Generating Classified Parallel Unit Tests

Ali Jannesari[1,2], Nico Koprowski[1,2], Jochen Schimmel[3], and Felix Wolf[1,2]

[1] German Research School for Simulation Sciences, Aachen, Germany
[2] RWTH Aachen University, Aachen, Germany
[3] Karlsruhe Institute of Technology (KIT), Germany

Abstract. Automatic generation of parallel unit tests is an efficient and systematic way of identifying data races inside a program. In order to be effective parallel unit tests have to be analysed by race detectors. However, each race detector is suitable for different kinds of race conditions. This leaves the question which race detectors to execute on which unit tests. This paper presents an approach to generate classified parallel unit tests: A class indicates the suitability for race detectors considering low-level race conditions, high-level atomicity violations or race conditions on correlated variables. We introduce a hybrid approach for detecting endangered high-level atomic regions inside the program under test. According to these findings the approach classifies generated unit tests as low-level, atomic high-level or correlated high-level. Our evaluation results confirmed the effectiveness of this approach. We were able to correctly classify 83% of all generated unit tests.

1 Introduction

Today, unit testing is an essential part of software development. A software artifact may consist of billions of lines of code. A full error analysis can be very time consuming and is often unnecessary. Usually, only new and modified code regions have to be tested. For this, developers create unit tests for the considered software. By creating unit tests, small parts of the program can be effectively tested without executing redundant code regions to find new bugs. From unit testing, a new field of research and work has emerged: automatic unit test creation [1]. One remarkable aspect of this work is the parallel unit test generator which focuses on creating unit tests for concurrency bugs.

However, parallel unit tests have to be analysed by external concurrency bug detectors. In general, each of these tools has varying suitability for different classes of concurrency bugs. Today, parallel unit tests do not come with any information on the potentially contained class of bug. Therefore, applying the correct concurrency bug detector is left to the user and mostly results in trial-and-error application.

In this work we present a parallel unit test generator which produces classified unit tests for race detection. We generally distinguish tests by whether they are suited for low-level or high-level race detectors. Additionally, we further classify high-level unit tests according to their suitability for race detectors for correlated variables. In order to realise this, our work builds on the existing unit test generator AutoRT [2] which analyses and creates parallel unit tests from method pairs. In the scope of this paper we want to introduce an extension of this work: AutoRT+. We enhance AutoRT by identifying and analysing possibly violated high-level atomicity inside the method pairs.

For a total of 10 applications AutoRT+ automatically generated and classified 130 parallel unit tests for low-level and 106 parallel unit tests for high-level race detectors. From these 106 high-level tests AutoRT+ classified 52 for race detectors on correlated variables. We analysed the generated unit tests with four different race detectors. During our evaluation we observed that 83% of all unit tests were correctly classified.

2 Background

In this section we introduce terms which we use in the scope of this paper.

2.1 High-level Data Races

In our work, we define a race condition as an atomicity violation when accessing variable values. We further divide race conditions into *high-level* and *low-level* race conditions, according to the number of variables that are part of the data race.

```
Thread Normalize
    Acquire Lock
        $len = \sqrt{x^2 + y^2}$;
        $x = \frac{1}{len} * x$;
        $y = \frac{1}{len} * y$;
    end
end

Thread Double
    Acquire Lock
        $x = 2 * x$;
    end
    Acquire Lock
        $y = 2 * y$;
    end
end
```

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

Low-level races violate the atomicity of a single variable access. 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.

High-level races violate the atomicity between several variable accesses. A high-level race condition is generally harder to detect, since identifying high-level atomicity requires an understanding of the program semantics. Figure 1 gives an example for such a high-level data race: All accesses have been secured by locks. However, if we have the interleaving in which the vector is normalized 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 has been violated.

There are different approaches for identifying high-level atomic regions. In our work, AutoRT+ relies on the following two approaches.

Variable Correlations. Two variables are correlated iff their values are, or are meant to be, in a semantic relationship during the entire program execution [9]. Therefore, accesses to correlated variables form high-level atomic regions. Violations of this atomicity lead to correlation violations: the semantic relationship between the correlated variables becomes violated. We already introduced an example of a variable correlation in figure 1, where the two variables x and y constitute a vector. Here, the high-level atomicity inside *Double* is endangered.

Region Hypothesis. The region hypothesis [3] employs the concept of computational units in order to identify high-level atomic regions. Thereby, a computational unit is the longest sequence of instructions which satisfies the following two conditions:

1. The instructions are data and control dependent on one another. Thus, there exist no independent computations inside a single computational unit.
2. Inside a single computational unit a shared variable is not read after it has been written to.

The concept assumes that a typical atomic operation on a shared data structure consists of three parts: reading, computing and storing. The high-level atomic region to be protected inside a computational unit is called the shared region. It starts with the first access to a shared variable and ends with the last.

2.2 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 time instead of the whole program. This promises better precision and shorter testing times since bug detection can be focussed on the relevant code without analysing or executing the whole program. A unit test verifies the correctness of the corresponding part of the program informs about and reports any anomalous behaviour. For this verification we have to execute the unit test. During execution the program part to be tested is invoked and the results of the invocation are compared to the expected results.

```
Function Parallel Unit Test()
    // Initializing context
    // Concurrent invocation
    Thread1.Start(Method1);
    Thread2.Start(Method2);
    // Wait for methods to
       finish
    Thread1.Wait();
    Thread2.Wait();
end
```

Fig. 2. General structure of a parallel unit test

Parallel unit tests are a special subclass of unit tests which distinguish themselves in the following ways:

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 analysed by tools for concurrency bug detection.
3. The parallel unit test is independent with respect to 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 2 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.

3 Related Work

We present some approaches for the automatic generation of parallel unit tests and race detection approaches used in the scope of this paper. ConCrash [4] uses a static race detection approach to identify methods for unit test generation. The actual generation process is done by employing a Capture-And-Replay technique. ConCrash only considers methods in which race conditions have been found. Katayama et al. [5] 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. Musuvathi et al. [10] use reachability graphs to generate unit tests for parallel programs. The approach proved to be very effective on small programs. However, it is not scalable regarding programs with a large amount of parallelism. A. Nistor et al. [7] 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. The approach only considers some parts of the program for unit test generation and neglects multiple class interactions.

MUVI [8] is a hybrid race detector for correlated variables. The algorithm employs a static correlation detection analysis based on data mining techniques. Subsequently, MUVI executes a dynamic race detection on the program under test. The correlation detection does not consider data dependencies between correlated variables and the race detection cannot identify continuous locks for high-level atomic regions. $Helgrind^+$ for correlated variables (H^{Corr}) [9] is a dynamic race detection approach. Parallel to the race detection the approach dynamically detects correlated variables by identifying computational units. H^{Corr} considers variables correlated if they are accessed in the same computational unit. CHESS [10], [11] dynamically searches a program for concurrency bugs including races, deadlocks and data errors. Through user annotations CHESS is also able to identify high-level races. However, for whole coverage the approach needs to perform a dynamic analysis for each interesting thread scheduling. This naturally leads to comparatively high analysis times for large programs. The Intel Thread Checker is a commercial dynamic race detector for low-level race conditions. It is part of the Intel Inspector [12], which is a known tool for detecting conventional concurrency bugs as well as memory leaks. [13] presents a dynamic race detector (NDR) which is able to detect low-level and high-level races by identifying non-deterministic reads. Thereby, a non-deterministic read is a read access on a value which is written dependent on the scheduling of threads. NDR also dynamically identifies correlated variables by detecting patterns of data and control dependencies. Finally, NDR reports for each found race condition the violated variable correlations.

4 Approach

This section presents AutoRT+, a parallel unit test generator which produces classified unit tests. The approach extends the parallel unit test generator AutoRT [2] by classification analysis techniques. First, we shortly introduce the original AutoRT approach. Thereafter, we present our new methods and describe the new features of AutoRT+.

4.1 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 generation steps AutoRT filters this candidate set to the most significant method pairs and generates unit tests based on them. Figure 3 gives an overview of the approach.

The algorithm identifies significant method pairs in two independent analyses:

1. A static analysis filters the candidate set to parallel dependent method pairs, i.e. method pairs containing accesses to the same variables.
2. A dynamic analysis reduces the candidate set to method pairs which truly run in parallel.

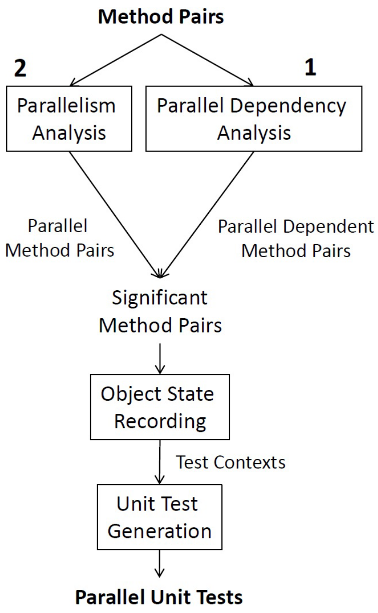


Fig. 3. Overview of AutoRT

Having obtained a significant candidate set, AutoRT employs a Capture-and-Replay technique: It dynamically records the object states which are necessary for invoking each method pair in parallel, called the test context. After AutoRT has filtered out equivalent contexts, the algorithm creates a parallel unit test for each different context of a method 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.

4.2 Overview of AutoRT+

AutoRT+ introduces an approach for producing classified unit tests. It distinguishes between three classes of parallel unit tests: (a) low-level unit tests which are suited for low-level race detectors, (b) high-level atomicity unit tests, which should be analysed by high-level race detectors in general, and (c) high-level correlation unit tests which are suitable for high-level race detectors considering correlated variables.

Figure 4 gives an overview of the unit test generator. Extensions are colored and are detailed in the following:

1. We have extended the dynamic parallelism analysis to protocol the encountered variable identities. This information is used in the subsequent computational unit and correlation detection to improve their precision.
2. The static parallel dependency analysis now also reports the variables which cause the parallel dependency of the method pair.
3. After the parallelism and parallel dependency analysis, we identify all possible computational units of each method pair.
4. We perform a correlation detection analysis considering all identified computational units in the program.
5. For each method pair we determine endangered high-level atomic regions. In order to do this, we consider the identified computational units and the detected correlated variables.
6. Finally, we classify the method pairs according to whether they contain endangered atomic regions and in what way they are endangered.

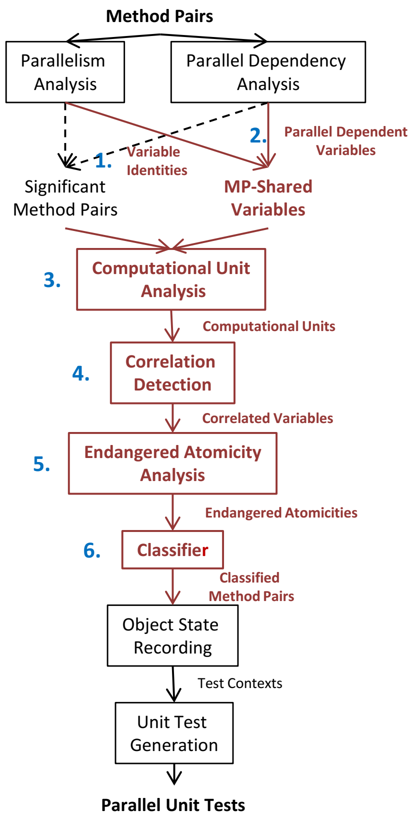


Fig. 4. Overview of AutoRT+

Then, AutoRT+ employs the Capture-and-Replay technique for generating the parallel unit tests. Finally, the class of the unit test is determined by the class of the method pair it is based on.

4.3 Shared vs. Method Pair-Shared (MP-Shared)

A variable is shared iff multiple threads access the variable during program execution. We further differentiate method pair shared (mp-shared) variables.

A variable is mp-shared for a method pair iff each of the two methods in the pair accesses the variable from a different thread. Obviously, an mp-shared variable is also a shared variable but a shared variable is not necessarily mp-shared for every method pair.

In the context of AutoRT+ we can identify mp-shared variables with the help of the dynamic parallelism and static parallel dependency analysis. The mp-shared variables of a parallel method pair are the members of the set of variables on which the two methods are parallel dependent.

4.4 Identifying Computational Units

For a given method pair we perform a static approach to identify the computational units for each method. In order to do so, we need information about the data and control dependencies between instructions and the shared variables. For this reason, we employ an analysis on the method to detect the data and control dependencies between its contained instructions. Further on, for the shared region we do not regard shared variables in general but only the mp-shared variables of the method pair. We use the information gained by the preceding parallelism and parallel dependency analysis in order to determine the mp-shared variables.

By traversing the control flow graph of the method on a specific path we are now able to identify the computational units a method consists of. However, different paths may lead to different computational units. In principle, for all possible computational units we would need to traverse all possible paths. But a method includes an infinite number of possible control flow paths for traversal when it contains loops. Therefore we follow a more relaxed branch coverage, meaning that for every branch in the method there exists a path we traverse which covers that branch. For each encountered instruction on a path we perform the following operations:

```
Function $SetSize(newSize)$
    $SizeFt = newSize;$
    if $SizeFt > 6$ then
        $Big = true;$
    end
    $SizeCm = SizeFt * 30,48;$
    $MethodCount++;$
end
$CU_1 = \{newSize, SizeFt, SizeCm\};$
$CU_2 = \{newSize, SizeFt, Big, SizeCm\};$
$CU_3 = \{Count\};$
```

Fig. 5. All three possible computational units for the method *SetSize*

- An instruction without a data or control dependency is initially assigned its own computational unit.
- The same goes for an instruction which accesses a previously written mp-shared variable.
- For other instructions, we merge the computational units of the instructions on which they are data and control dependent and assign the resulting merged computational unit to the instruction.

Figure 5 shows an example for the computational units detection. The presented method contains two possible control flow paths. Our first path skips the control flow branch of the if-statement. As a result, we identify CU_1 and CU_3. Since we demand a full branch coverage of the control flow, our second path follows the control flow branch and identifies CU_2 and CU_3. Thus, we have identified all three possible computational units of the method.

4.5 Identifying Correlated Variables

For identifying correlations between variables we perform an analysis on the given computational units of the whole program. Our approach is based on the concepts of H^{Corr} [9] and MUVI [8]. According to H^{Corr}, variables are correlated if they are accessed within the same computational unit. This implies a strong relationship between data/control dependencies and variable correlations. However, this criterion seems to be too weak for successfully detecting correlated variables. Variables that may be initialized in the same computational unit but bear no further connection during the rest of the program can hardly be called correlated. We expect correlated variables to be in relation to each other during most of the program's execution. The approach MUVI uses is based upon this assumption. Here it is assumed that variables which are accessed relatively often near to each other are likely to be correlated. However, it does not additionally regard data and control dependencies for its analysis.

We introduce a hybrid approach for identifying correlated variables which combines the ideas of H^{Corr} and MUVI. As a result, we consider variables whose accesses appear relatively often in the same computational units to be correlated. The more frequently variables are accessed in the same computational units, the higher the probability that these variables are actually correlated. We call this probability the correlation probability. In order to compute the correlation probability for a variable pair, we sum up the total number of accesses to these variables inside the program. Then, the correlation probability is the percentage of accesses that appear inside a computational unit accessing both variables of the pair.

We only identify correlations between shared variables. Trivially, a correlation consisting only of local variables cannot be involved in an atomicity violation; there is just one thread accessing the participating variables.

Figure 6 gives an example for the correlation detection approach, in which we consider only two methods accessing shared variables. In this situation the variables $SizeFt$ and $SizeCm$ are correlated. During the method $Initialize$ the data dependencies between $Count$ and $SizeFt$ is just arbitrary since they have the same initial values. The computational unit obtained from $SetSize$ gives a better representation of the semantic relationships between the variables. When we apply our correlation detection algorithm on the two methods, we identify the correlated variables $SizeFt$ and $SizeCm$ with a correlation probability of 100%. The correlation probabilities involving the uncorrelated variable $Count$ are significantly lower.

```
Function Initialize()
    $SizeFt = 0$;
    $SizeCm = SizeFt$;
    $Count = SizeFt$;
end
Function SetSize(newSize)
    $SizeFt = newSize$;
    $SizeCm = SizeFt * 30,48$;
    $Count = Count + 1$;
end
```

$CU(Initialize) = \{SizeFt, SizeCm, Count\}$;
$CU_1(SetSize) = \{SizeFt, SizeCm\}$;
$CU_2(SetSize) = \{Count\}$;

$\#Accesses(SizeFt) = 5$;
$\#Accesses(SizeCM) = 2$;
$\#Accesses(Count) = 3$;

$CorrelationProb(SizeFt, SizeCm) = 7/7 = 1$;
$CorrelationProb(SizeFt, Count) = 4/8 = 0.5$;
$CorrelationProb(SizeCm, Count) = 2/5 = 0.4$;

Fig. 6. Correlation probabilities for the variable pairs accessed in two methods

4.6 Endangered Atomicity

After we have identified computational units and correlated variables, we determine endangered high-level atomic regions. Therefore, we consider the synchronisation instructions of the method pair. Any kind of synchronization instruction inside the mp-shared region of a computational unit suggests a possible atomicity violation. For this reason, we regard the atomicity of that computational unit as being endangered. We determine endangered correlated variables in a similar manner. A synchronization instruction which separates two accesses to correlated variables hints at a high-level atomicity violation.

If we do not detect any synchronization instructions, we can assume that the accesses in consideration are either fully and continuously synchronized or not synchronized at all. Of course, the latter case may lead to low-level race conditions and, if applicable, a violation of a high-level atomic region. Despite that, we do not consider this case an endangerment of high-level atomicity, since totally unsynchronized accesses naturally come with low-level race conditions. Therefore, the flaws can be found by low-level race detectors. Thus, a generated unit test for such a method pair should be classified as low-level and analysed by a low-level race detector. Figure 7 illustrates examples for high-level atomic regions we consider endangered or safe.

```
$temp = a$;
$a = temp + 1$;
```

(a) Safe atomic region

```
$temp = a$;
Acquire Lock
    $a =$
    $temp + 1$;
end
```

(b) Endangered atomic region

```
Acquire Lock
    $x = 2 * x$;
    $y = 2 * y$;
end
```

(c) Safe correlation accesses

```
Acquire Lock
    $x = 2 * x$;
end
Acquire Lock
    $y = 2 * y$;
end
```

(d) Endangered correlation accesses

Fig. 7. Examples of safe and endangered high-level atomic regions. Note: All variables are shared and the variables x and y are correlated.

4.7 Unit Test Classification

After we have determined the endangered high-level atomic regions we classify the method pairs accordingly:

Low-Level: The method pair should not contain any high-level race conditions. Therefore, it does not include computational units or correlated variables, which are endangered.

Atomic High-Level: The method pair contains at least one endangered computational unit. However, there are no endangered accesses to correlated variables.

Correlated High-Level: The method pair contains at least one endangered variable correlation.

In this kind of classification we willingly allow the possibility of low-level race conditions to be present inside high-level method pairs. This is because high-level race detectors are generally able to also identify low-level race conditions. On the other hand, low-level race detectors are unable to identify high-level race conditions. Therefore, we assign method pairs which may contain low-level and high-level race conditions to a high-level class.

5 Implementation

We implemented AutoRT+ in C# which runs within the .NET runtime. 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 parallelism analysis protocols 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 our approach we need to identify data and control dependencies. CCI already provides simple data and control flow analysis data structures. However, control flow branch analysis, which is required for the control dependencies, 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 atomic regions 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, as our analysis does not distinguish between the types of synchronization, it is therefore able to identify synchronization instructions in general.

6 Evaluation

We use sample programs as well as real-world applications for our evaluation purposes. Table 1 lists the programs and provides an overview of their most important characteristics and evaluation results.

We used the programs *Bank Account*, *BoundedQueue* and *Dekker* from CHESS, which provides small programs containing high-level data races. We chose an order-system from MSDN Code Gallery [15]. We implemented an alternative version containing various correlated variables. Furthermore, we evaluated the following open source programs:

Table 1. Summary of the evaluation programs

Program	LOC	Meth.	Thrds	Par. Meth.	Par. Meth. Pairs	Corr. Vars	Unit Tests	Gen. Time (ms)
Bank Acc.	25	4	2	4	4	0	2	1030
Queue	31	6	3	6	14	2	10	451
Dekker	15	3	3	3	5	2	3	238
Order Sys.	360	7	5	5	15	13	15	1820
Corr Sys.	480	18	5	10	27	18	14	1923
Petri Dish	1070	35	7	35	230	12	24	23050
Kee Pass	1240	58	16	58	478	18	18	49300
STP	1120	46	12	37	315	25	53	29400
.Net Zip	14k	2366	19	63	1343	35	87	93900
Cosmos	78k	12k	19	269	5660	35	87	224600

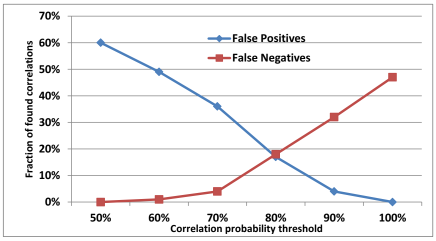
- PetriDish [16], a simulation of three categories of organisms, all growing, mating, and eating each other.
- KeePass [17], a password manager.
- SmartThreadPool (STP) [18], a thread pool library.
- DotNetZip [19], a toolkit for manipulating zip files.
- Cosmos [20], an operating system toolkit.

6.1 Correlation Detection Efficiency

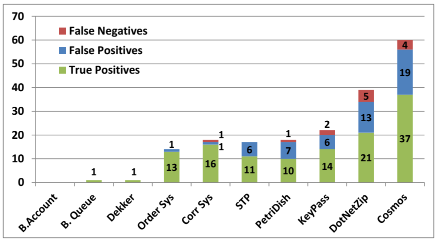
We compare the efficiency of different thresholds for the correlation probability in order to obtain the most suitable threshold. For this reason, we consider the number of variables falsely identified as being correlated, the false positives, and the number of missed correlated variables, the false negatives. In our evaluation we tested thresholds for 50% to 100% correlation probability.

In our 10 evaluation programs we detected 134 correlations in total. The efficiency of our correlation detection analysis depends highly on the chosen threshold for the correlation probability. Generally, we expected and observed that a low threshold leads to many false positives and fewer false negatives. A high threshold, on the other hand, prevents false positives but also drastically increases the number of false negatives.

Figure 8a shows the overall distribution of false positives and false negatives and relation to the correlation probability threshold. The break-even point between false positives and false negatives are approximately 80%. At this point,



(a) Percentage of false negatives and false positives in relation to the correlation probability threshold



(b) Distribution of false negatives and false positives for a correlation probability threshold of 70%

Fig. 8. Efficiency of the correlation detection

we observed 18% false positives and false negatives. However, we rate false negatives more critically than false positives. As a result, we regard a threshold of 70% to be ideal according to our observations. At this threshold the percentage of false negatives is less than 5%. But as a major drawback we have to deal with an average of 35% false positives.

Figure 8b shows the distribution of false positives and false negatives in regard to the single evaluation programs with a 70% correlation probability threshold. In the smaller test programs we observe far fewer false negatives and false positives, each less than 5%. Some programs do not contain many correlations and due to the small program size the contained correlations are rather obvious. The open source programs prove to be more representative: Only with KeePass was there a relatively low amount of false positives at 22%. The other programs are close to the average of 35%. Considering false negatives, only DotNetZip stands out, having 12% false negatives.

6.2 Classification Precision

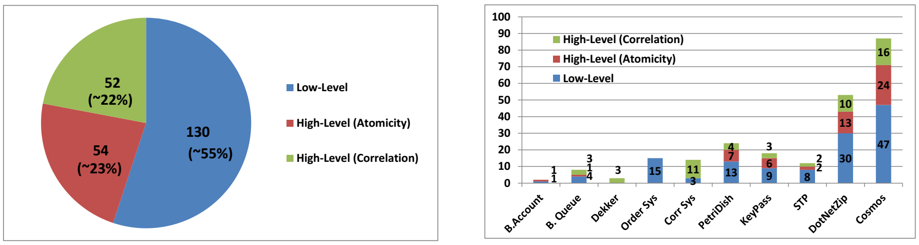
We identify falsely classified unit tests based on the contained atomic regions and the detected race conditions inside tests.

1. Low-level unit tests should not contain endangered high-level atomic regions or high-level race conditions.
2. High-level unit tests in general should not only contain low-level race conditions. Either they contain no race conditions or an arbitrary amount of race conditions from which at least one is a high-level race condition.
3. High-level atomic unit tests should not contain endangered variable correlations or race conditions on correlated variables.
4. High-level correlation unit tests should either contain no race conditions or an arbitrary amount of race conditions from which one is at least a race condition on correlated variables.

We consider unit tests which differ from the specification above as falsely classified. In order to identify race conditions we analysed the generated unit tests with four different race detectors: CHESS, ITC, H^{Corr} and NDR (see section 3).

CHESS and ITC are unable to detect high-level race conditions and were used to determine strictly low-level race conditions. H^{Corr} and NDR on the other hand are both able to detect high-level atomic and correlation race conditions.

AutoRT+ generated 236 parallel unit tests in total. Our approach was able to categorize these tests as shown in figure 9a. According to the observed distribution, the majority, roughly 55% of all unit tests, were categorized as low-level. Furthermore, our approach classified 25% of all generated unit tests as high-level atomic and 25% as high-level correlated.

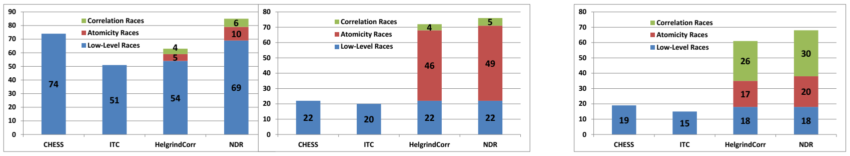


(a) Proportions of generated unit tests in total

(b) Proportions of generated unit tests for each evaluation program

Fig. 9. Distribution of generated parallel unit tests

Figure 9b shows the distribution of unit tests with regard to the evaluation programs. Again, we can observe a significant difference from the average distribution in the smaller test programs. The test program 'order system' was designed not to contain any high-level race conditions which naturally resulted in generating exclusively low-level unit tests. On the other hand, 'corr system' mainly consists of accesses to correlated variables. A higher amount of high-level unit tests was therefore expected. The unit tests for the open source programs follow the average distribution.



(a) Results for low-level tests

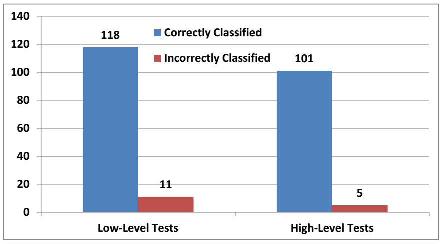
(b) Results for high-level atomicity tests

(c) Results for high-level correlation tests

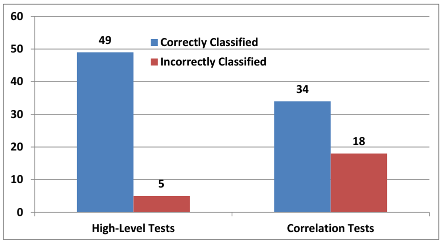
Fig. 10. Results of the race detectors applied to the classified unit tests

Figure 10 shows the findings of the race detectors which we applied on the classified unit tests. In the low-level tests we observed about 18% high-level findings. Furthermore, of the reported findings of the high-level atomicity test, 29% were low-level and 7% were correlated high-level findings. Finally, the high-level correlation tests included 28% low-level and 29% uncorrelated high-level findings.

The validity of the classification between low-level and high-level unit tests is illustrated by figure 11a. We observed that 11 of the unit tests (8%) which were classified as low-level are actually high-level unit tests. In these cases, the region hypothesis has failed to identify correct high-level atomic regions resulting in computational units that are too small. In this way, our approach was unable to identify the related atomic regions as endangered. Finally, 5 of the unit tests (3%) classified as high-level are actually low-level unit tests. In this case the employed region hypothesis lead to the estimation of atomic regions (computational units) that were too large.



(a) Differentiation between low-level and high-level unit tests



(b) Differentiation between high-level and correlated unit tests

Fig. 11. Precision of the categorization approach

The distinction between regular high-level unit tests and unit tests containing endangered correlations turned out to be far more imprecise, as figure 11b illustrates. 5 unit tests (9%) contained undetected endangered variable correlations. A major influencing factor is the false negatives of the correlation detection. Additionally, we have a high amount of unit tests falsely categorised as unit tests for correlated variables. In total, 18 unit tests (35%) were falsely categorized this way. Here, the high amount of false positives in the correlation detection heavily influences the outcome.

6.3 Performance

Figure 12 shows the time our analysis takes for classifying and ultimately generating the unit test in relation to the execution time. Since our approach analyses method pairs, the time of classification is heavily dependent on the number of parallel method pairs inside the program. The time for unit test generation is a sum of different partial times including the static parallel dependency analysis, the correlation analysis, the dynamic parallelism analysis and the capture-and-replay technique. Our experience is 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 by a factor between 16 and 266. Large programs with many objects and many parallel methods like Cosmos cause a high state recording time. The static correlation analysis only takes a small part of the overall generation time. Therefore, the categorization time only takes a small part of the total unit test generation time. In average about 5% of the total time goes into our additional analysis. In the smaller test programs we can even report a rate under 1%.

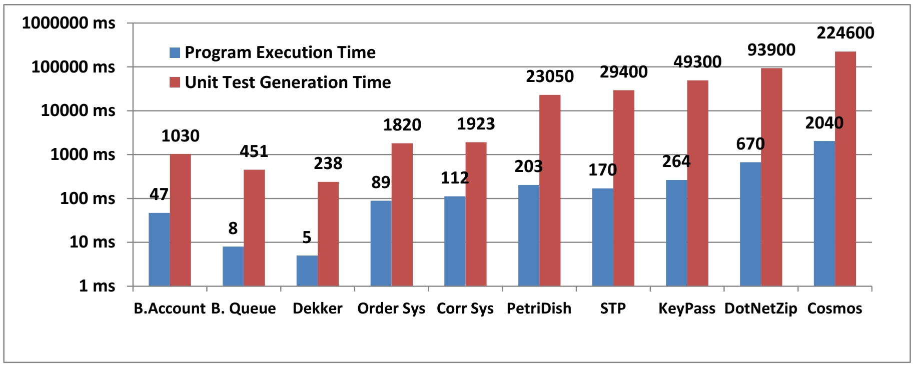


Fig. 12. Comparison between the unit test generation times of AutoRT+ and the execution times of the evaluation programs

7 Conclusion

In this paper we introduced an approach which enhances automatic parallel unit test generation and execution with a totally new dimension: classified unit tests. Our analysis is able to distinguish between unit tests that should be analysed by low-level race detectors, detectors for correlated variables or high-level race detectors in general. This supports testing of parallel software by reducing the number of unnecessary unit tests or unsuitable employed race detectors. Overall for ten different applications, we were able to classify 83% of the generated unit tests correctly.

In the future, we want to introduce new classes to AutoRT+. Generally, different race detectors vary in their effectiveness to detect 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 additional classification analysis and clustering metrics which state how method pairs are suited for specific race detectors. Furthermore, we can extend our heuristics to other concurrency bugs like deadlocks and order violations.

Another direction for our future work would be 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 annotation for correlation specifications e.g. CHESS [10] or [21]. However, race detectors with automatic correlation detection may profit from a reduced performance overhead and increased precision by our preceding correlation analysis.

References

1. Hamill, P.: Unit Test Frameworks: Tools for High-Quality Software Development. O'Reilly Series. O'Reilly Media (2008)
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. Xu, M., Bodík, R., Hill, M.D.: A serializability violation detector for shared-memory server programs. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 1–14. ACM, New York (2005)
4. Luo, Q., Zhang, S., Zhao, J., Hu, M.: A lightweight and portable approach to making concurrent failures reproducible. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 323–337. Springer, Heidelberg (2010)
5. 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 1999. IEEE Computer Society, Washington, DC (1999)
6. Wong, W.E.: 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, pp. 539–548. IEEE Computer Society, Richardson (2005)
7. 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, pp. 727–737. IEEE Press, Piscataway (2012)
8. Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R.A., Zhou, Y.: Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: SOSP 2007: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, pp. 103–116. ACM, New York (2007)
9. Jannesari, A., Westphal-Furuya, M., Tichy, W.F.: Dynamic data race detection for correlated variables. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) ICA3PP 2011, Part I. LNCS, vol. 7016, pp. 14–26. Springer, Heidelberg (2011)
10. Musuvathi, M., Qadeer, S.: Chess: Systematic stress testing of concurrent software. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 15–16. Springer, Heidelberg (2007)
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 2008, pp. 267–280. USENIX Association, Berkeley (2008)

12. Intel (Intel Inspector XE) (2013), `http://software.intel.com/en-us/intel-inspector-xe`
13. Jannesari, A., Koprowski, N., Schimmel, J., Wolf, F., Tichy, W.F.: Detecting correlation violations and data races by inferring non-deterministic reads. In: Proc. of the 19th IEEE International Conference on Parallel and Distributed Systems (ICPADS). IEEE Computer Society, Seoul (2013)
14. Cooper, K.D., Harvey, T.J., Kennedy, K.: (A simple, fast dominance algorithm)
15. Microsoft: Code gallery for parallel programs, `http://code.msdn.microsoft.com/Samples-for-Parallel-b4b76364`
16. Butler, N.: Petridish: Multi-threading for performance in c#, `http://www.codeproject.com/Articles/26453/PetriDish-Multi-threading-for-performance-in-C`
17. Reichl, D.: Keepass password safe, `http://keepass.info/`
18. Smart thread pool, `http://smartthreadpool.codeplex.com/`
19. Dotnetzip, `http://dotnetzip.codeplex.com/`
20. C# open source managed operating system, `https://cosmos.codeplex.com/`
21. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL 2006: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 334–345. ACM, New York (2006)