Automatic Generation of Parallel Unit Tests

Jochen Schimmel*, Korbinian Molitorisz*, Ali Jannesari†‡, Walter F. Tichy*
*Karlsruhe Institute of Technology (KIT), Germany
{schimmel, molitorisz, tichy}@kit.edu
†German Research School for Simulation Sciences, Aachen, Germany
‡RWTH Aachen University, Aachen, Germany
{a.jannesari}@grs-sim.de

***Abstract*—Multithreaded software is subject to data races. Currently available data race detectors report such errors to the developer, but consume large amounts of time and memory; many approaches are not applicable for large software projects. Unit tests containing fractions of the program lead to better results. We propose AutoRT, an approach to automatically generate parallel unit tests as target for data race detectors from existing programs. AutoRT uses the Single Static Multiple Dynamic (SSMD) analysis pattern to reduce complexity and can therefore be used efficiently even in large software projects. We evaluate AutoRT using Microsoft CHESS and show that with SSMD all 110 data races contained in our sample programs can be located.**

***Index Terms*—Data Races, Unit Testing, Multicore Software Engineering**

I. Introduction

Writing parallel programs is hard. Data races and deadlocks are error types specific for multicore software development. Finding such errors is a tedious and non trivial task: Due to alternating schedules, data races may only show up on rare occasions, rendering conventionell debugging and reproduction techniques effectless. To ease this situation, data race detection strategies are a prospering scientific field for several years. The data race detectors available today may coarsly be grouped into three categories: (1) static, (2) dynamic, and (3) model-driven. All categories have their own strenghts and weaknesses. However, all of them share the fact that they have not yet achieved broad acceptance in the professional developer community. The common main problem with data race detectors is that they either (1) consume too much memory and computation time or (2) report too many false positives. A fact that we see accross many race detectors is that the results are impressive when applied to small applications, but with increasing sizes of real world applications, race detection approaches become increasingly impractical. In general parallel unit tests would be well-suited to overcome the well-known issues with current data race detectors mentioned above. However, it is difficult to write and design a profound set of parallel unit tests: What method pairs need to be tested with what input data?

In this paper we present AutoRT, an extension to regular parallel unit tests: AutoRT automatically generates parallel unit tests from whole programs, produces a test context and checks for concurrency bugs using a dynamic race detector. Regular parallel unit tests run two or more methods in parallel on given input data and program state. With AutoRT we apply a data race detector on parallel unit tests instead of a whole program and therefore reduce complexity to a set of small fractions instead of a single large one. We use Microsoft CHESS [1], a race detector that is speficially designed to be applied to unit tests. Thus AutoRT overcomes scalability limitations. AutoRT uses a combined static and dynamic analysis pattern we call Single Static Multiple Dynamic (SSMD, chapter IV). In a first step, SSMD identifies method pairs containing accesses to common data through the notion of strong parallel dependency (chapter IV-A). These method pairs are subject to multiple dynamic analyses. SSMD identifies methods that effectively run in parallel, gathers corresponding test contexts and removes equivalent method pairs. Finally, parallel unit tests are created from method pairs and test contexts and passed on to Microsoft CHESS. We evaluated our approach using several academic sample programs and the Microsoft Parallel Sample Library as benchmark and show that AutoRT finds all of the 110 known data races.

II. Parallel Unit Test Basics

AutoRT extends regular parallel unit tests (parUT) and data race detectors. In this chapter we present basics to better understand the concept presented in chapter III. A parallel unit test is a dedicated test method in a separate class, which executes two or more methods in parallel. Each of these methods under test is executed within its own thread. The test class also initializes required parameters and global data. It returns as soon as all parallel methods under test finish their execution.

A. Parallel versus Conventional Unit Tests

A parallel unit test differs from a conventional unit test in several ways. A conventional unit test is self-verifying: After the execution of a test method, assert statements check if the test method behaved as expected. If not, an exception is thrown. One problem when dealing with parallelization errors is the lack of a mechanism to assert correctness. A parallel unit test is not self-verfiying: There is no assertion-like mechanism to verify that the test is successful or not. Even if a parallel unit test contained a regular assertion to check for the correct result, the data race would still not be found, because parallelization errors inherently occur extremely rare. An assertion can only report a deviation to the expected result, if it ocurrs, so a regular parallel unit test will very likely execute and terminate successfully, although it contains a data race. A regular parallel

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

unit test cannot decide whether the methods under test behave as inteded, but a combination with a data race detector can, as Figure 1 illustrates. `Increment()` and `Decrement()` that are show in Figure 4 both access `x`, so there obviously is a data race on variable `x`. AutoRT automatically generates parallel unit tests like in Figure 1 using SSMD and passes them on to a race detector. Related works such as ConCrash [2] generate unit tests as soon as a data race detector reports a data race to assist in error reproduction. In contrast to AutoRT this approach still has the scalability disadvantage: Applying race detection on a whole software system takes a very long time. A parallel unit test in AutoRT clearly focuses on the relevant portions of a software system that are worth the effort of running data race detection.

B. Extending Data Race Detectors

As explained, a parallel unit test is deliberately not self-verfiying and leaves the race detection to other tools. We decided that our unit tests do not anticipate a certain mechanism to detect races. Following the principle of separation of concerns we are able to use a variety of data race detectors on each parallel unit test. We also enable productive use of detectors that are too slow and memory intensive: Especially model based detectors may benefit from this design decision, as they evaluate a unit test as a single small instance instead of a whole program. In our implementation depicted in chapter V, we use Microsoft CHESS as a race detector. CHESS is designed to find races in small tests, executing only small fractions of large programs [1]. In extension to this, we are currently working on a .NET-implementation of Helgrind+ [3], a novel data race detector to find correlated races which can so far not be found by CHESS. The aspect of scalability accounts for a combination of different data race detectors even more: Applying any of these detectors consecutively on a complete program would consume even more time, but for an automatically generated parallel unit test the combination executes with acceptable speed.

III. Requirements of a Parallel Unit Test Generator

When parallel unit tests are created manually, an engineer has to identify the methods that could potentially be executed in parallel, locate relevant input parameters and assign object states in order to evoke the desired control flow. Even for experienced software testers, this task is far from trivial because of the following problem areas:

- A parallel execution might be overlooked resulting in incomplete test sets.
- Testins regions that are not executed in parallel unnecessarilly extend test execution duration.
- Methods containing data races but that will never occur due to control flow dependecies don't have to be tested in order to minimize false positives.

The automatic generation of unit tests face the same challenges as the manual generation, so we define the following goals for AutoRT: (1) Reduction of test cases to relevant code areas, (2) completeness of the set of test cases and (3) correctness of the executed tests. We will evaluate these goals in section VI. In the remainder of this section, we will explain the requirements to fulfill these threee goals. Section V presents the implementation of AutoRT.

```
// x defined in program assembly
// public int x;

[TestMethod]
void TestIncrementDecrement()
{
    x = 350;
    amount_1 = 500;
    amount_2 = 200;

    Thread t1 = new Thread( delegate {
        Increment(amount_1); }).Start();

    Thread t2 = new Thread( delegate {
        Decrement(amount_2); }).Start();

    t1.Join(); t2.Join();
}
```

Fig. 1. Test case generated by AutoRT for the methods shown in Figure 4.

1) Test Case Reduction: It is commonly known that exhaustive testing of parallel code leads to a tremendous number of states due to the large number of possible thread interleavings. Parallelization errors however can only occur in the relatively few areas that are effectively executed in parallel. This leads to a large search space although the problematic code regions are relatively small. At the same time we want to be able to filter out redundant test cases. We define the following requirements in order to reduce the number of test cases.

Red_1: Method pairs that never access common memory areas are excluded from the test set, as they cannot lead to data races. We therefore conduct a static pre-analysis to filter out those methods that don't access common data. Because of the potentially high number of common states AutoRT employs a lightweight analysis.

Red_2: Variables might influence the intra-procedural control flow. Variables that cannot lead to data races are excluded from the test set. We therefore use a dynamic analysis to check for reference identity at commonly used object instances. This is motivated by the fact that two parallel accesses to a common data type only result in a data race, if the same object instance is used. As [4] shows, this problem is statically undecidable, so we employ a dynamic analysis.

Red_3: Methods invoked with different parameter values that execute the same control flow path either contain a data race or not. If they do, it's the same data race and the methods only have to be checked with one parameter value. We therefore identify equivalence classes for variable allocations that lead to the same control flow path and might therefore lead to the same data race. A different strategy to retrieve equivalent classes of program states and variable assignments that evoke the same data races would be a symbolic execution. This procedure could enrich our dynamic analysis efficiently:

Capturing the execution paths at runtime would cause minor overhead. A symbolic execution would be able to detect equivalent classes but at the same time put the correctness at risk as shown in $Corr_2$. The correctness clearly is more crucial than having less redundancy in the test set, so we relinquish this mechanism.

Red_4: Method pairs that are never executed in parallel need not be tested. This requirement is also important for $Corr_1$.

2) Correctness: $Corr_1$: Method pairs that do not run in parallel in any thread interleaving can safely be ignored. We use a dynamic analysis to retrieve parallel methods and let the user remove incorrect test cases. The strengths of a dynamic analysis are the absence of false positives for parallel method pairs and scalability of the dynamic recording. The goal to reduce user interaction wherever possible still counts, but in this case user interaction might be necessary. An alternative is the may-happen-in-parallel analysis ([5]). It can successfully be used to retrieve certain parallel methods, but at the price of a worst-case-complexity of O(N^5), which is not suitable for complex applications. Although may-happen-in-parallel produces few false positives, it cannot reliably identify all parallel code areas.

$Corr_2$: Variable assignments for method pairs that evoke races when executed in parallel, but cannot occur at runtime are excluded from the test set. We employ the same dynamic analysis for the retrieval of variable assignments. The requirement for correctness demands the absence of false assumptions about race conditions. $Corr_2$ assures that only those variable assignments are taken into the test set, that effectively occur at runtime.

3) Completeness: Completeness of the generated test set is crucial, as unidentified parallel code blocks might contain data races. As the two previous requirements test case reduction and correctness might interfere with the completeness, we define the following requirements:

$Comp_1$ All method pairs that execute in parallel in any thread interleaving and stand in a strong parallel dependency relationship must be included in the test set (see chapter IV-A). This demands that no parallel method pairs are ignored. In a dynamic analysis, this can indeed happen due to unfavorable thread interleavings. As discussed, we do not use a may-happen-in-parallel analysis. Instead, we employ heuristics to raise the probability for a parallel interleaving. It adds a time supplement to methods, that executed very close to each other assuming that with a slightly different scheduling, they might have interleaved.

$Comp_2$ For all relevant method pairs all instruction interleavings must be covered without tampering with $Corr_2$. To achieve this, we evaluated symbolic execution, may-happen-in-parallel analysis and user unteraction. The two latter solutions can result in a complete test set but with the serious disadvantage that the two other goals correctness and test case reduction are violated. At the time of writing, the test user has to define a set of execution parameters that covers all relevant execution paths.

IV. Automatic Generation of Parallel Unit Tests using SSMD

Our approach to automatically generate parallel unit tests follows a pattern we call Single Static Multiple Dynamic (SSMD), as depicted in Figure 2. In SSMD, a single static analysis retrieves data required by multiple dynamic analysis runs. Our approach contains two different dynamic steps, which are repeated several times. The first dynamic run determins what methods in the program run in parallel and the second records object state information about them. To explore parallel regions which cannot be reached within a single program execution, these two steps have to be iterated. After data collection, AutoRT emits the test methods. AutoRT consists of four steps: (1) static parallel dependency analysis, (2) dynamic method analysis, (3) dynamic object state recording, and (4) test case generation. We present each step in this section.

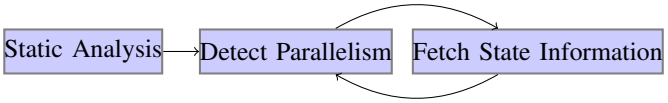

Fig. 2. The Single Static Multiple Dynamic Pattern (SSMD).

A. Parallel Dependency

We define the term parallel dependency as a foundation for our static analysis step. It is used to identify methods worth examining for unit test generation.

For all methods M in a program, S is the set of shared variables across multiple threads. The write access relation $W \subseteq M \times S$ and read access relation $R \subseteq M \times S$ are defined as follows:

$$W := \{(m,s) | m \in M \wedge s \in S \wedge \exists i \in M : \text{Value of s is written in instruction i}\} \tag{1}$$

$$R := \{(m,s) | m \in M \wedge s \in S \wedge \exists i \in M : \text{Value of s is read in instruction i}\} \tag{2}$$

The parallel dependency PD is a binary relation $PD \subseteq M \times M$ that puts two methods m_1 and m_2 in a relation, when they contain at least one instuction reading or writing a common variable. Formally, PD is defined as follows:

$$\begin{aligned} PD := \{(m_1,m_2) | m_1, m_2 \in M \wedge \exists s \in S : \\ (W(m_1,s) \wedge R(m_2,s)) \vee \\ (R(m_1,s) \wedge R(m_2,s)) \vee \\ (R(m_1,s) \wedge W(m_2,s)) \vee \\ (W(m_1,s) \wedge W(m_2,s))\} \end{aligned} \tag{3}$$

Definition 1: Two methods which both access a common variable are called parallel dependent. If at least one of the methods performs a write access, they are called strongly parallel dependent. If none of the accesses is a write access, they are called weakly parallel dependent. Any method which

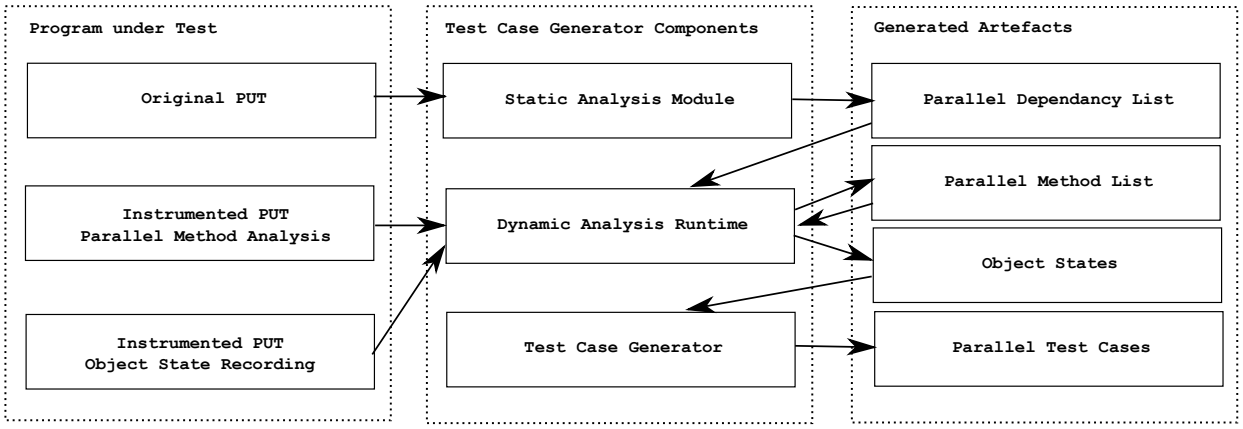

Fig. 3. The Architecture of AutoRT

is part of a [strongly—weakly] parallel dependent pair is also called [strongly—weakly] parallel dependent.

According to this definition we can further refine the relation PD as follows:

$$PD = PD_s \cup PD_w \tag{4}$$

where PD_s contains strongly parallel dependent methods and PD_w contains weakly parallel dependent methods, that are excluded from the dynamic analyses.

B. Static Parallel Dependency Analysis

We do not want to generate parallel unit tests for each method pair in a program. In order to determine which method pairs are relevant for data race detection, we perform a static pre-analysis. We use the notion of strong parallel dependency in order to identify relevant method pairs. For test generation we only consider strongly parallel dependent method pairs at the moment. These method pairs are candidates for test case generation, as they contain at least one write access to a potentially shared variable, which is a precondition for a data race. During dynamic analysis it is determined whether the variable is actually shared.

Although AutoRT currently only uses the strong parallel dependency analysis, we present a scenario that would profit from weak parallel dependency: Using strongly dependent method pairs is a sound approach to test programs that currently contain data races. However, during early stages of software development, a data race might only be included in later revisions. For this, regression tests are an interesting option. While the software is still subject to development changes, weakly dependent method pairs might additionally be used for test case generation. Weakly dependent method pairs contain harmless read accesses that might be updated to contain write accesses in future software revisions. Such test cases would currently not report any races, but would do so as soon as races sneak into the program. This shows a major difference between parallel test case generation and data race detection: While a data race detector has to be as precise as possible and should not report any false positives at all, test case generation embraces code, which does not contain a data race right now, but might easily do so in the future.

C. Dynamic Method Analysis

In this step, the program under test is instrumented to record entry and exit time stamps for the set of methods reported by the static analysis. After that, the program is run. Using the information recorded, it can be seen which methods ran simultaneously. These are the method pairs for which AutoRT has to generate parallel test cases. The methods of these pairs are called test case candidates.

Obviously, test cases can only be generated for program parts that have been executed during dynamic analysis. It is the software tester's responsibility to rerun the application with input data that cover all parallel program parts. This input data is already available in most software projects to test the correct program behaviour.

D. Dynamic Object State Recording

The program under test is instrumented again: All test case relevant methods are instrumented to write down the values of all parameters and objects they access as soon as the method is entered - we get a complete object state dump. Our runtime detects when the same instance of an object is used several times and stores this information - even accross thread boundaries. So methods running in parallel and accessing the same object instance will access a single object instance in the generated test case, too. If a method calls a sub-routine, this method will also dump its object state information. In the current version, each call to an instrumented method will produce a full state dump of the accessed objects, so obviously useless data is recorded; for example, frequently called getter-methods record the same values many times. We are currently working on improvements to dynamically decide if further dumps are required to generate the full set of parallel test cases to improve memory consumtion and run time.

E. Test Case Generation

For each method pair identified, at least one parallel test case has to be generated. In order to run both methods within

such a test case, all objects accessed by both methods need to be initialized to a valid state before the method calls. For each relevant method, one or more object state dumps are available - depending on how often the method has been executed. The test case generator will emit code to restore the object states for each tested method within a parallel unit test. We call the set of object states accessed by two (ore more) methods the test context. If both of these methods ran parallel during object state recording, we get such a test context directly. But it is also possible to construct a test context for two methods if both were executed sequentially. How to combine these states is not obvious, as both methods may have accessed a common global object with different state. If it is possible to combine two object states, we call them compatible. Object states are always compatible if they do not share any objects. They are also compatible if all fields of each common object contain the same value. At the moment, this is the only situation in which we apply test context construction. For future versions, we will explore how to merge object states with different values.

V. Implementation

We implemented our test case generator in C#. For instrumentation, we used the Phoenix compiler framework [6] in a first version and updated it to use Microsoft CCI [7]. We instrument .NET-Assemblies and generate test cases in C#. The C# file generated compiles into a .NET-Assembly which contains a class with test methods. Microsoft CHESS and other data race detectors can then execute the test methods.

Test cases generated with our sample implementation reconstruct values of global variables including data structures of the .NET Framework such as lists or dictionaries. Generic data types are included in this process, too. A generated sample test case is shown in Figure 1. A test case cannot set protected and private fields by assignment. For such elements, we emit .NET reflection API code. Using this technique, we avoid calling complex object initializers and constructurs, but may set only those parts of an object's state that can be seen be the methods under test.

The architecture of our implementation is depicted in Figure 3. First, our static analysis module will examine all CIL-methods of the program under test for parallel dependencies. Using the dependencies found, an instrumented version of the program under test is generated which links to our dynamic analysis runtime. During execution of the instrumented program under test, our runtime records which methods actually run in parallel. Multiple executions with different input may be necessary to examine all program parts. After that, the generated parallel method list is used to instrument parallel methods for object state recording. The program is executed again. Finally, our test case generator uses the program states.

A sample program under test with the according instrumented versions is shown in Figure 4. In (a), we can see the original version of two parallel dependent methods. They are strongly parallel dependent, as they both contain a write access to the global variable x and run in parallel. (b) shows the instrumented version for parallel method analysis: calls to our dynamic analysis runtime are added at the beginning and the end of the methods. The runtime logs timestamps for method entry and exit. After execution, the runtime decides which methods ran in parallel. For those methods, a second instrumentation of the program is performed (c): In this version, runtime calls are added for each global variable as well as any method parameter at the beginning of the method. With the resulting information, a test case can be generated which initializes the program state with the same values as during real program execution. Finally, Figure 1 shows the resulting parallel test case: Both methods are called within a thread for each method. Before starting these threads, parameter values as well as global variables accessed by these methods are initialized with the values recorded. A barrier will block the main thread of the test to wait for both methods to end.

VI. Evaluation

This section presents our sample applications and the results of our experiments.

A. Sample Applications

The samples chosen for evalution are composed from parallel sample applications found in the MSDN Code Gallery, samples from the CHESS data race detector and programs written on our own, such as a quad tree simulation. The applications from the CHESS package already contain data races as found in real-world applications; we chose them to show that our test case generator may bring up these kinds of errors. In the programs from the MSDN Code Gallery and in our own applications, we introduced data races manually by removing synchronisation instructions. After generating parallel unit tests, we used CHESS to find the data races. We then fixed those errors and applied CHESS again to the parallel unit tests, to show that the tests then execute without reporting any errors. The results of our evaluation are shown in table I.

B. Race Detection Efficiency

Our data race detector is not designed to detect data races itself. Instead, it generates test cases which enable data race detectors such as CHESS to detect data races. As the overall goal is to detect races, the number of detected races is the most important metric. Our data race detector generated 50 parallel unit tests for the 8 sample programs. The sample programs contained 110 data races, of which all have been found by CHESS, when applied to the parallel unit tests. Using the tests, we also found a data race in our quad tree application which we did not introduce deliberately. We were not able to find this bug without our generated test cases, as CHESS was not able to complete the application when running it as a whole program.

C. Performance

In order to generate parallel unit tests, three performance critical steps have to be performed: (1) static analysis, (2) dynamic method analysis and (3) dynamic object state recording.

```
int x = 0;

void Increment(int amount)
{
    x = x + amount;
}

void Decrement(int amount)
{
    x = x - amount;
}
```

(a) Two methods of the Bank Account sample.

```
int x = 0;

void Increment(int amount)
{
    InstrLib.Log(Entry, Increment);
    x = x + amount;
    InstrLib.Log(Exit, Increment);
}

void Decrement(int amount)
{
    InstrLib.Log(Entry, Decrement);
    x = x - amount;
    InstrLib.Log(Exit, Decrement);
}
```

(b) Instrumentation for Parallel Method Detection.

```
int x = 0;

void Increment(int amount)
{
    InstrLib.Log<int>(Increment, x);
    InstrLib.Log<int>(Increment, amount);
    x = x + amount;
}

void Decrement(int amount)
{
    InstrLib.Log<int>(Decrement, x);
    InstrLib.Log<int>(Decrement, amount);
    x = x - amount;
}
```

(c) Instrumentation for Object State Recording.

Fig. 4. Code excerpt of the Bank Account Sample with its instrumented versions.

We experience the most sever performance impact in steps (2) and (3). The slowdowns vary from good (factor 1.2) to dramatic (factor 190). The main reason for this is that we record methods blindly in our current implementation: If we instrument a method `foo` to record its object states during runtime, it will do so in every single method call. It will do so even if all required data from `foo` has already been recorded. If such a method uses many objects or is called within a loop, performance is doomed. Therefore, we will include a dynamic instrumentation in our next version, which will stop recording data on a per method basis, if enough sample data has been recorded[1]. We expect a significant speed up. Other obvious improvements include the way we store data: The generator emits XML-files during execution. We are currently switching to a more performant binary file format. The static parallel dependency analysis is executed within a few milliseconds for each of the sample applications. However, despite the performance drawbacks described above, we can apply CHESS to the test cases generated, even if we cannot apply it to the whole application due to performance limitations. Furthermore, the test cases may be re-run deliberately. Even though we need to further evaluate our test case generator on large scale applications, we can already see that automatically dividing programs in smaller portions using a test based approach may improve the usability of current data race detectors.

D. Search Space Reduction

The quad tree sample application shows the usefullness of the static analysis: 12 parallel dependent method pairs have been found within the 58 methods. However, during dynamic parallelity analysis, 1,254 parallel running method pairs have been found. Without the static analysis, all of these would have to be instrumented for parallel data analysis; but as only 12 method pairs may share common data, we can reduce the instrumentation overhead to 1% only.

[1]After dynamic method analysis, we know for which method pairs and control flow paths we need to generate unit tests. During object state recording, we can disable recording for each methods after recording appropriate method calls.

VII. Related Work

Data Race Detectors: In our implementation we used the dynamic race detector Microsoft CHESS ([1]). Dynamic race detectors ([8], [9], [10], [11], [12]) monitor the program execution and look for races at runtime. For this reason they return few false positives, but they can only identify errors, when they ocurr. As races ocurr rarely, dynamic detectors have to be re-run often or for a long time. Another disadvantage is the slowdown of the program under test resulting from the runtime analysis.

Static race detectors ([13], [14]) produce large numbers of false positives. For static tools it is hard to detect data or control flow dependencies between objects or method pairs. Also it is hard to identify the parallel execution of methods correctly.

In [15], G. Szeder defines a framework for the execution of unit tests for Java programs. A unit test is regarded successful, if it produces the same output for any possible thread interleaving. It uses Java Pathfinder [16] as model checker to verify all possible execution branches.

Parallel Test Case Generators: In [17], A. Nistor et al. generate parallel unit tests for concurrent usage of public class methods. However, they don't generate tests for whole programs and don't test multiple class interaction. Additionally, they don't use existing race detectors but implement their own mechanism.

In [18], W. Wong et al. use of reachability graphs to generate parallel unit tests. The test cases are reduced via four criteria to priorize them and to sort them topologically. It also uses the *Stubborn Set Method* as model checker. It was applied to small programs but it does not scale to higher degrees of parallelism and bigger programs.

In [19], T. Katayama et al. map the behaviour of a parallel program onto to a graph structure containing the control

TABLE I
RUNTIME EVALUATION OF THE PARALLEL TEST GENERATOR IMPLEMENTATION.

Program	Bank Account	Async Queue	MT Printing	WaitHandle	ProgressBar	QuadTree	BoundedQueue	Argument Dependency
LOCs	25	147	20	14	31	520	21	21
Methods	4	19	2	2	5	91	5	3
Threads	8	61	10	2	100	15	6	6
Parallel Methods	4	5	1	3	3	58	5	2
Parallel Methods Pairs	9	30	1	2	6	1258	14	3
Runtime (ms)	47	150	236	6	N/A (GUI)	300	8	7
Runtime (ms) instrumented 1	120	2500	280	10	N/A (GUI)	28000	66	175
Runtime (ms) instrumented 2	830	5200	320	16	N/A (GUI)	57000	360	320
Depth Call Stack	3	4	2	2	4	27	2	2
Heap Objects	1	255	1	1	2	472	24	16
Test Cases Generated	5	17	1	1	3	10	10	3
Data Races	5	17	2	1	2	48	32	3
Data Races Detected	5	17	2	1	2	48	32	3

flow and explicit thread synchronizations. Via predefined test criteria this graph is reduced to unit tests.

In [2], Q. Luo et al. use static race detection and employ a capture-and-replay-technique to detect data races. In contrast to AutoRT, ConCrash produces unit tests only for program executions that led to a certain exception.

J.-D. Choi et al. track thread schedules of a parallel program and are able to replay them ([20]), which enables a deterministic execution and facilitates debugging of parallel programs. DejaVu itself does not use unit tests but could be used to capture and replay them deterministically.

VIII. CONCLUSION

This is work in progress and we currently work on extending the automatic test generator presented in this paper. For the future we motivate a combination of multiple race detectors at once for two purposes: (1) filtering false positives and (2) using detectors specialiced on certain error patterns.

In this paper we present AutoRT, a concept for automatic parallel unit test generation. We use a combination of static and dynamic program analysis to generate small test cases as input for current data race detectors. We have found all 110 races in our sample applications using our generated tests and the data race detector CHESS. We plan to extend our analysis runtime using dynamic instrumentation: current instrumentation is never changed during a single program run; using dynamic instrumentation, we can enable and disable object state recording deliberately. We will present results of our test case generator in combination with other data race detectors, such as TachoRace ([9]) and the .NET-implementation of Helgrind+.

ACKNOWLEDGMENT

The authors would like to thank Filip Dimitrov for his support during design, implementation and experimentation of AutoRT.

REFERENCES

[1] S. Q. Madanlal Musuvathi and T. Ball, "Chess: A systematic testing tool for concurrent software," Microsoft Research, Tech. Rep., Nov 2007.

[2] Q. Luo, S. Zhang, J. Zhao, and M. Hu, "A lightweight and portable approach to making concurrent failures reproducible," in *Proceedings of the 13th international conference on Fundamental Approaches to Software Engineering*. Springer-Verlag, 2010, pp. 323–337.

[3] A. Jannesari, M. Westphal, and W. Tichy, "Dynamic data race detection for correlated variables," 11th International Conference on Algorithms and A. for Parallel Processing, Eds. Melbourne, Australia, 2011.

[4] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav, "On the complexity of partially-flow-sensitive alias analysis," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 13:1–13:28, May 2008.

[5] G. Naumovich and G. S. Avrunin, "A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel," in *Proceedings of the 6th international symposium on Foundations of software engineering*. NY, USA: ACM, 1998, pp. 24–34.

[6] M. Research, "Phoenix academic program," 2007, http://research.microsoft.com.

[7] ——, "Cci: Common compiler infrastructure," 2009, http://ccimetadata.codeplex.com.

[8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," *IBM Systems Journal*, vol. 41, pp. 111 –125, 2002.

[9] J. Schimmel and V. Pankratius, "Exploiting cache traffic monitoring for run-time race detection," in *Euro-Par'11 Proceedings of the 17th international conference on Parallel processing - Part I*, 2011.

[10] A. Jannesari, K. Bao, V. Pankratius, and W. F. Tichy, "Helgrind+: An efficient dynamic race detector," in *Proceedings of the 23rd international Parallel & Distributed Processing Symposium (IPDPS'09)*. IEEE, 2009.

[11] D. Vyukov, "Relacy race detector," http://www.1024cores.net.

[12] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[13] Coverity. Coverity prevent.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the 2002 Conference on Programming language design and implementation*, NY, USA, 2002, pp. 234–245.

[15] G. Szeder, "Unit testing for multi-threaded java programs," in *PADTAD '09: Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. ACM, Jul 2009,.

[16] "Effective generation of test sequences for structural testing of concurrent programs," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 539–548.

[17] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012, 2012, pp. 727–737.

[18] W. Wong, Y. Lei, and X. Ma, "Effective generation of test sequences for structural testing of concurrent programs," in *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, june 2005, pp. 539 – 548.

[19] T. Katayama, E. Itoh, Z. Furukawa, and K. Ushijima, "Test-case generation for concurrent programs with the testing criteria using interaction sequences," in *Software Engineering Conference, 1999. (APSEC '99) Proceedings. Sixth Asia Pacific*, 1999, pp. 590 –597.

[20] J.-D. Choi and H. Srinivasan, "Deterministic replay of java multithreaded applications," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, ser. SPDT '98. New York, NY, USA: ACM, 1998, pp. 48–59.