Dynamic Data Race Detection for Correlated Variables

Ali Jannesari, Markus Westphal-Furuya, and Walter F. Tichy

Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology
76131 Karlsruhe, Germany
{jannesari,westphal,tichy}@kit.edu
http://www.kit.edu

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. For data race detection itself, we combine a modified version of the lockset algorithm with happens-before analysis providing 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 correct bug fixing.

Keywords: data race detection, parallel programs, dynamic analysis, correlated variables.

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?

Y. Xiang et al. (Eds.): ICA3PP 2011, Part I, LNCS 7016, pp. 14–26, 2011.

1.1 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 Figure 1, 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.

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. In this paper, we present a new approach to data race detection that will help close this gap. Roughly speaking, we adapt the lockset algorithm and the happens-before analysis to build a *dynamic hybrid data race detector for correlated variables and logical operations*. 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:

Data: shared vector (x, y); local variables a, b, max; lock m

Function *scaleVector*
lock(m) ;
$(a, b) \leftarrow (x, y)$;
unlock(m) ;
$max \leftarrow a$ **if** $a > b$ **else** b
lock(m) ;
$(x, y) \leftarrow (\frac{x}{max}, \frac{y}{max})$;
unlock(m) ;

Fig. 1. Function with concurrency bug

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 will be based on this definition.

2 Related Work

There is a lot of prior work dealing with data race detection for single memory locations [2,6,4]. 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: The papers [7,8] 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. MUVI [10] 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. The Serializability Violation Detector (SVD) [11] uses dynamically derived control and data dependencies to detect computational units on the fly. When computational 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.

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 [7], 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

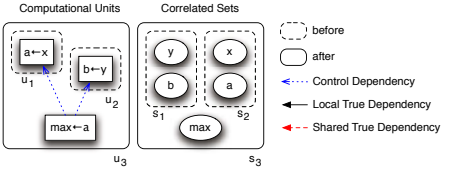
3.1 Inferring Correlated Sets and Computational Units

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* [11] for computational units: (a) All operations of a computational unit are related through either true data dependencies (read after write) or control dependencies. (b) 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:

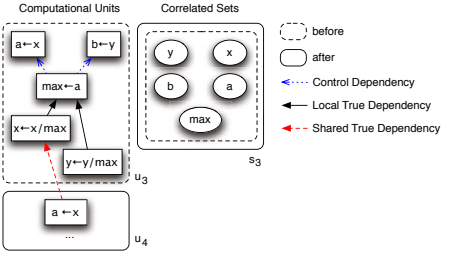
1. Initially, each dynamic operation (instruction) forms its own computational unit and each memory location its own correlated set.
2. When an operation op_1 is executed:
   - We *merge* the computational units of all dynamic operations that op_1 depends on through a control dependency.

- We *merge* the computational units of all dynamic operations that op_1 depends on through a true data dependency[1]. 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 Figure 2 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 u_3. Furthermore, when executed a second time, a shared true dependency on x is observed, ending u_3 and starting u_4.



(a) Merging computational units and correlated sets due to control dependency



(b) Ending a computational unit due to shared true dependency

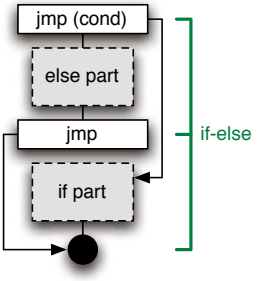
Fig. 2. Region hypothesis applied to the function *scaleVector* of Figure 1

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.

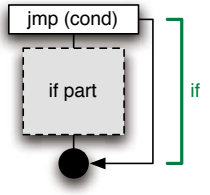
For the *scaleVector* function, the region hypothesis obviously led to correct results. However, because of its heuristic nature, this must not always be the case: In fact, experiments in [11] showed that the region hypothesis holds on the most common paths of 14 examined atomic regions but fails on some rare paths. One

[1] 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.

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 [8] and [7]). 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 [11] control dependencies were completely ignored. 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 [12]. 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 3 illustrates these two cases. However, in contrast to [12] and [11] that are limited to `if` and `if-else` constructs, we're also able to identify loops. This is possible, because of using our loop detection patterns introduced in [3,4]. Furthermore we support non-local jumps caused by `break`, `continue` or `return`.



(a) `if-else` construct



(b) `if` construct

Fig. 3. Finding the reconvergence point (black filled circle)

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 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.

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 Figure 4):

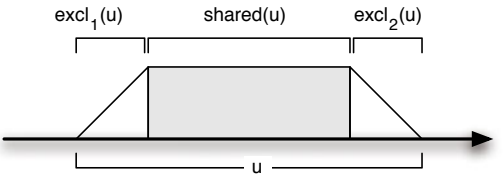


Fig. 4. Different parts of a computational unit u

- $excl_1(u)$: u accesses only exclusive variables
- $shared(u)$: 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:

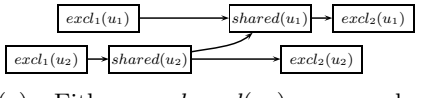
$$L_u := \begin{cases} held_u, & shared(u) \neq \emptyset \\ all\_locks, & \text{otherwise} \end{cases}$$

where

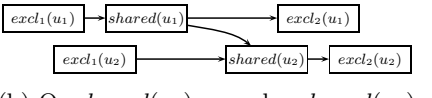
$$held_u := \{\text{Lock } m \mid m \text{ held throughout } shared(u)\}$$

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 section 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).

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:

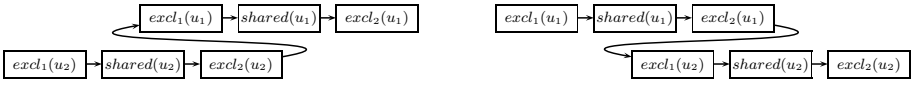


(a) Either $shared(u_2)$ precedes $shared(u_1)$



(b) 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 section 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 the left side while the second is always equivalent to the right side:



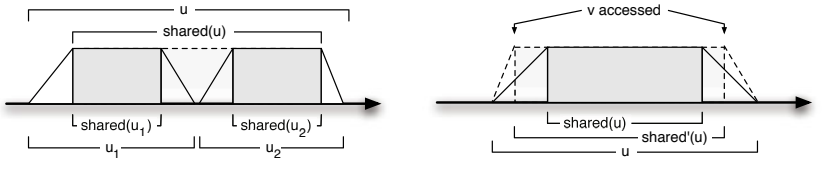
Therefore, our initial assumption implies that all possible interleavings of u_1 and u_2 must be equivalent to either $u_2 \rightarrow u_1$ or $u_1 \rightarrow u_2$ and are thus *serializable*. If we generalize this observation for s with $C_s \neq \emptyset$ and an arbitrary number of u_i accessing 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 $shared(u_2)$ (Figure 5 (a)).
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 (Figure 5 (b)).



(e) Merging of computational units (f) Exclusive variable becomes shared

Fig. 5. Reasons for $shared(u)$ to change

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:

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

$\mathbf{l}_t$ 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 $m.lock()$ 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. 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:

$$L_u = \{\text{Lock } m \mid \overbrace{\mathbf{l}_{\text{fst},u}(m)_{\text{acq}} - \mathbf{l}_{\text{fst},u}(m)_{\text{rel}} = 1}^{m \text{ held at the beginning of } shared(u)} \wedge \underbrace{\mathbf{l}_{\text{lst},u}(m)_{\text{rel}} - \mathbf{l}_{\text{fst},u}(m)_{\text{rel}} = 0}_{m \text{ not released until the end of } shard(u)} \}$$

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

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

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:

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

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 Figure 1. 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

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)	$\mathbf{(1,0)}$	$excl_1$	–	–	*all*	*all*
$(a,b) \leftarrow (x,y)$	$(1,0)$	$\boldsymbol{shared}$	$\mathbf{(1,0)}$	$\mathbf{(1,0)}$	$\mathbf{\{m\}}$	$\mathbf{\{m\}}$
unlock(m)	$\mathbf{(1,1)}$	*shared*	$(1,0)$	$(1,0)$	$\{m\}$	$\{m\}$
$max \leftarrow a$ **if** $a > b$ **else** b	$(1,1)$	$\boldsymbol{excl_2}$	$(1,0)$	$(1,0)$	$\{m\}$	$\{m\}$
lock(m)	$\mathbf{(2,1)}$	$excl_2$	$(1,0)$	$(1,0)$	$\{m\}$	$\{m\}$
$(x,y) \leftarrow (\frac{x}{max}, \frac{y}{max})$	$(2,1)$	$\boldsymbol{shared}$	$(1,0)$	$\mathbf{(2,1)}$	$\mathbf{\emptyset}$	$\mathbf{\emptyset}$

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, $\mathbf{l}_{\mathrm{lst},u}(m)$ 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 detector, our approach therefore combines the lockset algorithm with the *happens-before relation* $\rightarrow_{\mathrm{hb}}$ that tracks the temporal and causal ordering of events. For two such events e_1 and e_2 we define:

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

The $\rightarrow_{\mathrm{hb}}$ 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 [14,13]. For two events e_1 and e_2 that take place at times $\mathbf{v}_1$ and $\mathbf{v}_2$, we then have $e_1 \rightarrow_{\mathrm{hb}} e_2 \Leftrightarrow \mathbf{v}_1 < \mathbf{v}_2$ with element-wise comparison.

To combine the lockset algorithm and $\rightarrow_{\mathrm{hb}}$, we use a state machine based on our race detector Helgrind$^+$ [5,4,3]. 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 $\mathbf{v}_s \not< \mathbf{v}'$.

4 Evaluation

Our implementation borrows from Helgrind$^+$ [4], which in turn is based on the Valgrind framework [15,16]. The general principle behind Valgrind is called *disassemble-instrument-resynthesize*. Helgrind$^+$, among other things, provides an implementation of *shadow memory*. Our implementation uses shadow values to store references to point the correlated set and a collection of computational units. In experimental evaluation, we look at complex examples taken from real-world applications to check if computational units, correlated sets, and extended data races are detected. We compare our results to our enhanced race detector, Helgrind$^+$ [3,4], which serves as representative for tools that do not natively support multi-variable race detection.

4.1 Detecting Extended Data Races

The results of our evaluation focuses on the detection of correlations and extended data races shown in Table 2. The simplified codes taken from real applications are indicated as Tests 1 to 8 in the table.

Table 2. Detected correlated sets, computational units and data races

Test-No.	Description	Expected	Helgrind$^+$	Our approach CUs & CSets	Our approach Race
1	ScaleVector without locks	Race	Race	✓	Race
2	ScaleVector with interrupted locks	Race	× *No race*	✓	Race
3	Normalize with consistent locking	No Race	No race	✓	No race
4	Different locks	Race	× *No race*	✓	Race
5	AppendBuffer without locks	Race	× *Multiple races*	✓	Race
6	Swapping correlated variables with locks	No race	No race	✓	No race
7	Swapping uncorrelated variables with locks	No race	No race	×	× *Race*
8	Independent calculations	–	–	×	–

Tests 1 and 2 simply represent the function *scaleVector* from Figure 1. For test 1, the locks were completely omitted, whereas in test 2 the locks were kept as in Figure 1. While Helgrind$^+$ is able to detect the locking violation in test 1, it fails in test 2: 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 2.

Test 3 represents another arithmetic function for vector normalization (Figure 6). 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.

```
1 normalize() {
2  lock(&m);
3  float len = sqrt(a*a + b*b);
4  a = a/len; b = b/len;
5  unlock(&m);
6 }
```

Fig. 6. Test 3: Vector normalization

A similar scenario is represented by test 4, shown in Figure 7: 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 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,

```
1  append(char *s) {
2   lock(&m1);
3   mCont = f(mCont,s);
4   unlock(&m1);
5  }
6  reflow(char *s) {
7   lock(&m2);
8   mLen = g(mLen,s);
9   unlock(&m2);
10 }
```

Fig. 7. Test 4: Using different locks for correlated variables

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 5 is shown in Figure 8; it is part of Apache's `log_config` module, 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 `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 8 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`.

```
1  void bufferAppend(char *str) {
2     int len = strlen(str);
3     if (len+outCnt >= BUFSIZE) {
4        // flush buffer!
5        outCnt = 0;
6     }
7     for (int i = 0; i < len; i++) {
8        outBuf[outCnt+i] = str[i];
9     }
10    outCnt = outCnt+len;
11 }
12 int strlen(char *str) {
13    int ctr = 0;
14    while(str[ctr]) { ctr++; }
15    return ctr;
16 }
```

Fig. 8. Test 5: Appending to a buffer without locks

In tests 6 and 7, two shared variables `a` and `b` are swapped using a temporary variable:

```
tmp = a; a = b; b = 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

```
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. 9. Test 8: Independent operations within a atomic region

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 8 [11], shown in Figure 9, 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.

5 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 demonstrated how to modify the lockset algorithm and the happens-before analysis for 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 informations that helps identify the problem's root cause.

Some technical improvements are necessary 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. `Task`s and `Operation`s that are being dispatched to execution queues, or `Future`s naturally encapsulate concepts similar to computational units. 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, pp. 329–339. ACM, New York (2008)
2. 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)
3. 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 (19-23, 2010)
4. Jannesari, A., Bao, K., Pankratius, V., Tichy, W.F.: Helgrind+: An efficient dynamic race detector. In: International on Parallel and Distributed Processing Symposium, pp. 1–13 (2009)
5. Jannesari, A., Tichy, W.F.: On-the-fly race detection in multi-threaded programs. In: PADTAD 2008: Proceedings of the 6th Workshop on Parallel and Distributed Systems, pp. 1–10. ACM, New York (2008)
6. 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)
7. Hammer, C., Dolby, J., Vaziri, M., Tip, F.: Dynamic detection of atomic-set-serializability violations. In: ICSE 2008: Proceedings of the 30th International Conference on Software Engineering, pp. 231–240. ACM, New York (2008)
8. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL 2006: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 334–345. ACM, New York (2006)
9. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
10. 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 2007: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 103–116. ACM, New York (2007)
11. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. SIGPLAN Not. 40(6), 1–14 (2005)
12. 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, pp. 129–140. IEEE Computer Society, Washington, DC, USA (2004)
13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
14. Fidge, J.: Timestamps in Message Passing Systems that Preserve the Partial Ordering. In: Proc. 11th Australian Computer Science Conf., pp. 55–66 (1988)
15. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (2007)
16. Nethercote, N., Seward, J.: Valgrind: A program supervision framework (2003), `http://valgrind.org/`
17. Butenhof, D.R.: Programming with POSIX Threads. ser. Professional Computing Series. Addison-Wesley, Reading (1997)