7th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2014)
Amsterdam, Netherlands
July 3-4, 2014

Unit Tests for Correlated Variables in Multithreaded Code

Ali Jannesari[1,2] · Felix Wolf[1,2]

Abstract A notorious class of concurrency bugs is the race condition on correlated variables which makes up about 30% of all concurrency non-deadlock bugs. A solution to prevent this problem is the automatic generation of parallel unit tests. This paper presents an approach to generate parallel unit tests for variable correlations in multithreaded code. We introduce a hybrid approach for identifying correlated variables. Furthermore, we estimate the number of potentially violated correlations for methods executed in parallel. In this way we are capable of creating unit tests which are suited for race detectors considering correlated variables. We were able to identify more than 85% of all race conditions on correlated variables of 8 applications by applying our parallel unit tests. At the same time, we reduced the number of generated unit tests by up to 50% while maintaining the same precision and accuracy in terms of race detection.

Keywords Unit tests, automatic testing, parallel programming, debugging, race detection, program analysis, correlated variables.

1 Introduction

Nowadays, unit testing plays a major role in the sphere of software development. Software may consist of billions of lines of code and a full error analysis can be very time consuming and is often unnecessary. Normally, only new and modified code regions have to be tested. For this reason developers create unit tests with which small parts of the program can be effectively tested without executing redundant code regions to find new bugs. This is even more helpful when dealing with multithreaded software and concurrency bugs. The same bug in multithreaded software can have different behavior for different thread interleavings making the debugging of multithreaded software hard and ex-

[1] German Research School for Simulation Sciences, Aachen, Germany
[2] RWTH Aachen University, Aachen, Germany
{a.jannesari, f.wolf}@grs-sim.de

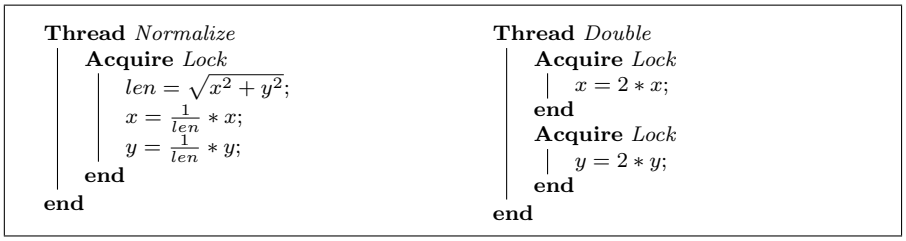


Fig. 1: A high-level data race violating the semantics of the vector (x, y).

pensive. A remarkable type of unit test to address this is the *parallel unit test*, which focuses on unit testing to find concurrency bugs.

To the best of our knowledge there exists no work which covers parallel unit test creation for data races on correlated variables. There is no support for race detectors considering correlated variables in their analysis when dealing with unit tests. Studies showed that over 30% of all race conditions involve correlated variables [1]. Also, due to the non-determinism of the thread scheduling, such races are in general hard to reproduce. This further necessitates the need for unit tests covering race conditions on correlated variables.

In this work we want to combine the benefits of automatic parallel unit test generation with the advantages of race detection considering correlated variables. In order to realize this, our work is based on the existing unit test generator AutoRT [2]. In the scope of this paper we introduce an extension called CorrRT which enhances AutoRT by identifying possibly violated correlations of method pairs. The higher the number of correlations found, the higher the probability that a potential race condition violates a variable correlation.

We automatically generated 100 parallel unit tests for correlated variables on eight different applications. A race detector for correlated variables, H^{Corr} [3] , analyzing the unit tests reported more than 85% of the race conditions violating variable correlations. Furthermore, we were able to reduce the number of redundantly generated tests by up to 50% in comparison to the AutoRT approach.

2 Background

In this section we introduce terms which we use in the scope of this paper. Also, we present some basic techniques our algorithms apply and explain how we define correlations between variables.

2.1 High-level Data Races

In our work, we define a race condition as an anomalous behaviour of a program due to a variable's value unexpectedly depending on the scheduling of threads. We divide race conditions into *high-level* and *low-level* data races, according to whether we need a semantic understanding of the code for identifying the race. For low-level data races we can neglect semantics. A low-level data race occurs when two concurrent threads access a shared variable without synchronization and when at least one of these accesses is a write operation.

A high-level race condition can be harder to detect. Generally, when the anomalous behaviour of a race condition is caused by a violation of the underlying program semantics and if the anomaly is not a low-level data race, we speak of a high-level data race. Figure 1 gives an example for such a high-level data race. All accesses have been secured by locks. However, if the runtime normalizes the vector in between the doubling operation, the values of x and y are not correctly tuned to each other any more. We recognize that the semantics of those variables have been violated.

As was seen in the given example, one kind of program semantics are the semantic relationships between variables and their values. The violation of these relationships by an anomaly, a race condition, is considered a high-level data race. Our work aims to detect this kind of high-level data race by applying the concept of variable correlations to a parallel unit test generation approach.

2.2 Variable Correlations

Two variables are correlated if their values are, or are meant to be, in a semantic relationship during the whole program execution. We already introduced an example of a variable correlation in figure 1, where the two variables x and y constitute a vector. In figure 2 we show another example for a variable correlation. We can see a function maintaining two variables $Euro$ and Yen which contain the same value in different currencies. The code implies a semantic relationship between these two variables (both variables depend on $euValue$). Obviously, this relationship may be violated when these two variables do not express the same value, in different currency, anymore.

Function *Currency(int euValue)*
  $Euro = euValue$
  $Yen = euValue * 107.201;$
end

Fig. 2: Two correlated variables $Euro$ and Yen.

2.3 Parallel Unit Tests

Unit testing has become a common practice in the field of software engineering. The idea of unit testing is to concentrate debugging on small parts at a

```
Function Parallel Unit Test()
    // Context
    // Initializing objects and variables...
    // Concurrently invoke a method pair
    Thread1.Start(Method1);
    Thread2.Start(Method2);
    // Wait for the methods to finish
    Thread1.Wait();
    Thread2.Wait();
end
```

Fig. 3: The general structure of a parallel unit test.

time instead of the whole program. This promises better precision and shorter testing times since bug detection can be focused on the relevant code without analyzing and/or executing the whole program. A unit test verifies the correctness of the corresponding program part and informs us when anomalous behaviour has occurred. For this verification we have to execute the unit test. During execution, the program part to be tested is invoked and the results gained are compared to the expected results.

Parallel unit tests are a special class of unit tests which distinguish themselves in the following way:

1. A parallel unit test contains the parallel invocation of two methods, a method pair.
2. It should not be executed directly but is intended to be analyzed by tools for concurrency bug detection.
3. The parallel unit test remains independent concerning execution. This means, it can be executed without any additional support. This is an important feature for dynamic concurrency bug detection tools which need to execute the code for analysis.

Figure 3 illustrates the generic structure of a parallel unit test, divided into three parts: Initializing the necessary context, concurrently invoking the methods and synchronizing with the main thread. Note that the parallel unit test does not include assertion statements or the like: Bug detection is realized only by the tools analyzing and/or executing the parallel unit test. As an example, a race detector can analyze and/or execute the code region specified by the parallel unit test in order to identify race conditions.

The generation of unit tests, and especially parallel unit tests, bears a crucial challenge: Gaining sufficient test coverage without generating redundant or irrelevant unit tests. A unit test generator runs into the danger of either neglecting relevant test cases or generating false positives, like test cases which cannot happen during real program execution.

3 Related Work

We present some recent works focusing on the identification of correlated variables and the automatic generation of parallel unit tests.

MUVI [4] is a hybrid race detector for correlated variables. The algorithm recognizes correlations among variables by applying a static analysis which uses data mining techniques. It identifies accesses to variables in the code which commonly happen in the same method and occur relatively close to each other. The approach assumes semantic relationships between these identified accesses and considers their variables to be correlated. However, MUVI does not consider data or control dependencies during variable correlation detection.

$Helgrind^+$ for correlated variables (in short H^{Corr}) [3] is a dynamic race detection approach. It is based on the dynamic race detector $Helgrind^+$ [5], [6], [7], [8] for single variables. Like its predecessor, H^{Corr} is a tool developed for the virtual execution environment Valgrind [9]. The approach uses a dynamic analysis to identify code regions which constitute units of computation. A computational unit expresses a sequence of variable accesses forming an atomic computation. With the help of these computational units, H^{Corr} identifies sets of correlated variables in the following way: In the beginning, each variable has its own set. Inside one computational unit, whenever there is a data or control dependency established between two variables of different sets, the sets are merged. Thus, bigger sets of variable correlations are formed.

The approach has two main weaknesses: It relies heavily on the identification of computational units, by considering a specific write and read access pattern. This pattern can just heuristically identify units of computation and tends to either measure them too broad, including accesses that do not belong to that computation, or measure them too narrowly, excluding relevant accesses. Obviously, this has a big impact on the precision for identifying correlations. Another issue is that the approach identifies two variables to be correlated as soon as it identifies one data or control dependency between their values. Even if these dependencies are of a temporary nature and do not constitute a real semantic relationship between the variables.

ConCrash [10] uses static as well as dynamic approaches to reactively generate unit tests for a given program. First the algorithm performs a static race detection in order to identify race conditions inside methods. In the next step, the concerned methods are instrumented: They track the program state and the scheduling of threads during execution. A subsequent dynamic analysis on these methods captures this information whenever the program throws an exception. Finally, ConCrash uses the captured information to generate unit tests for the concerned methods.

Katayama et al. [11] explain an approach for the automatic generation of unit tests for parallel programs. The approach uses the Event InterAction Graph (EIAG) and Interaction Sequence Test Criteria, ISTC. EIAG represents the behaviour of parallel programs. It consists of Event Graphs and interactions. An Event Graph is a components control flow graph of a parallel program. The interactions, in turn, represent synchronizations between

threads. The ISTC criteria are based on the sequences of interactions and reduce the number of unit tests which the EIAG provides.

Musuvathi et al. [12] use reachability graphs to generate unit tests for parallel programs. In order to avoid a state explosion (of unit tests) they introduce four different techniques to prioritize and topologically sort code regions. This approach was implemented and evaluated in combination with the model checking approach *Stubborn Set Method*. The approach is very effective on small programs. However, it is not scalable regarding highly parallel programs.

A. Nistor et al. [13] generate parallel unit tests for randomly selected public class methods. The approach appends complex sequential code to the unit tests in order to increase the precision of concurrency bug detection. Furthermore, they employ a clustering technique to reduce the number of falsely identified bugs. However, the approach only considers some parts of the program for unit test generation and neglects multiple class interactions.

4 Approach

This section presents CorrRT, a parallel unit test generator for high-level data races on correlated variables. The approach essentially combines parallel unit test generation with the identification of correlated variables, in order to obtain highly specialized unit tests for the detection of correlation violations. First, we shortly introduce the method in AutoRT. Thereafter, we describe our own methods and deal with the detailed presentation of our employed analysis.

4.1 CorrRT in Relation to AutoRT

AutoRT is a proactive unit test generator for parallel programs which uses both dynamic and static approaches for program analysis. For a given program the algorithm considers all possible method pairs as candidates for unit testing. In its subsequent generation steps AutoRT filters this candidate set to the most significant method pairs. A method pair is significant if its two methods are parallel dependent on each other and if they are executed in parallel (parallelism) during program execution. The algorithm identifies parallel dependency and parallelism of a method pair in two subsequent analysis:

1. A dynamic analysis checks which method pairs truly run in parallel during program execution. Additionally, it reduces the candidate set to these parallel method pairs.
2. A static analysis operates on the reduced candidate set. The analysis further filters the candidate set to parallel dependent method pairs, i.e. method pairs containing accesses to the same variables.

Having obtained a significant candidate set the test generation approach employs a Capture-and-Replay technique for creating unit tests out of the remaining method pairs. This means AutoRT dynamically records the object

states which are necessary for invoking each method pair in parallel, called the context. After AutoRT has filtered out equivalent contexts, the algorithm begins with the actual unit test creation process. The generator creates a parallel unit test for each method pair and each different context of that pair. Since the Capture-and-Replay technique reconstructs only contexts which actually existed during program execution, the generated unit test cases do not depict situations which never happen during runtime.

As we can see, AutoRT uses a multi-step candidate set reduction technique to filter out irrelevant method pairs for unit test creation. CorrRT aims to support and enhance this process by introducing analysis that is able to analyze this candidate set for method pairs which are likely to contain correlation violations, which equal high-level data races. This information can be used for two purposes. On the one hand, we can further reduce the candidate set and pass it on to the original AutoRT process for unit test generation. As a result, we obtain a set of parallel unit tests which are likely to contain high-level data races. This is useful to reduce the overall unit test generation time which can be very significant for larger applications. On the other hand, we can just pass along the likelihood for a method pair containing a high-level data race. As a result, the developer can easily decide whether a parallel unit test should be analyzed by a race detector considering high-level data races. This is very useful, since race detectors for high-level data races tend to be generally slower than their conventional counterparts.

In the following listing we give an overview of our enhancements of the unit test generation process. Our extension is divided into six parts:

1. We have extended the dynamic parallelism analysis to additionally monitor and protocol encountered shared variables.
2. A static analysis on the control flow of the whole program uses the information to detect correlations which involve shared variables. For this purpose our algorithm uses methods of pattern matching and probability to estimate whether two variables are correlated.
3. For each method pair (which constitutes a unit test candidate) we determine the number of accesses to variable correlations which may be violated by the parallel execution of the method pair.
4. From these recognized correlation accesses, we identify especially endangered correlations.
5. We use the information from the former analysis to compute the correlation rank of the method pair. The correlation rank states the number of accesses to potentially endangered correlations a method pair contains in comparison to its total number of accesses.
6. We create unit tests from method pairs with high correlation ranks only. For this reason we rely on the approach of AutoRT by dynamically recording the object states inside the method pairs and generating unit tests in which we reconstruct the recorded object states and invoke the method pair in parallel.

The result of the steps above is a set of unit tests which are especially suitable for race detectors considering correlated variables. The following subsections explain the most important enhancements of our approach in detail.

4.2 Correlation Patterns

For identifying correlations between variables we perform a static analysis of the given program. Our approach is based on the concepts of H^{Corr} [3] and MUVI [4]. According to H^{Corr} variables are correlated if they become data and/or control dependent on each other during a computational unit. This implies a strong relationship between these dependencies and variable correlations. Further on, MUVI assumes that variables which are accessed relatively often near to each other, are with a high probability correlated to each other.

We combine both ideas. As a result, we consider variables which are relatively often data and control dependent on each other to be correlated. Therefore, we identify correlations on the basis of predefined patterns which indicate strong data and control dependencies between the variables. We use these patterns to find *indications* of correlations between variables. If we recognize enough patterns which support this indication we then regard the variables to be correlated. We introduce five types of patterns: vertical, horizontal, parental, control and chain patterns.

```
Euro = 300;
Yen = Euro * 107;
```

(a) Vertical pattern between Yen and $Euro$.

```
if isPrivate == true then
    Street = "PrivateStreet";
end
```

(b) Control pattern between $isPrivate$ and $Street$.

```
Person.Street = "PrivateStr.";
Person.StreetNo = 107;
```

(c) Parental pattern between $Street$ and $StreetNo$.

```
Minutes = Seconds/60;
Hours = Seconds/3600;
```

(d) Horizontal pattern between $Minutes$ and $Hours$.

```
Euro = 300;
Yen = Euro * 107;
Dollar = Yen * 0.012;
```

(e) Chain pattern between $Euro$ and $Dollar$.

Fig. 4: Examples for correlation patterns.

Vertical pattern: After the execution of an assignment, the written variable is data dependent on each read variable. Data dependent values are in a logical relationship, since we generated one value from the other. Therefore, a data dependency and thus a variable assignment also indicates a correlation between the written and the read variable. We call this correlation a *vertical correlation*. In Figure 4a we can observe the vertical correlation pattern between $Euro$ and Yen. Since the first assignment does not contain any read variable we do not detect any correlation in it.

Control pattern: Variables written inside a control flow branch become control dependent on the variables read inside the branching condition. Just like a data dependency, a control dependency also infers a correlation between the corresponding variables: Their values become logically related to each other. Therefore, we search the code for write accesses inside control flow branches with branching conditions reading variables. Each written variable becomes correlated with the read variable, analogous to a variable assignment. Figure 4b shows an example of a control pattern. Here we see that the value of $Street$ is dependent on the evaluation of the $isPrivate$ value.

Parental pattern: 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 to be higher if the two variables are written subsequently, as they tend to belong to the same computation. Figure 4c shows the variables $Street$ and $StreetNo$ as correlated by a parental pattern.

Horizontal patterns infer implicit data dependencies from multiple variable assignments. If two variables are data dependent on the same variable at the same time, we consider these two variable's values to be in a logical relationship which is independent of the variable they data depend on directly. Therefore, we search for two subsequent assignments on two different variables which share a same read variable. Figure 4d shows an example of a horizontal pattern. Note that each horizontal pattern also includes two vertical patterns. Therefore, we do not only recognize a correlation between $Minutes$ and $Hours$ but also between $Seconds$ and $Minutes$ as well as $Seconds$ and $Hours$.

The *chain pattern* considers transitive data dependencies through identifying chain assignments. Those are assignments in which the assigned variable is used as a read variable in the subsequent assignment. A chain assignment indicates that an assigned variable is correlated to the previously assigned variables as well as the subsequently assigned variables in the chain. Figure 4e depicts an example of a chain assignment. The transitive data dependency recognized through the chain assignment indicates a correlation between $Euro$ and $Dollar$.

The probability of a correlation between two variables being real is the ratio of write accesses inside a correlation pattern, indicating such a correlation, to the total amount write accesses on the variables being considered. When a variable gains a value which is logical dependent on another variable's value, this write access supports the claim that the two variables are correlated. However, if the value has no such logical dependency it opposes this claim. For example in Figure 5, a vertical pattern in function A indicates a correlation between $Euro$ and Yen. Furthermore, function C contains a parental pattern indicating the same correlation. However, there is a write access on Yen in

function B which does not support this claim. Therefore, the probability that $Euro$ and Yen are correlated is about 67%: Two out of three write accesses on Yen belong to a correlation pattern.

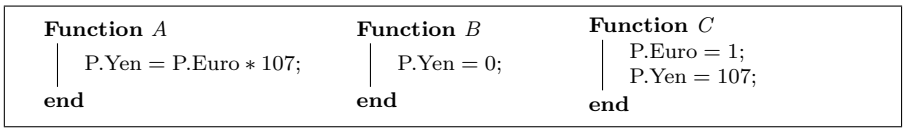


Fig. 5: Function A and C indicate a correlation between $Euro$ and Yen, function B does not.

We only have to identify correlations which involve at least one shared variable. Trivially, a correlation consisting only of local variables cannot be violated by atomicity violations: There is just one thread accessing the participating variables. Of course, two correlated variables which are shared can potentially be involved in an atomicity violation. But this also counts for a correlation between a shared and a local variable. When another thread changes the value of the shared variable inappropriately, the values of the correlated variables loose their logical relationship. As a result, the program may behave unexpectedly. Figure 6 shows the execution of two threads where thread B violates the correlation between the shared variable $Euro$ and the local variable Yen. On the assumption that $Euro$ and Yen are correlated we expect that thread A stores in $Euro$ an amount of money in the currency of Euro and subsequently stores in Yen the same amount in the currency of Yen. When the scheduler executes thread B between the two assignments this logical relationship is lost.

In order to solve the issue with the violated correlation, we can execute the two logically related operations of thread A under one continuous lock. This will remove the possibility of thread B executing between the two assignments so that the final results remain correct.

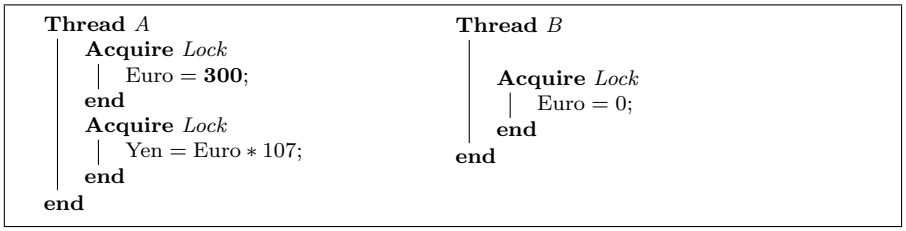


Fig. 6: A violated correlation between the shared variable $Euro$ and the local variable Yen.

4.3 Correlation Accesses

After we have identified the correlation patterns inside the program, we analyze for a method pair for which correlations may be violated when the two methods are executed in parallel. We firstly consider the accessed variables inside the methods. For further analysis, we only regard accesses to variables which indeed are accessed by several threads and by the method pair, where at least one of the methods must write the variable. In other words: We only consider accesses to shared and strongly parallel dependent variables for each method. We take the information about the shared variables from the formerly executed dynamic parallelism analysis. For obtaining strongly parallel dependent variables for one method m_1 we statically compare all accesses of that method with the write accesses inside the other method m_2 of the pair. The set of variables which both methods access in this way are the strongly parallel dependent variables of m_1. Analogously, we identify the strongly parallel dependent variables of m_2. Furthermore, we filter the accesses on variables which are not correlated to other variables. In this way, we gain the accesses on correlated variables which can potentially be violated during the parallel execution of the two methods.

For each method of the pair we determine the accessed correlations. A method accesses a correlation if and only if it contains accesses to both variables of the correlation. This means one access to a correlated variable is not enough to count as an access to a correlation. Henceforth, one variable access may belong to more than one correlation access. On the assumption that $Euro$ and Yen are correlated Figure 7 contains four correlation accesses. The two read accesses (1), the two write accesses (2), the write access on $Euro$ and the read access on Yen (3) as well as the read access on $Euro$ and the write access on Yen (4) are correlation accesses.

```
Function A
    Output(Euro);
    Output(Yen);
    ...
    Euro = 300;
    Yen = Euro * 107;
end
```

Fig. 7: A function containing four accesses to the correlation between $Euro$ and Yen.

We determine the correlation accesses for both methods in the method pair separately. The set of correlation accesses of the method pair equals the union of the correlation accesses of both methods. We consider these the accesses to the correlations which can potentially be violated during parallel execution of the pair.

4.4 Endangered Correlations

For each correlation access determined in the previous step, we further investigate how endangered the corresponding correlation is. Therefore, we consider the synchronisation instructions inside the methods of the pair. Correlated

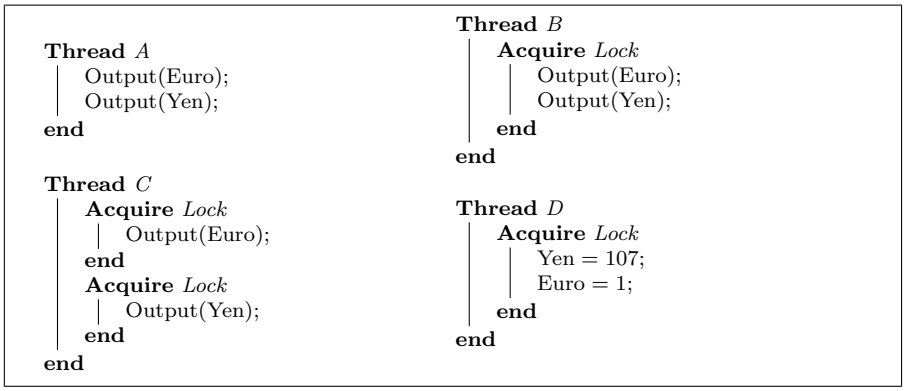


Fig. 8: Three parallel reading threads and one parallel writing thread: Only in thread B is the correlation between $Euro$ and Yen not violated.

variables should be accessed in atomic regions. Therefore, if we encounter accesses to a correlation which are separated by a synchronization instruction, the accessed correlation is especially endangered. If we do not detect any synchronization instruction in between the correlation access, we can assume that the accesses on the variable are either fully protected or not protected at all. This means we either do not have a correlation violation or we encounter a low-level race condition instead of an atomicity violation. Figure 8 illustrates this reasoning. Thread A does not execute a synchronization in between the accesses on the correlated variables $Euro$ and Yen. The accesses are in fact not synchronized in any form. Therefore, we have a low-level race condition with the execution of thread D. Also thread B does not execute any synchronization instruction in between the access on the correlation. We can see that the correlation is not violated in B since a lock continuously protects the read accesses to $Euro$ and Yen. Finally, only the parallel execution of thread C and D yield a high-level data race excluding a low-level data race.

4.5 Correlation Rank

The correlation rank is a metric for our approach to decide whether a method pair should be used for unit test generation. Therefore, it is important not only to consider the number of correlation accesses but also to regard the probability of a correlation violation. The number of endangered correlation accesses gives a quantitative as well as a qualitative statement about the accesses to correlations inside the method pair. In the following we will refer to the sum of the number of endangered correlations as the correlation amount of a method pair.

We consider the correlation amount relative to the accesses on uncorrelated variables and variables, whose correlations cannot be affected by the parallel

execution of the method pair. We call these accesses the uncorrelated amount of the method pair. If a method pair consists of an uncorrelated amount to a great degree in comparison to the correlated amount, we should not consider the method pair to be suitable for a unit test for correlated variables: The method pair is more likely to contain concurrency bugs, which do not involve correlated variables. Due to this reasoning we defined the following metric for the correlation rank R_C, with C_A as the correlation amount and U_A as the uncorrelated amount:

$R_C = \frac{C_A}{C_A + U_A}$

A high rank indicates a high number of accessed correlations and/or especially endangered correlations. When the correlation amount of a method pair is zero, the correlation rank equals 0%. Analogously, an uncorrelated amount of zero equals a correlation rank of 100%. Thus, a method pair with a high correlation rank has a higher probability of containing high-level data races resulting from correlation violations.

After we have ranked all method pairs we pass only those with a high correlation rank to the dynamic object recording. For these pairs we generate unit tests in which we reconstruct the recorded object states and invoke both methods in parallel.

4.6 Example

For a better understanding of our concepts, we present an example of our presented approach on a small program. The parallelism analysis and the parallel dependency analysis have identified four functions used as parallel unit test candidates (shown in figure 9). Furthermore, we identified the method $RefreshSecs$ to be parallel to the three other methods $SetMins$, $SetHours$ and $SetTime$. The variables $Hours$, $Mins$ and $Secs$ are shared. We perform the correlation pattern analysis on the control flow of the program. We identify a vertical pattern between $Hours$ and $Secs$ inside the method $SetHours$. Analogously, we detect a vertical pattern between $Minutes$ and $Secs$ inside $SetMins$. Moreover, inside the method $SetTime$, we can identify a horizontal pattern between $Minutes$ and $Hours$. The horizontal pattern spans both invoked methods $SetMins$ and $SetHours$. Since there are no other write accesses to the given variables inside the program, we calculate a 100% certainty for each indicated correlation.

In the next step, we determine the correlation accesses and the endangered correlations for each of the four methods. For this reason, we consider the parallel method pairs ($SetMins$, $RefreshSecs$), ($SetHours$, $RefreshSecs$) and ($SetTime$, $RefreshSecs$). $SetMins$ contains exactly one correlation access, but since there is a continuous lock surrounding the considered variable accesses the correlation is not endangered. The same holds for the method $SetHours$. $RefreshSecs$ on the other hand does not contain a correlation access. However, $SetTime$ includes three correlation accesses: The previously mentioned correlation accesses inside both invoked methods and the access

```
Function SetMins()
    Acquire Lock
        Mins = Secs/60;
    end
end

Function SetTime()
    SetMins();
    SetHours();
end

Function SetHours()
    Acquire Lock
        Hours = Secs/3600;
    end
end
Function RefreshSecs()
    Acquire Lock
        Secs = SysTime();
    end
end
```

Fig. 9: Four methods accessing three shared variables. $SysTime()$ constitutes a system internal method whose body is not visible.

Program	**B. Account**	**B. Queue**	**Dekker**	**Order Sys.**	**Corr Sys.**	**Petri-Dish**	**Key Pass**	**STP**	**Sum**
LOCs	25	31	15	360	480	1070	1240	1120	-
Methods	4	6	3	7	18	35	58	46	-
Depth Call Stack	3	2	2	4	5	16	8	12	-
Heap Objects	1	24	2	56	59	371	98	73	-
Threads	2	3	3	5	5	7	16	4	-
Parallel Methods	4	6	3	5	10	35	58	37	-
Parallel Method Pairs	4	14	5	15	27	230	478	315	-
Detected Corrs	3	4	1	5	5	16	13	9	**46**
Low-Level Races	1	3	0	20	10	3	5	11	**63**
High-Level Races	1	2	2	25	5	2	5	4	**46**
Detected: CHESS	0	0	0	0	0	2	2	6	**10**
Detected: H^{Corr}	1	2	2	21	5	2	3	3	**39**

Table 1: Evaluation results of CorrRT.

to the correlation between $Mins$ and $Hours$. Furthermore, this correlation is also endangered since there is no continuous lock protecting the variable access to $Mins$ and $Hours$. As a result the method pair ($SetTime$, $RefreshSecs$) acquires a correlation rank of 100% while each of the other pairs have a correlation rank of 0%. Therefore, we only consider ($SetTime$, $RefreshSecs$) for parallel unit test creation and dismiss the other method pairs. While AutoRT would have generated unit tests for all four method pairs, our approach was able to reduce the candidate set to only one method pair. That is, the only method pair containing a (high-level) data race for correlated variables. By

this approach, we have eliminated redundant unit tests which do not contain potential data races, particularly unit tests which do not contain data races on correlated variables.

5 Implementation

We implemented the approach in the managed environment .NET C#. For data and control flow analysis as well as the code instrumentation we employed the Common Compiler Infrastructure (CCI) framework. Therefore the presented analysis works on the Common Intermediate Language (CIL) which underlies every .NET program.

The dynamic shared variable analysis monitors the encountered field variable accesses. We identify each field variable by its unique field identifier (acquired from the CCI framework) and the hash code of its parent object.

For the correlation pattern detection we need to identify assignments and control flow branches. CCI already provides analysis data structures for detecting assignments. However, control flow branch analysis is not supported by the framework. Therefore, we identify the scope of control flow branches via post dominator analysis and apply a simple and efficient algorithm, which was presented in [14].

Detecting endangered correlation accesses requires the identification of synchronization instructions. In .NET 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 by their distinctive namespace: $System.Threading$. All methods belonging to that namespace manage synchronization operations between threads. Also, our analysis does not distinguish the kinds of synchronization, it is therefore able to identify synchronization instructions in general.

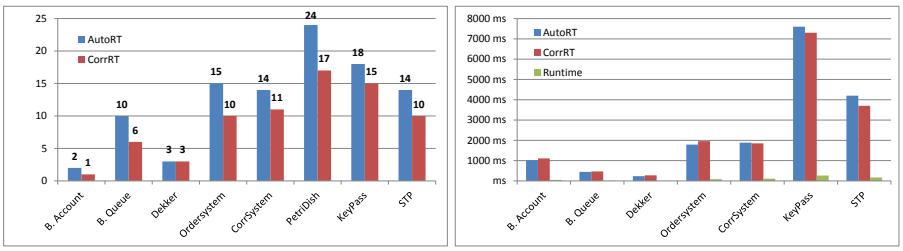
In our implementation we consider variable correlations with a probability of 50% and above to be true. Furthermore, we only generate unit tests for method pairs with a correlation rank above 30%.

6 Evaluation

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

6.1 Test Environment

We use sample programs as well as real-world applications for our evaluation purposes. The race detector CHESS [15], [16] provides small program examples containing high-level data races. For our evaluation we used the programs *Bank Account*, *BoundedQueue* and *Dekker* from the CHESS examples. MSDN Code Gallery [17] contains applications which demonstrate



(a) Number of generated unit tests (b) Total generation time

Fig. 10: Comparison between AutoRT and CorrRT

the functionality of parallel programming in .NET. We chose an order-system simulation (a master thread manages many worker threads executed concurrently) from MSDN. We additionally implemented an alternative version of it containing correlated variables for each correlation pattern. Furthermore, we evaluated the open source programs PetriDish [18], the program library of KeyPass [19] and SmartThreadPool (STP) [20]. The programs are listed in Table 1.

We evaluate our unit test generation approach according to three metrics: The efficiency for the race detection, the performance and the search space reduction of all possible unit test candidates.

For the efficiency of the race detection we take the number of detected high-level data races in relation to the total number of high-level data races inside the program into account. We expect that our test cases are especially suited for high-level data races only and therefore do not contain many low-level data races. Hence, we compare the ratio of found low-level data races with the ratio of high-level data races. For these purpose we use Microsoft Research CHESS and H^{Corr} as race detectors. H^{Corr} is able to detect race conditions on correlated variables automatically. For CHESS the user has to specify correlations inside the given program. We did not provide CHESS with any information about the variable correlations, hence it can only detect low-level race conditions. By this, we ensure that the race conditions involving correlated variables inside the programs cannot be found by race detectors considering low-level race conditions only.

Besides the race detection, the search space reduction is one of the main metrics of our evaluation. Filtering out unit test candidates without potentially violated correlations is a major purpose of the presented approach. By comparing the number of generated unit tests by AutoRT and CorrRT, we can directly measure the effectiveness of our method.

For the performance, we consider the overall unit test generation time of our approach and qualify it to the unit test generation time of AutoRT.

6.2 Race Detection Efficiency

Our unit test generator created 81 parallel unit tests for the eight evaluation programs. The programs contained 46 race conditions on correlated variables, of which H^{Corr} found a total of 39 when applied to our generated unit tests. We observed that this number of missed data races was caused by the inaccuracy of H^{Corr} itself: The generated unit tests contained all race conditions on correlated variables but the race detector was not able to detect all of them inside the tests. The reason for this behaviour is the detection of computational units which the race detector identified as too short. As a result H^{Corr} missed some correlations between the variables and did not detect the corresponding high-level races. Still, H^{Corr} was able to overall detect more race conditions of the programs when analysing the unit tests than analysing each program as a whole. From the 46 races H^{Corr} could only detect 31 when applied directly to the programs (instead of the 39 it detected when applied to the generated unit tests). We observed that H^{Corr} is generally more precise on small programs. In this case, our parallel unit tests for correlated variables even resulted in an increased precision of 11%.

CHESS detected a total of 10 data races when analysing the unit tests. This means our generated unit tests are indeed specialized on correlated variables. From the total of 46 low-level race conditions inside the programs only 10 were captured by our generator. These low-level races resided in method pairs which also contain race conditions on correlated variables. In the generated unit tests for the corresponding method pairs they, therefore, appear as a by-product.

6.3 Test Case Reduction

Figure 10a shows the number of generated unit tests by AutoRT and CorrRT. Compared to the original AutoRT we observed a reduction of generated unit tests up to 50%. On average, we were able to reduce the number of redundant unit tests about 20% compared to the number of generated parallel unit tests by AutoRT. Whether our approach is able to reduce the search space depends highly on the distribution of accesses on correlated and uncorrelated variables inside the program as well as the number of endangered correlations of course. We observed the best reduction on programs in which methods either access only uncorrelated or only correlated variables. This is because our approach can safely exclude method pairs which do not contain accesses to correlated variables. Otherwise, the intersection of method pairs which potentially contain race conditions on correlated variables and method pairs which contain data races in general is higher.

6.4 Time Overhead

The time for unit test generation as seen in Figure 10b is a sum of different partial times including the static parallel dependency analysis, the static corre-

lation analysis, the dynamic parallelism analysis and the dynamic object state recording. We have experienced that the most critical performance impact lies in the dynamic analysis. Multiple executions of the same program code and expensive object recording cause a major slow down. The ratio between the overall unit test generation time and the execution time of the program varies wildly between a factor of 16 and 266. Big programs with many objects like PetriDish cause a high state recording time. The static correlation analysis, only takes a small part of the overall generation time. On average our additional analysis took about 15% of the total generation time. The analysis time varies from 25% to less than 5% depending on the execution of the program. Though we perform an additional analysis on the code and the method pairs, the generation time of our implementation is negligible compared to the whole generation time. Additionally, because of the unit test candidate reduction we partially experience an overall less generation time on comparatively large programs such as PetriDish, KeyPass and STP.

7 Conclusion

In this paper we introduced an approach which enhances automatic parallel unit test generation for correlated variables. Our approach is able to identify highly correlated regions which are especially vulnerable to concurrency bugs. This reduces the search space in unit test generation and uncovers code suited for our test analysis approach. During our evaluation we detected more than 85% of the race conditions involving correlated variables inside eight different applications by using our generated parallel unit tests.

In the future, we want to further optimize the static correlation analysis. We may apply heuristics for identifying correlated variables for an even bigger coverage. We may use data mining approaches to infer correlations in addition to data or control dependencies. Moreover, the region hypothesis established in [21] and advanced in [3] can also supply correlations between more than two variables.

Also, we want to pass the results of our correlation detection to the race detectors executing the generated parallel unit tests. This would be especially useful for detectors which normally rely on the user intervention for correlation specifications e.g. [15] or [22]. However, race detectors with automatic correlation detection may profit from a reduced performance overhead and increased precision by our preceding correlation analysis.

Generating parallel unit tests for correlated variables explores new directions for other future work. Generally, different race detectors vary in their effectiveness in detecting specific kinds of concurrency bugs. Even detectors for correlated variables vary in precision depending on the structure of the code. Therefore, as a next step we want to provide analyses and metrics which state how method pairs are suited for specific race detectors. With that information, we do not have to restrict ourselves to correlated variables but may also consider detectors for low-level data races or high-level atomicity viola-

tions. As a result we would be able to tell which race detector is most suited for executing a parallel unit test.

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, New York, NY, USA, ACM (2008) 329–339
2. Schimmel, J., Molitorisz, K., Jannesari, A., Tichy, W.F.: Automatic generation of parallel unit tests. In: 8th IEEE/ACM International Workshop on Automation of Software Test (AST). (2013)
3. 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
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, New York, NY, USA, ACM (2007) 103–116
5. Jannesari, A., Tichy, W.F.: Library-independent data race detection. IEEE Transactions on Parallel and Distributed Systems (TPDS) **PP** (2013) 1 – 11
6. Jannesari, A., Tichy, W.F.: Identifying ad-hoc synchronization for enhanced race detection. In: IEEE International Symposium on Parallel Distributed Processing (IPDPS) (2010) 1 –10
7. Jannesari, A., Bao, K., Pankratius, V., Tichy, W.F.: Helgrind+: An efficient dynamic race detector. In: IEEE International Symposium on Parallel Distributed Processing (IPDPS) (2009) 1 –13
8. Jannesari, A., Tichy, W.F.: On-the-fly race detection in multi-threaded programs. In: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging. PADTAD '08, New York, NY, USA, ACM (2008) 6:1–6:10
9. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42** (2007) 89–100
10. Luo, Q., Zhang, S., Zhao, J., Hu, M.: A lightweight and portable approach to making concurrent failures reproducible. In: Proceedings of the 13th international conference on Fundamental Approaches to Software Engineering. FASE'10, Berlin, Heidelberg, Springer-Verlag (2010) 323–337
11. Katayama, T., Itoh, E., Ushijima, K., Furukawa, Z.: Test-case generation for concurrent programs with the testing criteria using interaction sequences. In: Proceedings of the Sixth Asia Pacific Software Engineering Conference. APSEC '99, Washington, DC, USA, IEEE Computer Society (1999)
12. W.E. Wong, Yu Lei, X.M.: Effective generation of test sequences for structural testing of concurrent programs. In: 10th IEEE International Conference on engineering of Complex Computer Systems. ICECCS 2005, Richardson, TX, USA, IEEE Computer Society (2005) 539 – 548
13. 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 2012 International Conference on Software Engineering. ICSE 2012, Piscataway, NJ, USA, IEEE Press (2012) 727–737
14. Keith D. Cooper, Timothy J. Harvey, K.K.: (A simple, fast dominance algorithm)
15. Musuvathi, M., Qadeer, S.: Chess: Systematic stress testing of concurrent software. In Puebla, G., ed.: Logic-Based Program Synthesis and Transformation. Volume 4407 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 15–16
16. 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

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. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. SIGPLAN Not. **40** (2005) 1–14
22. 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