{\text{hb}} e_2 \Leftrightarrow \mathbf{v}_1 < \mathbf{v}_2$ with element-wise comparison.

To combine the lockset algorithm and $\rightarrow_{\text{hb}}$, we use a state machine based on our race detector Helgrind$^+$ [8]. This state machine can distinguish between parallel and ordered accesses to a correlated set s as follows: whenever s is accessed, a copy of $\mathbf{v}$, called $\mathbf{v}_s$, is stored with s. Any subsequent access to s at time $\mathbf{v}'$ is then parallel to the first one, iff $\mathbf{v}_s \not< \mathbf{v}'$.

The full state machine is depicted in Fig. 11. Transition labeled with '$\rightarrow$' and '$\|$' denote ordered and parallel accesses, respectively. It consists of the following six states:

Not-Accessed The correlated set s was not yet accessed; on the first access, we enter one of the states Exclusive-Read or Exclusive-Write.

Exclusive-Read and Exclusive-Write s is in exclusive possession of a single computational unit u (and has, in case of Exclusive-Write, been modified by u). We stay in these exclusive states as long as subsequent accesses are ordered by $\rightarrow_{\text{hb}}$, in which case C_s is overwritten with the latest L_u. Should any parallel access come along, we refine C_s according to the lockset algorithm and enter one of states Shared-Read, Shared-Write or Race, depending on the kind of access and C_s.

Shared-Read and Shared-Write Computational units from different threads, unordered by $\rightarrow_{\text{hb}}$, have accessed s. We need to refine C_s on each further access to detect possible violations of our locking policy. In case of an ordered access, we can go back to one of the exclusive states and reset the lockset algorithm (see, as an example, Fig. 10 for the complete algorithm in the Shared-Write state).

Race a race has happened on s. We won't leave this state, to prevent our tool from reporting the same race over and over again (Fig. 11).

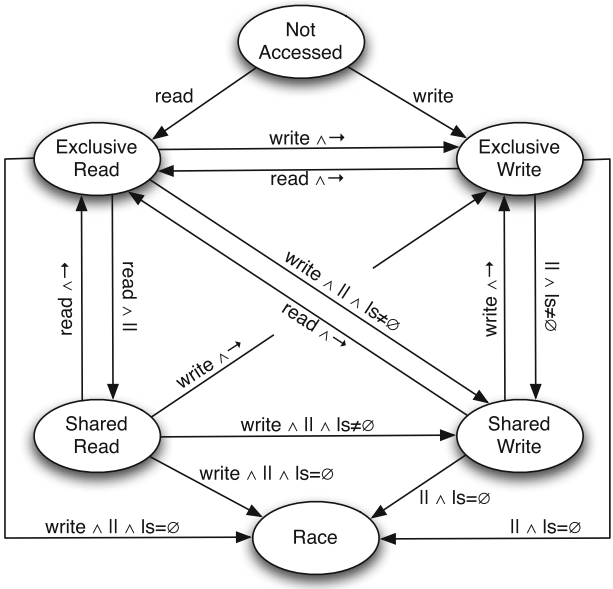


Fig. 11 State machine for correlated sets

4 Implementation

Our implementation borrows from Helgrind$^+$ [8], which in turn is based on the Valgrind framework [20,21]. The general principle behind Valgrind is called *disassemble-instrument-resynthesize*:

- A program's binary is *disassembled* into a platform independent *intermediate representation* (IR).
- The IR is then handled to the *tool* for *instrumentation*: analysis code can be injected into the original program.
- The program and analysis logic is then reassembled to platform specific code and brought to execution.
- Valgrind supports multi-threading by intercepting calls to the POSIX Thread API [22] and redirecting them to its own threading implementation.

Furthermore, Valgrind sends events and callbacks to the active tool, for instance, when stack frames are opened and closed or when memory is allocated and released. Additionally, the tool is notified before and after certain library functions are executed. This includes those functions of the POSIX Thread API, making it possible to implement the lockset algorithm and the happens-before analysis.

Helgrind$^+$, among other things, provides an implementation of *shadow memory*: for every byte of memory b used by the application, there exists a mirrored 64-bit *shadow value* s_b for storing state information about the 'application byte'. Our implementation uses shadow values as follows: in s_b we store two references: the first one points to

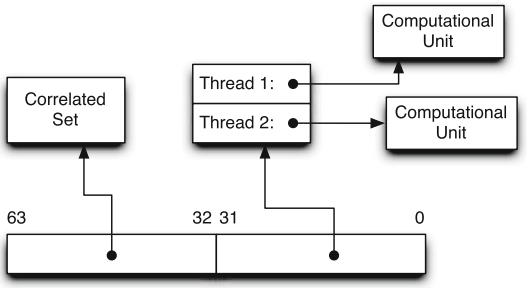


Fig. 12 Shadow value for a memory location accessed by two threads

the correlated set that b belongs to, whereas the second one points to a collection of computational units (see Fig. 12).

The latter is necessary for the following reason: we cannot actually represent computational units as sets of dynamic instructions, since such sets may grow too long, e.g. in case of loops or deep recursions. Instead, we represent a computational unit by the set of all accessed memory locations. A memory location references all non-ended computational units, which it was accessed by. Then, tracking computational units actually becomes propagating these references along the dynamic data flow graph.

5 Evaluation

The experimental evaluation is divided in three parts: first we check if our tool is able to correctly deal with basic synchronization operations. In the second part, we look at more complex examples taken from real-world applications (e.g. Apache, MySQL, etc) to check if computational units, correlated sets, and extended data races are detected. In the last part, we evaluate a set of micro-benchmarks and real-world applications to emphasize the effectiveness of our method considering number of false positives and false negatives. We compare our results to our enhanced race detector, Helgrind$^+$ [7,8][3] which serves as representative for tools that do not natively support multi-variable race detection.

5.1 Identifying Basic Synchronization

For the basic tests, shown in Table 2, we use the following notation:

- $read_A(x)$: variable x is read, but not written, by thread A.
- $write_A(x)$: x is written by thread A; the new value does not depend on x's old value, e.g. $x \leftarrow const$.
- $inc_A(x)$: x is first read and then written, depending on its former value, e.g. $x \leftarrow x + 1$.
- $\{\ldots\}_m$: segment $\{\ldots\}$ is protected by Lock m.

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

Table 2 Tests for synchronization

Test No.	Description	Expected	Detected
0	$inc_A(x) \parallel inc_B(x)$	Race	Race
1	$read_A(x) \parallel inc_B(x)$	Race	Race
2	$read_A(x) \parallel read_B(x)$	No race	No race
3	$\{inc_A(x)\}_m \parallel \{inc_B(x)\}_m$	No race	No race
4	$write_A(x) \parallel write_B(x)$	Race	$\times$ *No race*
5	$\{inc_A(x)\}_m \{inc_A(x)\}_m \parallel \{inc_B(x)\}_m$	No race	No race
6	$\{inc_A(x)\}_{m,n} \parallel \{inc_B(x)\}_m$	No race	No race
7	$\{inc_A(x)\}_m \parallel \{inc_B(x)\}_n$	Race	Race
8	$\{inc_A(x)\}_m \{inc_A(x)\}_n \parallel \{inc_B(x)\}_m$	Race	Race
9	$inc_A(x)\ {}_A{\rightarrow}_B\ inc_B(x)$	No race	No race
10	$\{inc_A(x)\}_m \parallel \{inc_B(x)\}_m\ {}_A{\rightarrow}_B\ inc_B(x)$	No race	No race
11	$\{inc_A(x)\}_m \parallel \{inc_B(x)\}_m\ {}_A{\rightarrow}_B\ read_A(x) \parallel read_B(x)$	Race	Race
12	$\{inc_A(x)\}_m \parallel \{inc_B(x)\}_m\ {}_A{\rightarrow}_B\ {}_B{\rightarrow}_A\ read_A(x) \parallel read_B(x)$	No race	No race
13	$\{inc_A(x)\}_m \parallel \{inc_B(x)\}_m\ barrier\ read_A(x) \parallel read_B(x)$	No race	No race

- ${}_A{\rightarrow}_B$: the last segment of thread A and the next segment of thread B are ordered by the happens-before relation, e.g. if A sends a signal to B.
- $\parallel$: denotes parallel segments.

Tests 0–8 show examples of unordered accesses to a shared variable x. If x is changed, it must be protected by a lock consistently to prevent data races. However, our approach fails on test 4 unlike the other tools. We look at this case more closely:

$write(x)$ overwrites x with a value that itself does not depend on x, e.g. with a constant expression or any uncorrelated variable. In such cases, our algorithm *replaces* x's old correlated set, and thus cannot detect the absence of locks (as described in point 2. of the algorithm in Sect. 3.1). Alternatively, we could have *merged* x's correlated set with the new value's correlated set. In this case, the race in test 4 would have been detected. However, such behavior would also increase the probability of over-estimating the size of correlated sets and computational units: for example, when independent computations reuse common exclusive variables, they would be considered as correlated (think of loop counters).

According to our observations, reusing exclusive variables occurs much more often than overwriting all members of a correlated set with new and independent values. The latter usually occurs only on initialization or when resetting data structures. We therefore decided to stick to the current policy, albeit further investigations may be needed.

As a representative for the remaining test cases that involve ordered accesses, we explain test 11 in more detail. The following diagram shows the order between all operations in this case:

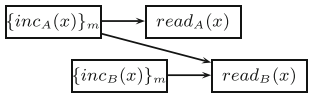


Table 3 Detected correlated sets, computational units and data races

Test No.	Description	Expected	Helgrind$^+$	Our approach	
				CUs and CSets	Race
14	ScaleVector without locks	Race	Race	✓	Race
15	ScaleVector with interrupted locks	Race	× *No race*	✓	Race
16	Normalize with consistent locking	No race	No race	✓	No race
17	Different locks	Race	× *No race*	✓	Race
18	AppendBuffer without locks	Race	× *Multiple races*	✓	Race
19	Swapping correlated variables with locks	No race	No race	✓	No race
20	Swapping uncorrelated variables with locks	No race	No race	×	× *Race*
21	Independent calculations	–	–	×	–

inc_A and inc_B are not ordered, but both are protected by a lock m; also, since inc_A and $read_B$ are ordered by ${}_A \rightarrow_B$, no data race is caused by these three instructions alone. Unlike test 10, however, there is another unprotected read: $read_A$ and inc_B are not ordered by any means, thus causing a data race. Our tool reliably detects this case.

The final two test cases basically represent the same scenario, but $read_A$ is now only parallel to $read_B$: in test 12, thread B sends a signal to A (${}_B \rightarrow_A$), and in test 13 a barrier synchronization is used. Since parallel reads are not racy by definition, no warning is issued.

5.2 Detecting Extended Data Races

The results of the second part of our evaluation focuses on the detection of correlations and extended data races shown in Table 3. The simplified codes taken from real applications are indicated as Tests 14–21 in the table.

Tests 14 and 15 simply represent the function *scaleVector* from Fig. 2. For test 14, the locks were completely omitted, whereas in test 15 the locks were kept as in Fig. 2. While Helgrind$^+$ is able to detect the locking violation in test 14, it fails in test 15: in this test, atomicity is violated by releasing and re-acquiring locks during a logical operation. However, Helgrind$^+$ is not able to detect this kind of bug, since each single access to a shared variable is properly protected by locks. The new approach detects the race and passes in test 15.

Test 16 represents another arithmetic function for vector normalization (Fig. 13). It has a more complicated dependency graph, because it uses the `sqrt()` function. Still, all correlations are detected. The normalization itself is consistently protected by a single lock, so it is data race free.

A similar scenario is represented by test 17, shown in Fig. 14: the two shared variables `mCont` and `mLen` are related through the string `s`, yet protected by different locks. Again, a typical race detector cannot detect this data race. In contrast, our

```
normalize() {
 lock(&m);
 float len = sqrt(a*a + b*b);
 a = a/len; b = b/len;
 unlock(&m);
}
```

Fig. 13 Test 16: Vector normalization

```
append(char *s) {
 lock(&m1);
 mCont = f(mCont,s);
 unlock(&m1);
}
reflow(char *s) {
 lock(&m2);
 mLen = g(mLen,s);
 unlock(&m2);
}
```

Fig. 14 Test 17: Using different locks for correlated variables

approach correctly infers the correlated set {s, mCont, mLen} and detects its empty lockset. This test also shows that our method handles function calls reliably: when f() is called, s is copied from append()'s stack frame to f()'s stack frame, making the two copies dependent. When f() ends, the return value is stored in a machine register that depends on the original s and mCont. This register value is finally moved to mCont establishing the correlation of mCont and s. Likewise, mLen is included in this correlated set.

Test 18 is shown in Fig. 15; it is part of Apache's log_config module [23], which contains a data race (now fixed). outCnt and outBuf implement a buffer for status messages. The correlation between those two variables is established through

```
void bufferAppend(char *str) {
   int len = strlen(str);
   if (len+outCnt >= BUFSIZE) {
      // flush buffer!
      outCnt = 0;
   }
   for (int i = 0; i < len; i++) {
      outBuf[outCnt+i] = str[i];
   }
   outCnt = outCnt+len;
}
int strlen(char *str) {
   int ctr = 0;
   while(str[ctr]) { ctr++; }
   return ctr;
}
```

Fig. 15 Test 18: Appending to a buffer without locks

```
1 static OBJ *list_head;
2 OBJ *dequeue_and_fill(int a, int b) {
3   OBJ *head = list_head;
4   head->a = a, head->b = b;
5   list_head = head->next;
6   return head;
7 }
```

Fig. 16 Test 21: Independent operations within a atomic region

`len`, so that `bufferAppend()` is correctly identified as a single computational unit. Since there are no locks to protect the shared resources, traditional race detectors report *multiple* data races on `outBuf` and `outCnt`. In contrast, our approach reports only a *single* data race on $s = \{\texttt{outCnt}, \texttt{outBuf}\}$. This can make it easier for the programmer to identify the root cause of the detected race and to apply a correct bug fix. Just reporting several seemingly unrelated data races on the other hand may mislead the programmer to wrap every access to a shared variable in locks, but overlook their correlation. This can be a serious problem, especially if the static distance between these accesses is bigger than in our example.

Figure 15 also shows a simple implementation of the `strlen()` function. We include it to demonstrate how control dependencies are tracked by our implementation: the incrementation of `ctr` depends on the outcome of the loop condition `str[ctr]`, leading to the correlation between `ctr` and `str`.

In tests 19 and 20, two shared variables `a` and `b` are swapped using a temporary variable:

$$\texttt{tmp} = \texttt{a}; \quad \texttt{a} = \texttt{b}; \quad \texttt{b} = \texttt{tmp};$$

The outcome depends on the previous state of the two variables `a` and `b`: swapping is correctly identified as a logical operation, when `a` and `b` were already correlated. But for independent values our approach fails for the same reason as in test 4 of the previous section: first, `tmp` inherits `a`'s correlated set, while `a`'s own correlated set is overwritten with the one of *b*. Then *b*'s set is overwritten by `tmp`'s. Effectively, `a` and `b` have now swapped their correlated sets as well. When the variables are accessed for the next time, protected by the same locks as before swapping their values, the locksets then become empty and false positives are reported.

Finally, in test 21 [17], shown in Fig. 16, a data structure consisting of semantically correlated variables is initialized, but the initialization values are independent. Inferring of correlated sets and computational units fails in such cases. This special case could be solved by considering address calculations for dependency analysis: `head->a` and `head->b` are computed by adding two fixed offsets to `head`. However, tracking address dependencies could cause over-estimation of correlated sets, since `struct`-members must not automatically be related: think, for example, of a data structure for counting incoming and outgoing data packets.

Table 4 False positives and false negatives

Program	LOC	Total races	Helgrind$^+$		Intel TC		Our approach	
			FN	FP	FN	FP	FN	FP
Micro Benchmarks	–	153	81	11	110	5	32	7
STP	1,120	15	8	13	10	2	5	10
KeyPassLib	1,240	10	5	16	5	3	1	14
PetriDish	1,070	5	4	10	4	3	3	10
Order-sim [Corrs]	480	15	11	0	14	0	3	0
Order-sim [Gaps]	372	22	19	1	22	0	11	0
Order-sim [Unsynch]	334	22	12	5	12	2	3	1
Order-sim [Synch]	360	0	0	0	0	0	0	0
Sum		242	140	56	177	15	58	42

5.3 Effectiveness

The results of the last part of our evaluation focuses on effectiveness of our method by presenting the number of false positives and false negatives shown in Table 4. We checked a set of micro-benchmarks (190 benchmarks in total) and some real-world parallel applications. The benchmarks are mostly taken from Data-race-test [24], a benchmark suite for race detectors that implements various scenarios including tricky situations that are difficult to analyze. The open source applications include PetriDish [25], the program library of KeyPass [26], SmartThreadPool (STP) [27], and different versions of Order-System Simulation [28].

Table 4 lists the programs and summarizes the actual numbers of false positives and false negatives that occurred during the race detection for all evaluated programs. In this part, we compare our results to Intel Thread Checker (Intel Inspector) [29], which is a commercial tool, and known for detecting conventional concurrency bugs, in addition to our enhanced race detector, Helgrind$^+$. Our approach has by far the lowest number of false negatives. It is effectively able to find high-level data races between correlated variables with few false positives and false negatives. The commercial tool, Intel TC, produces only 15 false positives, which is the lowest false positive rate among the tools. However, it misses 177 races, which is a high rate of false negatives. Few false negatives of our dynamic approach are race conditions that lie on unexecuted paths. Some false positives come from harmless (intentional) data races inside the programs. The detectors (including Intel TC and Helgrind$^+$) cannot distinguish between potentially harmful and intentional race conditions. Therefore, they share the encountered false positives because of intentional data races. Other reasons for false negatives and false positives produced by our approach are the scenarios categorized and discussed in previous sections. Figure 17 shows the overhead of our approach for the evaluated programs and compares it with other tools. In most cases the time and memory overheads are bearable and less than Intel TC.

In summary, our approach is capable of identifying a notorious class of data races that violates correlation between variables. Detecting these races is essential for any

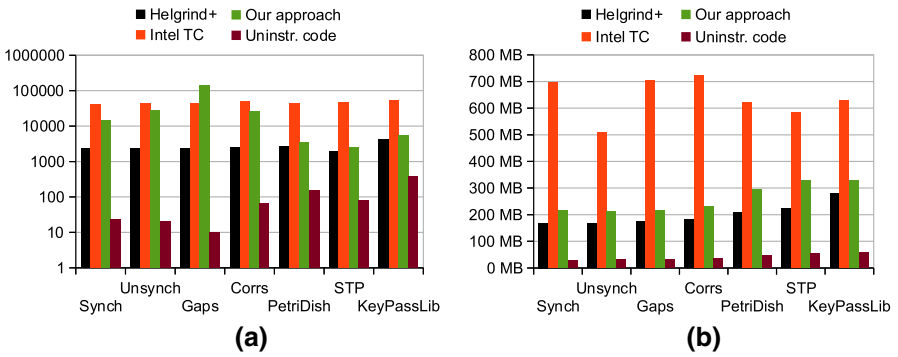


Fig. 17 Overhead of our approach compared to other tools. **a** Time overhead (ms). **b** Memory consumption (MB)

race detector, because otherwise a large number of data races are missed. We consider handling large data-intensive parallel programs by designing the shadow memory in a more efficient way to reduce the memory footprint. This could be implemented by using the signature technique [30] instead of the traditional double-layer table technique. A signature represents an unbounded set of data approximately with a bounded amount of state and is widely used in Transactional Memory. The proposed approach can be adapted to be used profitably for data-intensive parallel computing platforms using the signature technique.

6 Conclusion and Future Work

Traditional approaches to data race detection fail in cases where several correlated variables are involved. Based on our definitions of *extended* data races, computational units, and correlated sets, we have developed a new algorithm and demonstrated how to handle such cases. For our implementation, we have opted for inferring correlated sets and computational units fully automatically. We made use of the region hypothesis and proposed improvements based on the program structure, i.e. allowing shared true dependencies within function scope and limiting the effect of control dependencies to function scope.

The evaluation showed that our enhanced race detection approach is able to detect synchronization operations reliably. In contrast to previous approaches, it also works for the case of correlated variables and logical operations. Even if extended data races manifest as multiple single variable data races, our approach is still able to provide further information that helps identify the problem's root cause.

Some technical improvements are necessary and possible for the current implementation of our approach to integrate it into our race detector Helgrind$^+$ and make it more practical and usable.

We have also seen that in few cases inferring correlated sets and computational units fails. Note that one of our approach's feature is its orthogonality between race detection and finding correlated sets and computational units: we can switch to other methods for the latter, without the need to alter the former. This property will make

it easier to further improve the region hypothesis or use completely different ways to infer correlated sets in our implementation. For example, it can be worthwhile to require the user to specify at least correlated sets by annotations.

Alternatively, we could exploit new parallel programming paradigms that are currently gaining focus: i.e. `Tasks` and `Operations` that are being dispatched to execution queues, or `Futures` naturally encapsulate concepts similar to computational units. It is even feasible to extend the concept of computational units and apply it to automatic parallelization methods [31]. We leave exploring such possibilities for future work.

References

1. Lu, S., Park, S., Seo, E., Zhou, Y.: 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. ACM, New York, NY, USA, pp. 329–339 (2008). doi:10.1145/1346281.1346323
2. Netzer, R.H.B., Miller, B.P.: What are race conditions? Some issues and formalizations. ACM Lett. Program. Lang. Syst. **1**(1), 74–88 (1992). doi:10.1145/130616.130623
3. Raza, A.: A review of race detection mechanisms. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR, Vol. 3967 of Lecture Notes in Computer Science. Springer, Berlin, pp. 534–543 (2006)
4. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser : a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997). doi:10.1145/265924.265927
5. Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. SIGPLAN Not. **26**(12), 85–96 (1991). doi:10.1145/127695.122767
6. Jannesari, A., Tichy, W.F.: Library-independent data race detection. IEEE Trans. Parallel Distrib. Syst. **PP**(99), 1–13 (2013). doi:10.1109/TPDS.2013.209
7. Jannesari, A., Tichy, W.: Identifying ad-hoc synchronization for enhanced race detection. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–10 (2010). doi:10.1109/IPDPS.2010.5470343
8. Jannesari, A., Bao, K., Pankratius, V., Tichy, W. F., Helgrind+: an efficient dynamic race detector. In: Parallel and Distributed Processing Symposium, International 0, pp. 1–13 (2009). doi:10.1109/IPDPS.2009.5160998
9. Jannesari, A., Tichy, W.F.: On-the-fly race detection in multi-threaded programs. In: PADTAD '08: Proceedings of the 6th Workshop on Parallel and Distributed Systems, ACM, New York, NY, USA, pp. 1–10 (2008). doi:10.1145/1390841.1390847
10. Harrow, J.J.: Runtime checking of multithreaded applications with visual threads. In: Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification, pp. 331–342. Springer, London (2000). http://citeseer.ist.psu.edu/harrow00runtime.html
11. Pozniansky, E., Schuster, A.: Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: research articles. Concurr. Comput. Pract. Exp. **19**(3), 327–340 (2007). doi:10.1002/cpe.v19:3
12. Yu, Y., Rodeheffer, T., Chen, W.: Racetrack: efficient detection of data race conditions via adaptive tracking. SIGOPS Oper. Syst. Rev. **39**(5), 221–234 (2005). doi:10.1145/1095809.1095832
13. Hammer, C., Dolby, J., Vaziri, M., Tip, F.: Dynamic detection of atomic-set-serializability violations. In: ICSE '08: Proceedings of the 30th International Conference on Software Engineering, ACM, New York, NY, USA, pp. 231–240 (2008) doi:10.1145/1368088.1368120
14. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, pp. 334–345 (2006). doi:10.1145/1111037.1111067
15. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)

16. Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y.: Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: SOSP '07: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, ACM, New York, NY, USA, pp. 103–116 (2007). doi:10.1145/1294261.1294272
17. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. SIGPLAN Not. **40**(6), 1–14 (2005). doi:10.1145/1064978.1065013
18. Collins, J.D.,Tullsen, D.M., Wang, H.: Control flow optimization via dynamic reconvergence prediction. In: MICRO 37: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, Washington, DC, USA, pp. 129–140 (2004). doi:10.1109/MICRO.2004.13
19. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). doi:10.1145/359545.359563
20. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42**(6), 89–100 (2007). doi:10.1145/1273442.1250746
21. Nethercote, N., Seward, J.: Valgrind: a program supervision framework. http://valgrind.org/
22. Butenhof, D.R.: Programming with POSIX Threads. Professional Computing Series. Addison-Wesley, Reading (1997)
23. Apache http server project, http://www.apache.org/
24. Data-race-test: a test suite for data race detectors. http://code.google.com/p/data-race-test/
25. Butler, N.: Petridish: Multi-threading for performance in c#. http://www.codeproject.com/Articles/26453/PetriDish-Multi-threading-for-performance-in-C
26. Reichl, D.: Keepass password safe. http://keepass.info/
27. Smart thread pool. http://smartthreadpool.codeplex.com/
28. Microsoft, Code gallery for parallel programs. http://code.msdn.microsoft.com/Samples-for-Parallel-b4b76364
29. Intel inspector xe 2013. http://software.intel.com/en-us/intel-inspector-xe
30. Kim, M., Kim, H., Luk, C.-K.: Sd3: A scalable approach to dynamic data-dependence profiling. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43, IEEE Computer Society, Washington, DC, USA, pp. 535–546 (2010). doi:10.1109/MICRO.2010.49
31. Li, Z., Jannesari, A., Wolf, F.: Discovery of potential parallelism in sequential programs. In: Proceedings of the 42nd International Conference on Parallel Processing. PSTI '13, Washington, DC, USA, IEEE Computer Society, pp. 1004–1013 (2013)