Detecting Correlation Violations and Data Races by Inferring Non-deterministic Reads

Ali Jannesari[1,2], Nico Koprowski[1,2], Jochen Schimmel[3], Felix Wolf[1,2], and Walter F. Tichy[3]

[1]German Research School for Simulation Sciences, Aachen, Germany
[2]RWTH Aachen University, Aachen, Germany
[3]Karlsruhe Institute of Technology (KIT), Germany
{a.jannesari, n.koprowski, f.wolf}@grs-sim.de, {schimmel, tichy}@kit.edu

Abstract—**With the introduction of multicore systems and parallel programs concurrency bugs have become more common. A notorious class of these bugs are data races that violate correlations between variables. This happens, for example, when the programmer does not update correlated variables atomically, which is needed to maintain their semantic relationship. The detection of such races is challenging because correlations among variables usually escape traditional race detectors which are oblivious of semantic relationships. In this paper, we present an effective method for dynamically identifying correlated variables together with a race detector based on the notion of non-deterministic reads that identifies malicious data races on correlated variables. In eight programs and 190 micro benchmarks, we found more than 100 races that were overlooked by other race detectors. Furthermore, we identified about 300 variable correlations which were violated by these races.**

I. Introduction

A notorious class of concurrency bugs in parallel programs are race conditions. A race condition on a variable may cause the variable to acquire an unexpected value, which may lead to anomalous program behavior. Such variables may even correlate with each other, which means their values are mutually dependent on each other. Concurrency bugs in general are hard to reproduce and to identify manually [1]. As a result, automatic race detection has been an important research field for several years. However, the impact of correlated variables has only recently been of interest [2], [3], [4]. Conventional race detectors often fail to identify race conditions involving correlations between two or more variables. Taking into account correlations between variables was shown to enhance the precision at which races are detected. So far, the successful detection of races on correlated variables was thought to require the prior identification of correlations among variables [2], [3], [4]. For this purpose, current approaches either request information on correlations among variables directly from the user or apply correlation detection algorithms to the code.

In this paper, we introduce a novel approach to the detection of races: It uses a race detection criterion that does not depend on any prior knowledge of correlations, but is still able to identify correlations while detect race conditions. In this way, we can detect races on correlated variables, whose correlations escape conventional methods. As we still identify correlations between variables in the analysed program we can additionally report potential correlations that a race condition violates. As a result the user can categorize observed race conditions and better foresee their impact on the program. During the evaluation of our approach we found more than 100 races among eight different programs and 190 micro benchmarks which have not been found by other race detectors. At the same time we were able to identify about 300 correlations that were violated by these races.

In Section II, we introduce the basic terms of race conditions and variable correlations. Further on, Section III discusses related work, detailing previous approaches to race detection on correlated variables. Section IV describes the core of our approach. Here, we present the race detection criterion, the method of finding variable correlations, and the overall race detection algorithm. Section V briefly explains how we implemented the approach presented in Section IV. In Section VI, we present experimental results of our approach, which we compare to three other race detectors. Finally, Section VII summarizes the paper and gives an outlook on future work.

II. Basics

In this section we introduce terms that we use throughout the paper. We also present some basic techniques we apply and explain how we define correlations between variables.

Race Conditions: Parallel computation may lead to concurrency bugs, which include race conditions, atomicity violations, order violations and deadlocks [5]. In the context of this paper, we restrict ourselves to race conditions. The exact definition of a race condition varies wildly [6]. Generally, a race condition is an anomalous behaviour due to unexpected critical dependence on the relative timing of events.

In this work, we focus on race conditions involving variables and threads. By race condition we mean an anomalous behaviour due to a variable's value unexpectedly depending on the scheduling of threads. Despite the restriction on variables and threads our definition is also valid for most other work covering this topic. This is because race conditions are usually considered for variables and threads only. In Section III, we mention some of these works.

Variable Correlations: We consider the code example in Figure 1. Thread A converts a currency value represented in Euro to its representation in Yen. Obviously, the value of the Yen variable depends on the value of the Euro value. Since

```
Thread A                      Thread B
    Euro = 300;                   Euro = 500;
    Print(Euro);                  Print(Euro);
    Yen = Euro * 107;             Yen = Euro * 107;
    Print(Yen);                   Print(Yen);
end                           end
```

Fig. 1. A race condition on $Euro$ violating the correlation between $Euro$ and Yen.

we calculate the Yen value from the Euro value, Euro and Yen are correlated. Variables correlate iff there exists a data, control or logical dependency between their values. Logical dependencies are particularly hard to identify in code: How should an analysis determine that a name and a number form an address and the respective variables are therefore correlated? For example, in Figure 2 we can see a data dependency between $Street$ and $PrivateStreet$. Also, the figure contains a control dependency between $isPrivate$ and the variables $Street$ as well as $StreetNo$. From the labeling of the variables $Street$ and $StreetNo$ we can infer a logical dependency between them. But without any data or control dependency involved, it is a very difficult task for an algorithm to decide whether two variables are logically correlated. Hence, we believe that the identification of correlations is at least as complex as the race detection process itself. However, there are race detectors which have to identify correlations first before they are able to detect races. Therefore, they may tend to be more unreliable than a race detector which is able to report all races independently of recognized correlations.

```
if isPrivate == true then
    Street = PrivateStreet;
    StreetNo = 59;
end
```

Fig. 2. Example of data, control and logical dependencies.

A race condition may violate correlations between variables and, therefore, cause (more) anomalous behavior inside the program. Therefore, the interaction of correlated variables and race conditions may negatively influence program behavior. In fact, more than 30% of all race conditions involve correlated variables [5]. Figure 1 illustrates an example of such a violation. Two threads each convert a value in the currency of Euro to the currency of Yen. We can see that, for this, the value of the $Euro$ variable is converted and stored to Yen. The two concurrent write accesses on $Euro$ may influence the data dependency between $Euro$ and Yen in both threads. In thread A, we expect Yen to contain the converted amount of 300 Euro. But depending on the scheduling of threads it may also acquire the converted amount of 500 Euro. The result does not meet the expectations. Therefore, we have a race condition which violates the correlation between $Euro$ and Yen. The reasoning for thread B is analogous. To resolve the race condition, contiguous lock sections (i.e., critical sections) containing all statements involved in the update are necessary (see Figure 3).

```
Thread A                          Thread B
    Acquire Lock                      Acquire Lock
        Euro = 300;                       Euro = 500;
        Print(Euro);                      Print(Euro);
        Yen = Euro * 107;                 Yen = Euro * 107;
        Print(Yen);                       Print(Yen);
    end                               end
end                               end
```

Fig. 3. Resolving the race condition in Figure 1 by using lock operations.

III. Related Work

Race detection is a broad field of work. There are many works which cover the detection of conventional race conditions such as [7] or works considering atomicity violations [8], [9]. However, in the scope of this paper we want to focus on race detectors considering correlations between variables.

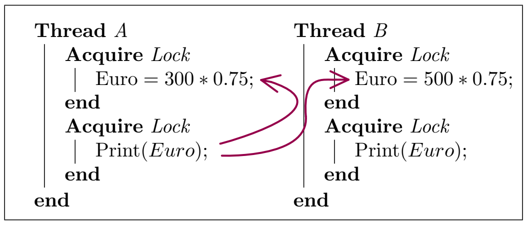
CHESS [10], [11] dynamically searches a program for parallelization bugs including races, deadlocks and data errors. Through user annotations CHESS is also able to identify race conditions violating correlations between variables. However, to achieve full coverage the approach needs to perform a dynamic analysis for each thread schedule of interest. This naturally leads to comparatively high analysis times. [12] and [3] introduce a dynamic race detector for correlated variables. The race detector identifies variable correlations through user annotations in the source code. The algorithm checks concurrent accesses to correlated variables for serializability. In this way, unserializable access patterns are identified as race conditions. However, the occurrence of harmful access patterns is highly schedule dependent. MUVI [4] is an hybrid race detector for correlated variables. The algorithm recognizes correlations among variables by applying a static analysis which uses data mining techniques. A subsequent dynamic analysis checks for race conditions by identifying concurrent and unsynchronized accesses to correlated variables. $Helgrind^+$ for correlated variables [2] (H^{Corr}) is a purely dynamic race detection approach. It is based on the dynamic race detector $Helgrind^+$ [13] for single variables which runs on the virtual execution environment Valgrind [14]. The approach uses dynamic analysis to identify variable correlations using data and control dependencies. At the same time, the algorithm dynamically checks for concurrent and unsynchronized accesses to already recognized correlated variables. Common to all approaches above is that they all need information on correlated variables either via user annotations or via correlation-detection heuristics.

IV. Approach

We introduce a dynamic analysis method capable of detecting both correlations and race conditions at the same time, but independently from each other. This is in contrast to previous works in which the results of the race detection were dependent on the identification of correlations. Nevertheless, for a given race condition reported by our detector, we are able to state which correlations are affected and provide valuable information to the end user.

A. Race Detection

We identify race conditions regardless of whether the involved variables are correlated or not. Therefore, we first define the criteria on how we can recognize a race condition, which



```
Thread A                         Thread B
    Acquire Lock                     Acquire Lock
        Euro = 300 * 0.75;               Euro = 500 * 0.75;
    end                              end
    Acquire Lock                     Acquire Lock
        Print(Euro);                     Print(Euro);
    end                              end
end                              end
```

Fig. 4. Sample where each read access on $Euro$ has two write dependencies.

we introduce as non-deterministic reads. Identifying these criteria, in turn, requires information about the synchronization between instructions executed in parallel. For this reason we define synchronization states. Subsequently, to improve performance via search space reduction, we introduce the concept of epochs that group variable information into epoch variables. The race detector is oblivious of correlations. Correlation detection is performed independently of race detection. As soon as a race is reported, correlation information is attached to the race description.

Non-deterministic Reads: A read access on a variable can obtain its value from one or more possible write accesses on that variable. We refer to these write accesses as the *write dependencies* of that read access. Only the thread scheduling decides which of the possible write accesses creates the read value during execution.

In Figure 4 you can see an example of a read access having two write dependencies. All instructions are locked. However, the read access for the print can either acquire the written value from the assignment in thread A or B. This entirely depends on the thread scheduling. Hence, we regard both write accesses as write dependencies of the read.

From write dependencies we can infer non-deterministic reads. A non-deterministic read is a read access that either

1) has multiple write dependencies and/or
2) has at least one write dependency executed in parallel.

We use non-deterministic reads as a detection criterion for race conditions.

A non-deterministic read indicates a variable's value dependence on thread scheduling. Additionally, since the value is read it influences the program behavior. This comes close to our definition of a race condition criterion. However, our criterion includes false positives. This means, not every non-deterministic read is actually a race condition. A non-deterministic read inside a program can also be intentional or harmless. Figure 5 shows an example of an intentional or harmless non-deterministic read. During the incrementation, i has a parallel write dependency inside the other thread. However, what matters in this case is the final result which will always be $i + 2$.

Even though our criterion returns false positives, we consider it as complete: We can identify all sources where potentially harmful non-determinism results from thread scheduling. The reasoning behind this statement is follows: Obviously, if every read access inside a program is deterministic we can exclude a race condition by definition. There is either no variable value which is dependent on the scheduling of

```
Thread A                  Thread B
    Acquire Lock              Acquire Lock
        i++;                      i++;
    end                       end
end                       end
```

Fig. 5. An example of intentional and harmless non-deterministic reads.

threads or it does not influence program behavior since it is not read. Conversely, a program containing a race condition must have at least one write access which causes a variable's value dependency on the thread scheduling. Since the race must influence program behavior, the value created by that write access must be read. Hence, the read access may acquire the variable's value depending on the thread scheduling. That means, the read access must be a non-deterministic read.

For identifying non-deterministic reads we have to identify the write dependencies of a read access. We do this by introducing happens-before relations and locks. Let r be a read access and w be a write access on variable v, then w is a write dependency of r iff all of the following criteria are met:

1) **Relevance:** r does not happen before w.
2) **Directness:** There exists no write dependency d of r for which it holds that w happens before d.
3) **Freedom:** There exist no lock l and no write access w' on v for which it holds that l continuously secures w' as well as r and l secures w.

Figure 6 illustrates some scenarios for the criteria defined above. The criterion $relevance$ ensures that the program does not $always$ execute a potential write dependency after the read access. If it did, the write obviously could never have any effect on the read access. The left code snippet shows a write access on x which is executed after the read access and is therefore no write dependency. $Directness$ means that we always regard the latest potential write dependency that is executed before the read access or in parallel to it. So, we exclude write accesses, which are always executed before other writes. In the middle code snippet the first write access on x is overwritten by the subsequent write access. Therefore, only the second write access is a write dependency of the read access. Finally, in criterion $freedom$ we also consider locks. When a read access is contiguously locked with a write dependency, then no other write access, secured by the same lock, can be a write dependency of the read access. As an example of this, see the right code snippet: All other write accesses on x secured by $Lock$ are either executed before (making them indirect) or after (making them irrelevant) the depicted critical section.

```
Relevance:        Directness:       Freedom:
    Print(x);         x = 2;            Acquire Lock
    x = 4;            x = 5;                x = 5;
end                   Print(x);             Print(x);
                  end                   end
                                    end
```

Fig. 6. Scenarios for the write dependency criteria.

Synchronization States: In order to obtain happens-before relations and locks we introduce synchronization states. At any moment during execution, a thread has specific synchronization properties with regard to other threads. Henceforth, we can say a thread is always in a certain state of synchronization. Such a state expresses the happens before and locking relations between instructions executed by this thread and instructions from other threads. A synchronization state consists of:

- A vector clock [15] for determining happens-before relations
- A lock vector indicating which locks the thread currently holds and how often the thread has acquired each lock so far

We use the number of acquisitions for each lock to determine whether the thread has released the lock in between. In this way we can recognize continuous locks between accesses.

Epochs: We have seen, that whether a write access is a dependency of a read access is dependent on its own and the read access's state of synchronization. Accesses sharing the same synchronization state can therefore be treated equally. Consider Figure 7 for example. Here, both read accesses on x in thread A share the same state of synchronization. It is sufficient to determine the parallel write dependency of only one read access. The other read must have the same dependency since its synchronization state is not different. This means, for determining write dependencies we can form equivalence classes of read and write accesses according to their synchronization state. This reduces our state space for performing our race detection analysis.

```
Thread A                 Thread B
    Print(x);                x = 5;
    Print(x);            end
end
```

Fig. 7. Thread A contains two read accesses with the same synchronization state. Both have the write access in thread B as a parallel write dependency.

We partition the thread execution into epochs. An *epoch* is a sequence of instructions that all have the same synchronization state, i.e., the thread does not execute any synchronization instructions within an epoch. Furthermore, we want to consider only $maximum$ epochs, i.e., epochs which consist of the longest possible sequence. In each epoch we store the information of which variables the thread has accessed and how (read, written or both) it accessed them. We will call this information *epoch variables*. Like between access we can recognize happens-before relations and locks. Therefore, we infer write dependencies between epoch variables with the same three established criteria for variable accesses: relevance, directness and freedom.

B. Correlations

Correlations imply expectations the user or the program has regarding the variables. If there is a correlation between two variables we expect that the values of these variables are in some way related to each other. We already mentioned that we regard this relation as either a data, control or logical dependency. One possible effect of a race condition is that it may violate such a relation and therefore also the corresponding correlation. In this section we present an approach that determines which correlations are violated if a read access is non-deterministic. For this, the process of inferring relations between variables (and therefore correlations) is inspired by the method of H^{Corr}.

During dynamic execution H^{Corr} tries to identify relations between variables by considering data and control dependencies. Assignments and control dependencies establish relations between the participating variables. During an assignment the read variables, the related variables of the read variables, and the written variable become all related to each other. Likewise, a variable which is control dependent on another variable becomes related to that variable and all its relations. At the same time H^{Corr} considers, for the purpose of race detection, these related variables to form a computational unit. These computational units are accesses which form a critical region regarding a specific computation. When H^{Corr} detects the end of a computational unit the approach clears the relationship of all involved variables and begins anew.

We try to find relationships and therefore correlations between variables in a similar way. However, our approach is enhanced in three ways:

1) Since we are not dependent on computational units for race detection, we do not consider them for clearing relationships between variables. Instead we tie the lifetime of a relationship to the lifetime of the responsible data and control dependency. As a result, we do not miss relationships between variables in future computations. The code example in Figure 8 illustrates this difference. After the execution of the code H^{Corr} would establish a new computational unit between the participating variables and forget about their relationships. Our approach still keeps this important information for later use.
2) We keep track of the directions of the dependencies. H^{Corr} can tell that the variables are related but not how they are related. We use this additional information to categorize the recognized correlations. This directly benefits the information content for the end user.
3) We do not only consider data dependencies established through assignments but also through hierarchical (parent-child) relationships. That is a type of relationship H^{Corr} misses entirely.

With our extensions we can formulate four different patterns, which cover all possible relationships derived from data and control dependencies (see Figure 9).

```
Dependencies
    Euro = 300;
    Yen = Euro * 107;
    Dollar = Yen/97;
    Print(Dollar);
end
```

Fig. 8. Code example where $Euro$, Yen and $Dollar$ become data dependent on each other.

Vertical correlations: We consider two variables which are data dependent on each other. This means there exists a correlation between these variables. We call this correlation a $vertical\ correlation$. It ends as soon as the data dependency between its variables ends. This means, as soon as any of those two variables is rewritten.

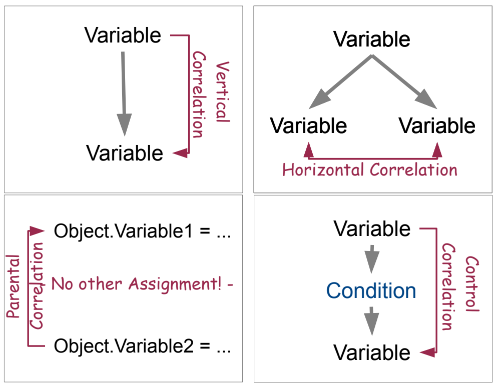


Fig. 9. Different correlation patterns.

Control correlations: Also, a control dependency infers a correlation between the corresponding variables. Therefore, two control dependent variables are in a control correlation. As the vertical correlation this correlation ends when the corresponding control dependency ends. In this case, the dependency ends if either of both variables is written outside of the corresponding control branch.

Parental correlations: In the context of classes and their field variables (OOP) we recognize data dependencies. Naturally, a variable is data dependent on the class (or object for non-static variables) it belongs to. If a program writes two variables sharing the same parent subsequently, without any write access in between, we can expect variables belonging to the same class to have something in common, like forming the address of a person. Therefore, we consider them to be parentally correlated. However, not all field variables are really correlated, as for example a person's hair color and size. We consider the chances of a real correlation higher if the two variables are written subsequently, as they tend to belong to the same computation. This correlation ends as soon as any two of its variables are written again.

Horizontal correlations infer implicit data dependencies from multiple variable assignments. If at any time, two variables are dependent on a common third variable with no redefinition of any of these variables in between, we consider these two variables horizontally correlated. This is due to the fact that the two variable's values are in a special relationship which is independent of the variable they depend on directly. The two variables retain this special relationship even if they are no longer data dependent on that variable, as their values did not change. Like the parental correlation, this correlation ends as soon as any two of its variables are written again.

We store in each read epoch variable the current correlations it participates in. The idea is to report the stored correlations of read epoch variable as being violated, as soon as we detect that the corresponding read access of the epoch variable is non-deterministic.

C. The Algorithm

Now, we will detail the race detection algorithm which is illustrated in Figure 10. In the first step, the algorithm collects correlation and access information. During dynamic execution we gather the epoch variables for each thread separately. At the same time, we independently detect variable correlations according to the criteria specified in Section IV-B. When a thread reads a variable we associate all current correlations to its corresponding epoch variable.

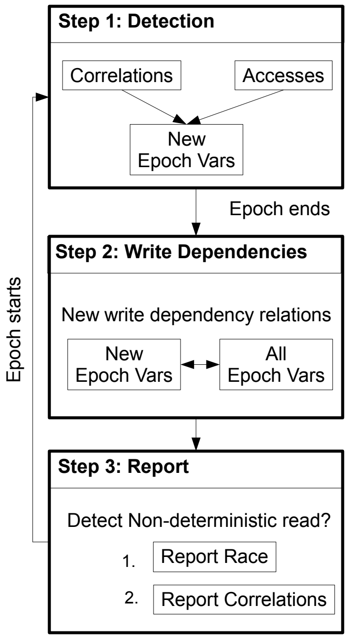


Fig. 10. Illustration of the race detection algorithm.

When a thread executes a synchronization instruction or terminates, its current epoch ends. Then, we execute the second step of the algorithm: At the end of an epoch we try to identify new write dependencies between the epoch variables. For each epoch variable that has been read, we determine its write dependencies from epoch variables of the currently and previously finished epochs. Analogously, for each epoch variable that has been written, we determine the epoch variables for which it is a write dependency. We infer write dependencies as usual by evaluating the three criteria relevance, directness and freedom. If, during that process, any epoch variable obtains more than one write dependency or one write dependency which is executed in parallel, we report a race condition for all read accesses associated with the respective epoch variable. Furthermore, we also report all associated correlations of that epoch variable, which we consider violated by that race condition.

If the thread under analysis has not yet terminated, we start a new epoch and continue with step one.

Example: Figure 11 shows an example of our algorithm. In this case the epochs we consider correspond to the lock blocks inside the threads. Hence, the execution of thread A contains two epochs while thread B only contains one epoch. In the first epoch of thread A, we gather the access information on the epoch variables for $Euro$ and Yen. We store that the epoch variable for $Euro$ is both read and written. For Yen we only identify a write epoch variable. Furthermore, due to the data dependency of the second assignment, we detect a vertical correlation between $Euro$ and Yen. As soon as the epoch finishes, we begin searching write dependencies for the epoch variable $Euro$. We recognize that it is its own write dependency since it was written first and then read. Trivially, it therefore meets all write dependency criteria: The epoch variable does not happen before itself (relevance), there is no write epoch variable in between (directness) and there is no other write dependency with which it is continuously locked(freedom). The epoch variable for Yen, on the other hand, does not have any write dependency after the first epoch has finished.

We proceed with the only epoch of thread B. At the end of the epoch we have gathered the epoch variable for $Euro$ with $written$ as its access information. We test whether this epoch variable is a write dependency of thread A's read epoch variable for $Euro$: The write epoch is relevant since it is executed in parallel. Also, it is direct since it does not happen before any other write epoch. However, it is not free because the read epoch variable is contiguously locked with itself and the write epoch is secured by the same lock. Therefore, the write epoch variable of thread B is not a write dependency of the epoch variable of the first epoch in thread A.

Finally, thread A executes its second epoch. We identify two read epoch variables for $Euro$ and Yen. When the epoch ends, we, furthermore, determine write dependencies. Trivially, concerning Yen, the write epoch variable of the first epoch in thread A is a write dependency to the current read epoch variable. Analogously we infer the same for the variable $Euro$. But additionally, we identify the write epoch variable from thread B as a second write dependency: It is parallel, direct and free. Therefore, we detect a non-deterministic read. It has two write dependencies, one of which is a parallel write dependency. As a result we report a race on the corresponding read access. Further on, since the correlation between $Euro$ and Yen is still alive, we report that this correlation has been violated by the detected race.

```
Thread A
    Acquire Lock
        Euro = 5;
        Yen = Euro * 107;
    end
    Acquire Lock
        Print(Euro);
        Print(Yen);
    end
end

Thread B
    Acquire Lock
        Euro = 2;
    end
end
```

Fig. 11. Example of a non-deterministic read on $Euro$ violating the vertical correlation to Yen.

V. Implementation

We implemented the race detector using the Microsoft .NET Framework. For data and control flow analysis as well as the code instrumentation for the dynamic race detection we employed the Common Compiler Infrastructure (CCI) framework. Therefore the presented algorithms work on the Common Intermediate Language (CIL) which underlies every .NET program. Synchronization instructions are method calls to the .NET core library which communicate with the operating system. We are able to detect these method calls inside the CIL code of the program and add our instrumented functions to replace them. Naturally, our implementation has to know all respective synchronization methods of the core library. Our implementation supports four method calls which realize locking/unlocking and signal/wait operations. Having these, the race detector currently covers the fundamental synchronization operations.

Furthermore, we need to identify control flow branches in order to recognize control correlations. We identify the scope of control flow branches via post dominator analysis and apply a simple and efficient algorithm, which was presented in [16].

For memory and state space reduction our implementation limits the number of epoch variables which we store for race detection. We consider only the n-newest epoch variables per variable and per thread. Where n is a user user specified number.

VI. Evaluation

In this section we introduce our test environment, including our evaluation metrics. Subsequently, we present our evaluation results.

A. Test Environment

For our evaluation, we created 190 different micro-benchmarks. In each micro-benchmark we execute two methods in parallel which both access one or more shared variables. In addition to the micro-benchmarks, we use real-world applications for our evaluation purposes. MSDN Code Gallery [17] contains applications which demonstrate the functionality of parallel programming in .NET. For the evaluation we chose an order-system simulation: A master thread manages many worker threads executed concurrently. We altered the synchronization operations and introduced different race condition scenarios in five different versions of the application. Additionally, we evaluated the open source programs PetriDish [18], the program library of KeyPass [19] and SmartThreadPool (STP) [20]. Their parameters are listed in Table I.

Program	# Methods	# Instr.	# Synchs	# Vars	# Threads
Synch	60	1445	24	18	5
Unsynch	60	1430	16	18	5
Locked	58	1473	60	18	5
Gaps	58	1473	20	18	5
Corr	78	2012	40	29	5
PetriDish	133	4661	43	268	7
KeePassLib	1324	54176	27	5162	4
STP	590	9591	207	277	4

TABLE I. Programs used for the evaluation. Synchronizations include starting and joining of threads.

We compare the results from our race detector to the results of Intel Thread Checker (ITC), Microsoft Research CHESS and H^{Corr}. The Intel Thread Checker is part of the Intel Inspector [21], which is a known tool for detecting conventional concurrency bugs as well as memory leaks. We chose ITC for the evaluation since it is commercial and widely used in the development of real-world applications. CHESS on the other hand is a very precise research-oriented race detector. Furthermore, for the purpose of this evaluation, we have re-implemented the race detector H^{Corr} in the .NET framework. H^{Corr} bears much resemblance to our approach in matters of correlation detection. Additionally, H^{Corr} is specialized on

Category	Expected	Races	Our tool	H^{Corr}	CHESS	ITC
R & R	No Race	0	0	0	0	0
R & W	Race	5	5	5	5	5
Locked	Race	5	5	3	0	0
False lock	Race	10	10	10	10	7
Timed	No Race	0	0	0	0	0
W & W	Race	5	5	2	5	5
Locked	Race	5	5	2	0	0
False lock	Race	10	10	4	10	8
Timed	No Race	0	0	0	0	0
Unread	No Race	0	0	2	5	5
I & I	Race	5	5	4	4	4
Locked	No Race	0	5	1	0	0
False lock	Race	10	10	9	9	7
Timed	No Race	0	0	0	0	0
I & R	Race	5	5	5	4	3
Partial lock	Race	22	19	16	0	0
Gaps	Race	22	18	15	0	0
I & W	Race	5	5	5	3	4
Partial lock	Race	22	20	20	0	0
Gaps	Race	22	21	20	0	0
Total (FN | FP)		153	10 | 5	24 | 7	103 | 5	110 | 5

TABLE II. RACE DETECTION RESULTS ON MICRO BENCHMARKS. R: READ, W: WRITE, I: INCREMENT

finding races between correlated variables. The source code for other tools which consider variable correlations was not available to us.

B. Race Detection

We present the results of our race detector according to missed race conditions and false races summarized as precision. In the second part we show the slowdown and the memory usage of our approach.

Precision: Table II summarizes results for our micro-benchmarks grouped into 6 access categories, which are (if applicable) further divided into:

- **Locked**: Accesses are contiguously locked.
- **False lock**: Accesses are protected with only one or two different locks.
- **Timed**: Accesses are fully time synchronized.
- **Partial locks**: Not all accesses are protected by a lock.
- **Gaps**: The lock protecting the accesses is not continuous.

Generally, CHESS and ITC deliver reliable results in categories where instructions are not locked correctly. However, in our evaluation they missed all order and atomicity violations in which all accesses were discretely locked. H^{Corr} is able to detect these races but occasionally misses some in various categories. We ascribe this behavior to incorrectly recognized computational units. Our detector is not dependent on recognizing computational units for race detection. Therefore, our approach delivers better overall precision. Only in the category of two fully locked concurrent increments does our race detector deliver false positives. In that category, we can observe a harmless non-deterministic read. Since our approach cannot distinguish between harmful and harmless non-deterministic reads, it reports a race condition.

```
Thread A                                  Thread B
  Acquire Lock                              Acquire Lock
    if Instance == null then                  if Instance == null then
      Instance = newInstance();                 Instance = newInstance();
    end                                       end
  end                                       end
end                                       end
```

Fig. 12. A harmless non-deterministic read on the variable $Instance$ in both threads.

Program	Total Races	Our tool FN	Our tool FP	H^{Corr} FN	H^{Corr} FP	CHESS FN	CHESS FP	ITC FN	ITC FP
Micro Bench-marks	153	11	5	32	7	103	5	110	5
Synch.	0	0	0	0	0	0	0	0	0
Unsynch.	22	2	0	3	1	0	0	12	2
Locked	22	2	0	22	0	0	0	0	0
Gaps	22	0	2	11	0	22	0	22	0
Corrs	15	1	1	3	0	9	0	14	0
PetriDish	5	0	2	3	10	5	0	4	3
KeyPassLib	10	2	8	1	14	3	8	5	3
STP	15	1	6	5	10	4	0	10	2
Total	**242**	**19**	**24**	**80**	**42**	**146**	**13**	**177**	**15**

TABLE III. FALSE POSITIVES AND FALSE NEGATIVES REGARDING RACE DETECTION.

Table III summarizes the actual numbers of false positives and false negatives that occurred during the race detection for all evaluated programs. Our race detector has by far the lowest number of false negatives. The false negatives of our dynamic approach are race conditions that lie on unvisited (unexecuted) control flow branches. Because our race detector just follows the control flow of the current execution it is unable to find race conditions on unexecuted paths. CHESS executes the code several times and has more coverage of the control flow. Inside the programs during execution the decision which branch to take depends on the scheduling of threads. Therefore it is able to detect some races our detector misses. ITC and H^{Corr} on the other hand have the same limitation as our approach: All race conditions our race detector misses are also missed by ITC and H^{Corr}. The false race conditions come from harmless (intentional) non-deterministic reads inside the programs. Figure 12 shows such an encountered harmless non-deterministic read. Each thread executes a write access on $Instance$ in parallel to a read access in the other thread: The read can either acquire the value $null$ or the instantiated object. Hence, the read access on $Instance$ in each thread is non-deterministic. But because the if-clause handles this non-determinism the non-deterministic read is harmless. Also the other detectors cannot distinguish between potentially harmful and harmless race conditions. Therefore, they share some of the encountered false positives. These are mainly false positives involving unlocked accesses that do not include correlations. Generally, we expect our approach to deliver many false positives in server-side programs, that naturally introduce much harmless non-determinism (remind figure 5 as an example).

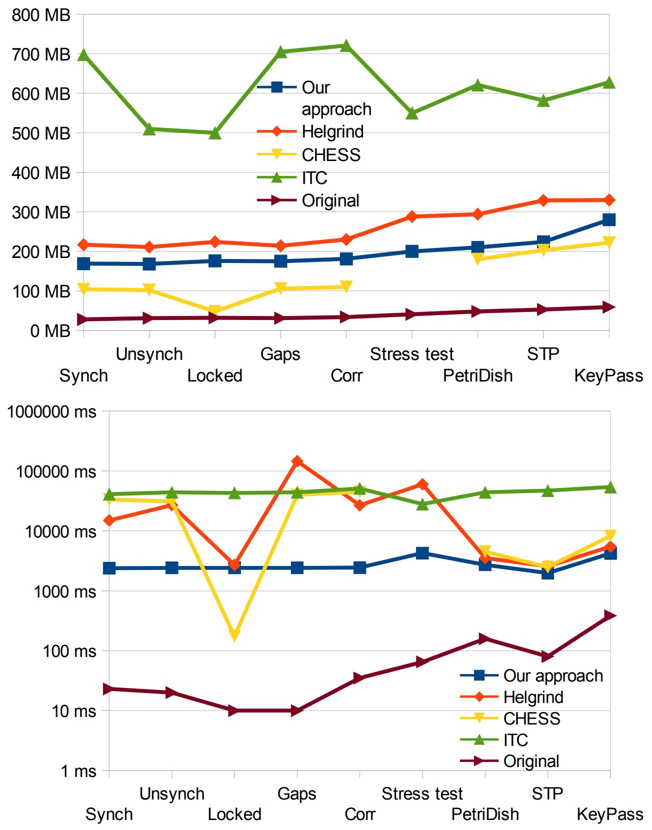


Fig. 13. Memory consumption (top) and speed (bottom) in comparison to other detectors.

Memory and time overhead: Figure 13 illustrates the slowdown and the memory usage of our dynamic race detection in comparison to the other detectors. The observed slowdown of our dynamic race detection varies from a factor of 17 to 425, depending on the analyzed program. We observed that the more threads, more synchronizations and higher percentage of shared variables a program contains the slower our implementation gets. The worst case scenario is represented by the stress test in which several threads repeatedly create objects and access their shared field variables. Remarkably, the stress test, which was the worst case scenario for our approach, turned out to have an even greater impact on other race detectors: H^{Corr} raised to a factor of 14500 while CHESS proved to be not scalable at all and crashed. We were not able to measure the time for the stress test for CHESS. Only ITC's slowdown was comparatively unaffected by the stress test. Despite the overall high slowdown, our approach is able to compete with H^{Corr}, CHESS and Intel Thread Checker. CHESS performs a faster analysis only on the program *Locked*. We observed, that CHESS operates faster the fewer synchronization operations and thread switches a program execution contains. As the program *Locked* contains only a few contiguous locks which cover most of the code, during execution we observe few synchronizations and few thread switches.

Furthermore, the observed memory usage of our implementation ranges between 4 to 6 times the original memory. Only CHESS consumes less memory for its analysis. The memory usage of H^{Corr} is slightly higher but tends to be comparatively close to our approach. ITC, on the other hand, consumes far more memory than any of the race detectors.

Because our algorithm showed to be not scalable with regard to the slowdown we consider it most suitable for parallel unit tests [22]: They represent small parallel program fractions which need to be checked for concurrency bugs. As these tests are very small, we can use them to find data races in large and complex software programs. We aim to use our detector in conjunction with some recent tools for automatic generation of parallel unit tests [23], [24].

C. Correlation Detection

This section describes the results of our dynamic correlation detection. Since we have already presented the slowdown of our race detector, we evaluate the results according to correctness and completeness. Since CHESS relies on user annotations for identifying correlations and ITC does not support correlations at all, we compared our results only to H^{Corr}. For our evaluation we considered correlations that have been violated by a race condition. In this sense a false negative means, that a detector failed to identify a violated correlation. Analogously, a false positive is an identified correlation which has not been violated.

Table IV shows the number of false positives and false negatives regarding the reported correlations. Our approach misses purely logical correlations, i.e. correlations without any data or control dependency. Obviously, our approach restricts itself to identify control and data dependencies only. Therefore our approach misses the violated logical correlations inside the programs. Since H^{Corr} also does not consider logical dependencies, it shares these false negatives with our approach.

Correlations established by transitive dependencies are another source of false negatives. For example in Figure 14, $Euro$ and Yen as well as Yen and $Dollar$ are directly data dependent on each other. However, there is also a transitive dependency between $Euro$ and $Dollar$. Our approach misses such dependencies and, thus, the corresponding correlations. Potentially, H^{Corr} considers transitive dependencies and is able to detect the encountered false negatives. However, the detector is strongly dependent on the correct recognition of computational units. Since recognizing computational units in runtime is difficult, H^{Corr} also missed some transitive correlations. For this reason H^{Corr} exhibits several other false negatives of correlations which our approach correctly identified.

```
Thread A
    Euro = 300;
    Yen = Euro * 107;
    Dollar = Yen/97;
end
```

Fig. 14. Transitive data dependency from $Euro$ to $Dollar$.

Falsely recognized parental correlations are a source of false positives: Closely written variables, which share the same parent are, contrary to our assumption, not correlated. An example is the variable for a person's hair and the variable for a person's size which are written subsequently but do not share any logical dependency. We did not observe false positives caused by correlations identified by other patterns. However, we expected this result: The other patterns require direct data dependencies between the values of the corresponding

Program	# Corr. Violations	Our tool FN	Our tool FP	H^{Corr} FN	H^{Corr} FP
Micro Benchmarks	209	33	23	36	59
Synch.	0	0	0	0	0
Unsynch.	30	10	0	12	6
Locked	0	0	0	0	0
Gaps	30	10	0	16	4
Corr.	38	7	3	8	13
KeyPassLib.	21	9	5	7	6
PetriDish	12	7	2	9	1
STP	33	14	3	17	2
Total	**373**	**90**	**36**	**114**	**91**

TABLE IV. NUMBER OF FALSE POSITIVES AND FALSE NEGATIVES REGARDING VIOLATED CORRELATIONS.

variables. This naturally leads to a correlation between those variables. During evaluation we even observed direct control dependencies between two variables to necessary lead to a correlation. The false positives of H^{Corr}, on the other hand, come from computational units that the approach has estimated their scope to be too large. Therefore, it reports correlations that have been unaffected by the race condition.

VII. CONCLUSION

In this paper, we introduced a general race detection approach considering correlated variables. Our race detector relies on a novel and precise concept of non-deterministic reads. It enables us to identify race conditions, including those that normally escape race detectors which ignore variable correlations. In contrast to earlier approaches for correlated variables, our approach does not rely on the successful identification of correlations. This makes it more reliable than previous methods. Finally, our detector precisely recognizes violated correlations at runtime. This enables developers to better judge the effects of non-determinism in their programs. In the future, we want to enhance our concept for identifying correlations by considering transitive data dependencies and static correlation detection. This enables us to take steps toward identifying logical dependencies, further reducing our number of false negatives. Furthermore, we want to reduce the number of false positives our detector reports by excluding *intentional* non-deterministic reads.

REFERENCES

[1] Yang, C.S.D., Souter, A.L., Pollock, L.L.: All-du-path coverage for parallel programs. SIGSOFT Softw. Eng. Notes **23**(2) (March 1998) 153–162

[2] Jannesari, A., Westphal-Furuya, M., Tichy, W.F.: Dynamic data race detection for correlated variables. In: Proceedings of the 11th international conference on algorithms and architectures for parallel processing - Volume Part I. ICA3PP'11, Berlin, Heidelberg, Springer-Verlag (2011) 14–26

[3] 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, New York, NY, USA, ACM (2008) 231–240

[4] 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 (2007) 103–116

[5] 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, New York, NY, USA, ACM (2008) 329–339

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

[7] Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. PLDI '09, New York, NY, USA, ACM (2009) 121–133

[8] Artho, C., Havelund, K., Biere, A.: Using block-local atomicity to detect stale value concurrency errors. In: Proc. ATVA 04, Springer (2004)

[9] Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '04, New York, NY, USA, ACM (2004) 256–267

[10] Musuvathi, M., Qadeer, S.: Chess: systematic stress testing of concurrent software. In: LOPSTR'06: Proceedings of the 16th international conference on Logic-based program synthesis and transformation, Springer-Verlag (2007) 15–16

[11] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the 8th USENIX conference on operating systems design and implementation. OSDI'08, Berkeley, CA, USA, USENIX Association (2008) 267–280

[12] 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, New York, NY, USA, ACM (2006) 334–345

[13] Jannesari, A., Bao, K., Pankratius, V., Tichy, W.F.: Helgrind+: An efficient dynamic race detector. In: Proceedings of the 23rd international parallel and distributed processing symposium (IPDPS'09), Rome, Italy, IEEE (2009)

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

[15] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (July 1978) 558–565

[16] Keith D. Cooper, Timothy J. Harvey, K.K.: A simple, fast dominance algorithm

[17] Microsoft: Code gallery for parallel programs. http://code.msdn.microsoft.com/Samples-for-Parallel-b4b76364

[18] Butler, N.: Petridish: Multi-threading for performance in c#. http://www.codeproject.com/Articles/26453/PetriDish-Multi-threading-for-performance-in-C

[19] Reichl, D.: Keepass password safe. http://keepass.info/

[20] : Smart thread pool. http://smartthreadpool.codeplex.com/

[21] Intel. http://software.intel.com/en-us/intel-inspector-xe Intel Inspector XE 2013.

[22] Szeder, G.: Unit testing for multi-threaded java programs. In: PADTAD '09: Proceedings of the 7th workshop on parallel and distributed systems: testing, analysis, and debugging, ACM (Jul 2009)

[23] Nistor, A., Luo, Q., Pradel, M., Gross, T.R., Marinov, D.: Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code. In: Proceedings of the international conference on software engineering. ICSE 2012

[24] Schimmel, J., Molitorisz, K., Jannesari, A., Tichy, W.F.: Automatic generation of parallel unit tests. In: Automation of software test (AST), 2013 8th International workshop on. (May 2013)