Debugging Multithreaded Programs as if They Were Sequential

Xiaodong Zhang*, Zijiang Yang†, Qinghua Zheng*, Yu Hao*, Pei Liu*, Lechen Yu*, Ming Fan* and Ting Liu*,
*Ministry of Education Key Lab for Intelligent Networks and Network Security,
Xi'an Jiaotong University, Xi'an, Shaanxi 710000, China
Email: {xdzhang,yhao,pliu,lcyu,mfan}@sei.xjtu.edu.cn, {qhzheng, tingliu}@mail.xjtu.edu.cn
†Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA,
Email: zijiang.yang@wmich.edu

***Abstract*—Debugging multithreaded software is challenging because the basic assumption that underlies sequential software debugging, i.e. the program behavior is deterministic under fixed inputs, is no longer valid due to the nondeterminism brought by thread scheduling. In this paper, we propose a proactive debugging method to restore this basic assumption so that programmers can debug multithreaded programs as if they were sequential. Our approach is based on the synergistic integration of a set of new symbolic analysis and dynamic analysis techniques. In particular, symbolic analysis is used to investigate the program behavior under multiple thread interleavings and then drive the dynamic execution to new branches. Dynamic analysis is used to execute these new branches and in turn guide the symbolic analysis further. The net effect of applying this feedback loop is a systematic and complete coverage of the program behavior under a fixed test input. We have implemented the proposed method in a software tool called Proactive-Debugger. Our experiments show that Proactive-Debugger outperforms both ESBMC and Maple, two state-of-the-art testing tools for detecting and reproducing bugs in multithreaded programs.**

I. Introduction

Multithreaded programming is a key technique to unleash the full potential of present and future generations of parallel computing systems based on the use of multi-core processors. However, the intrinsic nondeterminism of parallel execution can result in concurrency errors that are difficult to detect, reproduce, and debug [1]. While most mainstream programming languages today support concurrency, their debugging tools were designed primarily for sequential software development. Indeed, there is a lack of practical tools for handling the unique challenges in debugging multithreaded programs.

The typical assumptions for sequential program debugging is as the following. Given a program P and an test input I, a correct execution of the program indicates that P is correct under the input I. In other words, if one execution of P does not reveal any error, P is deemed correct under I and therefore a different test input I' will be chosen to continue the debugging. On the other hand, if an execution of P under I reveals an error, we expect to reproduce the error by merely executing P under I again. Subsequently, the programmer can locate the erroneous statements responsible for causing the error, modifying them to fix the bug, and test the revised program P' under the same input vector I.

Unfortunately, the basic assumption underlying sequential software debugging is no longer valid for multithreaded programs. The reason is that, due to scheduling nondeterminism, one execution of the program may be different from another even under the same test input. Consider the example in Figure 1, which has two *concurrent* threads executing the two functions *foo()* and *bar()*, respectively. There are two local variables a and b and two shared variables x and y. We assume that the developer would like to test if it is possible for y to be negative when the execution of the two threads terminates. Under the input $\langle a = 1, b = 0\rangle$, execution $\pi : \langle 1, 2, 3, 6, 7, 10\rangle$ leads to $(y = 0)$ at the end, which satisfies the requirement of $(y \geq 0)$. However, unlike in sequential software debugging, the fact that the first execution of the program meets the requirement does not mean that the program is correct under this input, as there exists another execution $\pi' : \langle 6, 7, 1, 10, 2, 5\rangle$ that violates the aforementioned requirement (the value of y is -1 at the end of this execution). Conversely, if π' were to be executed first, it would indeed show an error in the program, but a replay of the program under the same test input – if it were a free run – would not be able to guarantee that π' is reproduced.

```
Thread 1:
foo(int a) {
1  x = a;
2  if (x > 0)
3      y = y + 1;
4  else
5      y = y - 1;
}
```

```
Thread 2:
bar(int b) {
6   y = b;
7   if (y > 0)
8     x = x + 1;
9   else
10    x = x - 1;
}
```

Fig. 1. Code snippet showing two concurrent threads, with two local variables a, b and two shared variables x, y.

The above example illustrates that debugging sequential programs and debugging multithreaded programs are significantly different. This because sequential programs and multithreaded programs have a number of distinguishing features [2], which result in the following three major technical challenges: *nondeterministic thread schedule*, *numerous thread interleavings* [1], and *difficulty in execution replay* [3].

In this paper, we propose an approach to restore the basic assumption of sequential software debugging for multithreaded programs so they can be debugged as if they were sequential programs. Toward this end, we develop a synergistic debugging framework, which has three new components: a symbolic analysis component, a branch scanning component, and a guided execution component (for dynamic analysis). The

978-1-5090-4517-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SATE.2016.20

```
Thread 1:              Thread 2:
1   x_w^1 = a;         6   y_w^2 = b;
2   if(x_r^1 > 0);     7   if(y_r^2 > 0);
3   y_w^1 = y_r^1 + 1  10  x_w^2 = x_r^2 - 1
```
Fig. 2. The SSA form of the execution trace π_1.

entire framework forms a synergistic loop, where the symbolic analysis and the dynamic analysis reinforce either other to guarantee the systematic and complete traversal of all program behaviors for a given test input.

This paper makes the following contributions:

1. We propose a proactive debugging framework for multithreaded programs, which allows us to simultaneously analyze multiple interleavings of the same execution trace. The resulting formula is suitable for constraint solving based on Satisfiability Modulo Theories (SMT) [4].

2. We propose a scanning method for locating the not-yet-explored branches of the multithreaded program, and then computing new *thread schedules* to guide the subsequent dynamic analysis of the program under the same input.

3. We implement the proposed methods in a software tool and evaluate it on a number of multithreaded benchmarks. Our empirical study shows that the new proactive debugging approach outperforms both Maple [5] and ESBMC [6], the two state-of-the-art tools for detecting/reproducing failing executions of a multithreaded program.

II. Motivating Example

In this section we use the program given in Figure 1 as a running example to go though the basic steps of our approach. Given an test input $I : \langle a = 1, b = 0\rangle$, we would like to check whether y can be negative at the end of any execution under I. For a trace $\pi_1 = \{1, 2, 3, 6, 7, 10\}$ and its all interleavings, $y \geq 0$ always holds. But it fails in the trace $\pi_{err} = \{6, 7, 1, 10, 2, 5\}$. Assuming that π_1 is the first execution trace obtained, we illustrate the subsequent steps needed for discovering the violation in π_{err}.

We first encode π_1 as a quantifier-free first-order logic formula, denoted $\varphi_{\pi 1}$, which captures not only π_1 but all the other possible interleavings of the instructions in π_1. To obtain such a formula, we first convert π_1 into the Static Single Assignment (SSA) form shown in Figure 2, where for each variable v, we use v_r^i/v_w^i to denote the i-th *read/write* of v. In the following, we present the three types of constraints that, when combined together, symbolically encode all possible interleavings of the execution trace in SSA form.

Program Semantics Constraint. This type of constraint specifies program semantics requirement. Since SSA form already gives unique indices to shared variables, the translation from program input and SSA trace to such constraint is straightforward. For instance, the following equation gives the program semantics constraint for the trace shown in Figure 2, where the first two terms encodes initial values of a and b.

$a = 1 \wedge b = 1 \wedge x_w^1 = a \wedge x_r^1 > 0 \wedge y_w^1 = y_r^1 + 1 \wedge$
$y_w^2 = b \wedge \neg(y_r^2 > 0) \wedge x_w^2 = x_r^2 - 1$

Memory Order Constraint. The equation below gives the memory order constraint of π_1, where symbolic variable o_i represents the possible position of the statement at Line i in a valid execution, and $o_i < o_j$ means the statement at Line i happens before the statement at Line j.

$(o_1 < o_2 < o_3) \wedge (o_6 < o_7 < o_{10})$

We consider the sequential consistency model only in this work, and therefore restrict the order of statement execution to follow the order of program code. For now, we leave the order between statements from different threads unspecified; that is, without considering the subsequent constraints, the instruction at Line 10 may actually be executed before the instruction at Line 1. Of course, for most programs, the order of the statements among different threads cannot be totally arbitrary. We will discuss the encoding of synchronization primitives in Section III-A, which eliminates the bogus interleavings. In general, the memory order constraint specifies a partial order (instead of a total order) manifested in a particular execution. This is the reason why this encoding leads to predicative analysis of multiple interleavings of the given execution trace.

Read-write order constraint. The data flow of thread local variables is well defined and clearly indicated by the SSA form. Under the sequential consistency model, a value read by a thread-local variable is the most recently written value to the same variable. Therefore, we only need to define the matching relations between *read* and *write* operations for shared variables. Consider x_r^1 at Line 2. It may read value from either x_w^1 at Line 1 or x_w^2 at Line 10. If x_r^1 reads x_w^1, the execution of Line 10 must either occur after Line 2 or before Line 1. Similarly, in order for x_r^1 to read x_w^2, the execution of Line 10 must occur between those of Line 1 and Line 2. The read-write constraint regarding x_r^1 is given below:

$\{x_r^1 = x_w^1 \wedge o_1 < o_2 \wedge (o_{10} < o_1 \vee o_{10} > o_2)\} \vee$
$\{x_r^1 = x_w^2 \wedge o_{10} < o_2 \wedge (o_1 < o_{10} \vee o_1 > o_2)\}$

For every read of a shared variable, we need a constraint similar to the equation above. In addition, we need to use read-write constraint to specify, for correctness checking, which version of the variable y has to be checked at the end of an execution. Let the last read of y, i.e. the third read, be y_r^3. Its corresponding value can be written by either Line 3 or Line 6, as specified below:

$(y_r^3 = y_w^1 \wedge o_3 > o_6) \vee (y_r^3 = y_w^2 \wedge o_3 < o_6)$

The equation states that the final value of y is written at Line 3 if it happens after Line 6; otherwise the value is from Line 6. With such specification, the requirement that y must be zero or positive can be written as $(\rho = y_r^3 \geq 0)$. We need to give distinct superscript to y because it is uncertain which access of y in the program is the last one.

Finally, we construct a formula by combining together the three types of constraints: program semantics constraint φ_{sm}, memory order constraint φ_{mo} and read-write constraint φ_{rw}. This leads to the encoding of all possible interleavings of the given execution trace π_1: $\varphi_{\pi_1} = \varphi_{sm} \wedge \varphi_{mo} \wedge \varphi_{rw}$. To check the satisfiability of $\varphi_{\pi_1} \wedge \neg\rho$, we can use an off-the-shelf SMT solver. Since this symbolic formula is unsatisfiable, we can conclude that none of π_1's interleavings leads to a violation of the requirement.

However, we cannot conclude that the program is correct under I yet, because not all possible paths of the program have been considered. Specifically, the reason why $\varphi_{\pi_1} \wedge \neg\rho$ dosen't covers π_{err} is because both this trace contains new

statements that have not yet been executed by π_1. For π_{err}, it is Line 5. Since the goal of our project is to examine all possible program behaviors under the given input, we have to include π_{err} in our analysis.

Consider the conditional statement $b_2 : x > 1$ at Line 2, where the true branch is executed in π_1. If there exists an interleaving that leads to the execution of the false branch of b_2, we will have a new base for symbolic analysis. Here, the challenge is to decide whether such new path is feasible and, if it is feasible, how to enforce its execution during the subsequent dynamic analysis (i.e., make it show up when we run the program). We will discuss the algorithm more formally in Section III-B.

As for this example, we create a constraint $\neg(x_r^1 > 0) \wedge (o_7 < o_2 \rightarrow y_r^2 \leq 0)$, which mandates that x_r^1 must be less than or equal to 0 in the new path. In addition, if the branch at Line 7, i.e. b_7, happens before b_2, the outcome of b_7 must remain unchanged. This constraint ensures that b_2 is the first negated branch in a new path. Of course, we cannot simply add this constraint to φ_{π_1} because it contradicts the term $(x_r^1 > 0)$ already in φ_{π_1}. This requires us to remove the branch-related constraints in the existing formula before new paths can be discovered. Let the revised formula be φ'_{π_1}. If it is unsatisfiable, b_2 cannot be the first negated branch in any paths. Otherwise, its solution gives schedule to a to-be-explored path. In our example, the new schedule is $s : \langle 6, 7, 1, 10, 2, 3\rangle$.

After the negation of b_2, the program state will be deviated from what has already happened in π_1. Therefore, a symbolic analysis based on π_1 gives random results after b_2. However, any instructions happening before b_2 are still valid because they still follow the same control flow as that in π_1. As a result, $s' : \langle 6, 7, 1, 10, 2\rangle$ is not only valid but also guarantees a negation at b_2. If we enforce a guided execution following s' we will obtain $\pi_{err} : \langle 6, 7, 1, 10, 2, 5\rangle$ in which $y \geq 0$ fails. The new trace will trigger further symbolic and dynamic analysis.

The above example illustrates, in a nutshell, how our approach works. Details will be covered in the subsequent sections. Specifically, in the next section, we will give a more formal presentation, including the encoding with synchronization primitives and the issues with negating branches. We will also explain in more detail the integration of symbolic and dynamic analysis, whose combined efforts enumerate all possible thread interleavings under a fixed test input.

III. Debugging Algorithms

Our approach integrates explicit executions with symbolic analysis to systematically explore program behavior under fixed inputs. Its pseudo-code, shown in Algorithm 1, consists of three components: guided execution, symbolic analysis, and branch scanning. We name our top algorithm ProactiveTesting because it attempts to detect errors automatically, even though by checking one test input it is not a general testing tool.

We maintain a set of to-be-explored schedules S, initially with one item. The algorithm terminates when either S becomes empty, which indicates that the program is correct with respect to property ρ under test input I, or a bug is found. Guided execution enforces an execution to follow a predefined schedule prefix s removed from S. An execution that goes beyond s becomes random. Note that the initial schedule prefix is an empty vector $\langle\rangle$, so the first path is a random execution from the beginning. Symbolic analysis encodes a particular execution trace to analyze different thread interleavings with the same set of instruction. The encoding of a trace is incapable of predicating the program behavior involving different instructions. The purpose of branch scanning is to compute particular schedule prefixes that lead to executions with different branches.

Algorithm 1 ProactiveTesting(Prog P, Input I, Prop ρ)

```
1: ScheduleSet S = {⟨⟩};
2: TestedSet T = ∅;
3: while S ≠ ∅ do
4:    PartialSchedule s = S.remove() ;
5:    π = GuidedExecution(P, I, s);
6:    if π.abstract ∉ T then
7:       T = T ∪ {π.abstract};
8:       φ_π=SymbolicAnalysis(π, ρ);
9:       BranchScanning(φ_π, S);
10:   end if
11: end while
```

We have to maintain the set of explored traces to prevent redundancy. With the assumption of program termination, such set is not needed for sequential program path exploration as a depth-first search or breath-first search can systematically explore all the branches. This is not true for thread interleavings. We will give detailed explanation in Section III-C.

A. Symbolic Analysis

Symbolic analysis transforms an execution trace π into a quantifier free first order logic formula $\varphi_\pi = \varphi_{mo} \wedge \varphi_{sm} \wedge \varphi_{rw}$, where $\varphi_{mo}, \varphi_{sm}$, and φ_{rw} denote memory order constraint, program semantics constraint, and read-write constraint, respectively. Let the property constraint be ρ. If $\varphi_\pi \wedge \neg\rho$ is satisfiable, its solution gives a schedule that our tool can follow to replay the error. This addresses the issue of execution replay for multithreaded programs in case of failure.

As stated in the section of related work, the encoding of execution paths is not new. Our encoding differs from existing work only in technical details rather than concept.

Memory Order Constraint ***(φ_{mo})***. This constraint specifies the potential ordering of the instructions in an execution trace π. In this paper, we consider sequential consistence memory model only. If instruction i happens before instruction j in π and both belong to the same thread, we enforce $o_i < o_j$.

The inter-thread ordering is guarded by synchronization primitives. In multithreaded programs, the most popular synchronization operations are *lock/unlock* and *wait/signal*. Consider two *lock/unlock* pairs on the same mutex. The following constraint mandates that one pair must be executed either before or after another:

$$\varphi_{mo}^{L[m]} = \bigwedge_{l_i/u_i, l_k/u_k \in L[m]} o(u_i) < o(l_k) \vee o(u_k) < o(l_i).$$

$L[m]$ denotes the set of *lock/unlock* pairs on mutex m, and $o(x)$ represents the order of synchronization operation x.

Given a condition variable cd, let WT be the set of wait operations on cd, and SG the set of signal operations on cd. The constraint for *wait/signal* is:
$\varphi_{mo}^{W[cd]} = \bigwedge_{w \in WT} \bigvee_{s \in SG} (o_w < o_s < o_{w'} \wedge m_s^w = 1) \bigwedge \varphi_{SG}^{WT} \bigwedge \varphi_{WT}^{SG}$,
where $o_{w'}$ denotes the next event of wait on cd immediately after w in the same thread, $o_w < o_s < o_{w'}$ indicates that a signal operation s must be executed between w and w', and $m_s^w = 1$ flags that s is mapped to w. Equation 1 defines φ_{SG}^{WT} and φ_{WT}^{SG}, in which φ_{SG}^{WT} enforces that each wait operation w needs to map to at least one signal operation, and φ_{WT}^{SG} restricts that each signal operation s signals at most one wait operation.

$$\begin{aligned} \varphi_{SG}^{WT} &= \bigwedge_{w \in WT} \{\{ \sum_{s \in SG} m_s^w \} \geq 1\} \\ \varphi_{WT}^{SG} &= \bigwedge_{s \in SG} \{\{ \sum_{w \in WT} m_s^w \} \leq 1\} \end{aligned} \tag{1}$$

Constraints on other types of synchronization primitives are modeled similarly. The conjunction of these intra- and inter-thread constraints relaxes the total order observed in an execution trace π.

Program Semantics Constraint ***(φ_{sm})***. The constraint maps executed individual instructions to corresponding formula. We skip detailed presentation as it requires mapping rules for complete LLVM syntax. Note that φ_{sm} enforces the same control flow for all encoded thread interleavings as a derivation leads to an execution with instructions unknown to π.

Read-Write Constraint ***(φ_{rw})***. The program semantics constraint considers each thread individually, in which each appearance of a shared variable has a unique index. The purpose of read-write constraint is to enumerate all possible matchings between read and write instructions of shared variables. Consider a shared variable v. Let $R(v)$ and $W(v)$ be the sets of reads and writes on v, respectively. We use v_r to denote the read of v at instruction r, and v_w the write of v at instruction w. In addition, let o_r and o_w be the order variables of r and w. The read-write constraint on v is:

$$\begin{aligned} \varphi_{rw}^v = \bigwedge_{r \in R(v)} \bigvee_{w \in W(v)} \{(v_r = v_w \wedge o_w < o_r) \wedge \\ \bigwedge_{x \neq w \in W(v)} (o_x < o_w \vee o_r < o_x)\} \end{aligned} \tag{2}$$

The constraint above describes that, if r matches w, then it must be executed after w, and there are no other writes to v in between. Let V be the set of shared variables, the read-write constraint is: $\varphi_{rw} = \bigwedge_{v \in V} \varphi_{rw}^v$.

B. Branch Scanning

The solving of φ_π is able to analyze the executions that involve exactly the same set of instructions of π. That is, let $B = \{b_1, b_2, \ldots, b_n\}$ be the set of branch instances in π, φ_π allows only the permutations of π that execute the same set of branch instances. Therefore, the unsatisfiability of φ_π does not imply correctness as there are feasible executions under I not covered by φ_π. In order to address this issue we search for valid executions that follow different branches.

Algorithm 2 BranchScanning(Formula φ_π, ScheduleSet S)

```
1: Let C = {c_1, c_2, ..., c_n} be the branch constraints in φ_π;
2: φ'_π = remove C from φ_π.
3: for each c_i in C do
4:    φ^{c_i}_π = φ'_π ∧ ¬c_i ∧ ⋀_{c_j ≠ c_i} (o_j < o_i → c_j)
5:    if φ^{c_i}_π is satisfiable then
6:       extract the schedule s_i up to branch c_i from the
         solution to φ^{c_i}_π.
7:       S = S ∪ {s_i};
8:    end if
9: end for
```

```
Thread 1:
1  int x=a;
2  if (x>0)
3    y = y+1;
   else
4    y = y-1;
```

```
Thread 2:
5  int y=b;
6  if (y>0)
7    x = x+1;
   else
8    x = x-1;
```

Fig. 3. Code snippet with input ($a = 1, b = 0$) and shared variables x, y.

The pseudo-code for branch scanning is given in Algorithm 2. At Line 2 we obtain φ'_π by removing all the branch constraints from φ_π. Then for each $c_i \in C$, we check whether it can be the *first* branch in π that can be negated. That is, any branches before c_i must produce the same outcomes same as in π. The potential path can be represented as
$\varphi_\pi^{c_i} = \varphi'_\pi \wedge \neg c_i \wedge \bigwedge_{c_j \neq c_i} (o_j < o_i \rightarrow c_j)$,
where o_i represents the order of c_i. If $\varphi_\pi^{c_i}$ is satisfiable, we extract its solution that gives the schedule up to c_i. Note that the schedule after c_i is invalid because the negation of c_i leads to unknown behavior that cannot be determined statically. However, the schedule prefix s_i up to c_i is valid and we save it as a to-be-explored schedule in S.

C. Guided Execution

We have implemented a thread scheduler to enforce a particular scheduling specified in a schedule vector s. There are two pieces of critical information in item $s[i]$: the thread id $s[i].tid$ and the instruction $s[i].ins$. The scheduling vector commands an execution to execute $s[i].ins$ of thread $s[i].tid$ at i-th step. In most cases s specifies only a prefix up to a certain step. After executing the last instruction in s the execution runs to complete randomly.

In Section III-B we have explained that backtrack-based systematic exploration such as DFS is not suitable for branch scanning. As a result we have to maintain a set T of explored traces to avoid repeated exploration of the same traces. Consider the code snippet in Figure 3 with test input ($a = 1, b = 0$) and shared variables x, y. Let the initial execution be $\pi_1 = \langle 1, 2, 3, 5, 6, 8 \rangle$. A search for alternative branches confirms that the branch at Line 2 can be negated, which leads to a schedule prefix $\langle 1, 5, 6, 8, 2 \rangle$. An execution following the prefix results in the second execution $\pi_2 = \langle 1, 5, 6, 8, 2, 4 \rangle$. The branch instance at Line 2 in π_2 can be negated as well, with a schedule prefix of $\langle 1, 2 \rangle$. Following such schedule prefix we may execute $\langle 1, 2, 3, 5, 6, 8 \rangle$, same as π_1. Without T Algorithm 1 may not terminate.

There are two challenges to maintain the set of explored traces. Firstly, the number of traces, even under a fixed test input, can be exponential to the number of instruction instances. Therefore recording all the explored traces can be extremely expensive. Secondly, recording explicitly executed traces is not sufficient. T has to include the traces implicitly explored by SMT solvers. To address both issues, we abstract an execution trace π to avoid recording complete traces and at the same time cover all the implicit traces derived from π. Let π_i be a subsequence by projecting π onto thread t_i. We partition π into a set $\pi = \{\pi_i | 1 \leq i \leq N\}$, where N is the number of threads. Let $\pi_i^B = \langle b_i^1 b_i^2 \dots b_i^k \rangle$ be the subsequence of branches in π_i. The trace abstract of π is defined as $\pi.abstract = \{\pi_i^B | 1 \leq i \leq N\}$. During guided execution, we need to keep abstracts of only those traces that are explicitly explored. This is sufficient because according to our algorithm: (1) explicitly explored traces must be different at some branches, and (2) traces with difference interleaving but same branch instances are covered by the same SMT solving procedure. A set T based on abstracts not only keeps much shorter sequences of individual traces but also gives exponential reduction in the number of traces.

IV. Experiments

We have implemented the proposed method in a software tool Proactive-Debugger built upon LLVM [7], KLEE [8] and Z3 [4]. It targets multithreaded C programs implemented with the POSIX thread library. Our empirical study is conducted on eleven benchmarks that are obtained from well-known application suites SPLASH2 [9] and PARSEC [10], as well as experimental objects in previous studies on bounded model checker ESBMC [6] and a trace simplification technique [11]. We have created buggy versions for the original programs, which are inserted into assertions.

We compare Proactive-Debugger against two widely used concurrent software testing tools ESBMC [6] and Maple [5] in terms of bug detection capability. All three tools are capable of automatically exploring different thread interleavings. For a fair comparison we restrict them to consider the same test inputs in our experiments. Under a given test input, Maple keeps records of tested interleavings and actively seeks to expose untested interleavings through delaying certain statements to increase interleaving coverage. ESBMC, on the other hand, is based on bounded model checking. It terminates after they either find an error, or explores all possible interleavings under the current bound of context switches among threads. We add constraints on fixed inputs to reduce the search space of ESBMC. All our experiments were conducted on a Linux 3.13.0 desktop with quad-core 3.2 GHz Intel CPU and 16-GB RAM.

The experimental results are shown in Table I, where Column *Name* lists the names of the programs under testing. Column LOC and #T give the line of code and the number of threads, respectively. We have to collect different types of data due to different natures of the tools. For Proactive-Debugger, Maple and ESBMC, whether the bugs are detected is given in Columns R. Column *Ram* and *time* show the memory consumption and time usage, respectively. For Proactive-Debugger, Columns *#I*, *#F* and *#SAP* list the number of executed instructions, the number of constraints and the number of SAPs on a trace, respectively. A SAP is a shared access point where shared variables are read or written. For ESBMC, Columns *CS=1* and *CS=2* give the experimental results when the bound of context switches is set to 1 and 2, respectively.

By running programs instrumented by PIN [12], Maple observes the pattern of inter-thread dependence through shared-memory accesses and orchestrates the thread schedule to execute untested interleavings with an active scheduler. Because it uses heuristics to diversify thread interleavings during repeated executions, Maple does not have consistent behavior for individual test cases. Therefore for each program we run Maple 30 times to obtain its bug detection rate. Before each run we delete the results from previous runs. The bug detection rates range from 33% to 100%, with an average rate of 92%. Note that we only report the running time for the cases where bugs are detected. It takes much longer time when the bugs are not detected. Maple consumes very little memory — it is sufficient to reserve 100KB before testing for all the experiments.

ESBMC fails to detect most of the bugs. When context switch bound is set to 1, ESBMC detects the bug in *account_bug* in 0.38 seconds but fails to detect any bugs in other benchmarks if we set the loop bound below 10. That is because the bugs cannot be triggered under the current bound on context switches and loops. If we set the loop bound above 10, ESBMC fails to terminate due to memory limit. When context switch bound is set to 2, ESBMC is able to detect bugs in the four small benchmarks but cannot terminate for *fft*, *luc*, *lunc*, and *radix*, as indicated by MO, due to tremendous memory consumption even for a trivial loop bound below 10.

Proactive-Debugger detects the bugs in all the experiments. Without considering the last benchmark it gains an average speedup of 5.17X over Maple. In Maple, the overhead of online profile and active scheduler is up to 100X [5]. Although running programs on KLEE makes the executions much slower, Proactive-Debugger conducts significantly less number of executions with the help of symbolic analysis that implicitly enumerates most of the thread interleavings. However, the experiment on *swarm_bug* shows that Proactive-Debugger may have severe performance penalty if the underlying logic is not suitable for SMT solving. In the case of *swarm_bug*, Proactive-Debugger terminates after 12074 seconds, which is above our 3600 seconds time limit, due to non-linear computations. Z3 has limited support for non-linear expressions at a very high cost [4]. Nevertheless, Proactive-Debugger is able to detect the bugs when it finally terminates. Proactive-Debugger consumes more memory than that of Maple, but much less than that of ESBMC. Indeed, Proactive-Debugger is a trade-off between the two techniques on opposite ends of testing spectrum.

V. Related Work

There is a large body of work on testing/debugging concurrent bugs, including random testing [13, 14], systematic testing [1, 5, 15], and active testing [16–18].

TABLE I
PROACTIVE-DEBUGGER VERSUS ESBMC AND MAPLE ON BUG DETECTION

Name	LOC	#T	Proactive-Debugger						Maple			ESBMC	
			R	#I	#F	#SAP	Ram(M)	time(s)	R	Ram(K)	time(s)	CS=1	CS=2
account_bug[6]	54	4	√	55	49	2	4.5	0.016	97%	100	3.1	0.38	0.9
arith_bug[6]	84	3	√	255	964	26	11	0.39	100%	100	5.7	×	2.5
queue_bug[6]	153	3	√	417	3,144	23	10	0.32	73%	100	4.5	×	1.27
stack_bug[6]	111	3	√	389	6,996	30	16	3.47	33%	100	2.9	×	1.25
fft_bug1[9]	1466	3	√	6.7k	2,284	199	402	3.5	100%	100	69.1	×	MO
fft_bug2[9]	1466	3	√	6.7k	1,981	196	586	5.7	100%	100	48.8	×	MO
luc_bug1[9]	1386	3	√	4k	1,673	302	713	7.8	100%	100	38.4	×	MO
luc_bug2[9]	1386	3	√	8.1k	2,854	584	1,604	14.5	100%	100	14.9	×	MO
lunc_bug1[9]	1155	3	√	25.5k	7,225	1,021	778	10.4	97%	100	34.3	×	MO
lunc_bug2[9]	1155	3	√	25.6k	6,099	1,015	1,028	11.5	100%	100	13.1	×	MO
radix_bug1[9]	1537	3	√	6.5k	3,573	321	225	3.1	100%	100	33.4	×	MO
radix_bug2[9]	1537	3	√	6.5k	2,292	303	209	2.2	97%	100	14.1	×	MO
pfscan_bug[11]	985	3	√	5k	789	731	89	0.87	100%	100	14.7	×	MO
blackscholes_bug[10]	620	4	√	8.1k	1267	315	16	0.27	100%	100	7.3	×	×
swarm_bug[11]	2249	5	√	15.2k	77158	802	2525	TO	100%	100	13.2	×	MO
Avg.	1023	3.3	–	7.6k	7789	518	491	4.1	92%	100	21.2	–	–

Random Testing. To exercise a more diverse set of thread interleavings, random delays need to be inserted at global memory access points to perturb their execution order [13, 14]. Although random testing based techniques are scalable, they do not provide the systematic and complete coverage; therefore, they often cannot reveal concurrency bugs whose symptoms occur only in rare interleavings [19].

Systematic Testing. Systematic testing techniques guarantee to visit one unique interleaving at a time, and reach a predefined coverage goal if given sufficient time. Generally speaking, these techniques fall into two categories: coverage-driven systematic testing and stateless model checking. Coverage-driven systematic testing techniques aim to reach a coverage criteria such as synchronization coverage [15] and inter-thread dependencies coverage [5]. Whereas stateless model checking based techniques aim to exhaustively cover all possible thread interleavings up to a fixed number of context switches [1]. Unfortunately, since the number of possible interleavings can be enormous, these techniques often do not work well on large programs and possibly miss corner bugs. In contrast, our new method can reach a sweet spot within the general framework of systematic testing by exploiting the benefits of both symbolic analysis and dynamic analysis while avoiding their shortcomings.

Active Testing. Active testing techniques aim at detecting certain type of concurrency bugs including deadlock [16], data-race [17], and atomicity violation [18]. For example, Maiya [17] proposed a happens-before analysis based method for detecting data-races in Android applications. Shacham [18] proposed a testing method for detecting atomicity violations in compositions of atomic operations of concurrent libraries. Although our current prototype in Proactive-Debugger handles assertion failures only, it is significantly more general than the prior techniques because assertion can be used to capture a wide range of concurrency bugs in practice.

VI. CONCLUSION

We have presented a proactive testing method aimed to address the main challenges of testing and debugging multithreaded programs. Under a fixed input, the testing and debugging of a multithreaded program is fully automated and hidden behind the scene. Compared to state-of-the-art techniques, our method shows its stronger abilitiy for detecting and reproducing bugs in multithreaded programs.

REFERENCES

[1] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *PLDI*. ACM, 2007, pp. 446–455.

[2] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur, "Towards a framework and a benchmark for testing tools for multi-threaded programs," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 267–279, 2007.

[3] G. Tomas and C. W. Ueberhuber, *Visualization of scientific parallel programs*. Springer Science & Business Media, 1994, vol. 771.

[4] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*. Springer, 2008, pp. 337–340.

[5] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *OOPSLA*. ACM, 2012, pp. 485–502.

[6] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *ICSE*. ACM, 2011, pp. 331–340.

[7] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO 2004*. IEEE, 2004, pp. 75–86.

[8] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ISCA*. ACM, 1995, pp. 24–36.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, ser. PACT '08. ACM, 2008, pp. 72–81.

[11] N. Jalbert and K. Sen, "A trace simplification technique for effective debugging of concurrent programs," in *FSE*. ACM, 2010, pp. 57–66.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.

[13] S. D. Stoller, "Testing concurrent java programs using randomized scheduling," *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 4, pp. 142–157, 2002.

[14] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *ASPLOS*. ACM, 2010, pp. 167–178.

[15] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *ISSTA*. ACM, 2012, pp. 210–220.

[16] Y. Cai and W. Chan, "Magicfuzzer: scalable deadlock detection for large-scale applications," in *ICSE*. IEEE Press, 2012, pp. 606–616.

[17] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," in *PLDI*. ACM, 2014, pp. 316–325.

[18] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav, "Testing atomicity of composed concurrent operations," in *OOPSLA*. ACM, 2011, pp. 51–64.

[19] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," in *ASPLOS*. ACM, 2009, pp. 25–36.