Finding Schedule-Sensitive Branches

Jeff Huang, Lawrence Rauchwerger
Parasol Laboratory
Texas A&M University, USA
{jeff,rwerger}@cse.tamu.edu

ABSTRACT

This paper presents an automated, precise technique, TAME, for identifying schedule-sensitive branches (SSBs) in concurrent programs, *i.e.*, branches whose decision may vary depending on the actual scheduling of concurrent threads. The technique consists of 1) tracing events at fine-grained level; 2) deriving the constraints for each branch; and 3) invoking an SMT solver to find possible SSB, by trying to solve the negated branch condition. To handle the infeasibly huge number of computations that would be generated by the fine-grained tracing, TAME leverages concolic execution and implements several sound approximations to delimit the number of traces to analyse, yet without sacrificing precision. In addition, TAME implements a novel distributed trace partition approach distributing the analysis into smaller chunks. Evaluation on both popular benchmarks and real applications shows that TAME is effective in finding SSBs and has good scalability. TAME found a total of 34 SSBs, among which 17 are related to concurrency errors, and 9 are ad hoc synchronizations.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids; Tracing; Diagnostics*

Keywords

Schedule-Sensitive Branches, Symbolic Constraint Analysis

1. INTRODUCTION

Branch statements play a fundamental role in computer programs. For sequential programs, branches are either invariant (*i.e.*, always true or false), or input-sensitive that their decision depends on the program input. However, branches in concurrent programs can have another dimension of sensitivity: *schedule-sensitivity*. Depending on the scheduling of threads, *the same branch instance* in a concurrent program may vary in two runs with the same input.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786840

```
T1                          T2
if(is_good){                if(speed>50){
  useProtocolA();             is_good = true;
}                           }
else{                       else{
  useProtocolB();             is_good = false;
}                           }
```

Figure 1: An example of schedule-sensitive branch

```
T1  if(referencedColumnMap==null) {...}
    else {
      if(referencedColumnMap.isSet(...)){...}
    }
T2      referencedColumnMap=null;
```

Figure 2: A real schedule-sensitive branch in Derby

To understand this phenomenon, consider an example in Figure 1. Thread `T1` uses different protocols to transfer files over the network depending on the current network speed, while Thread `T2` monitors the network speed periodically and updates the shared variable `is_good` which indicates the current network condition. The branch "`if(is_good)`" executed by `T1` is schedule-sensitive, since its choice also depends on the schedule.

Although the branch schedule-sensitivity seems intuitive, it is often unintended by the programmer and frequently the result of programming errors. Figure 2 shows a real bug in `Apache Derby` [2]. Thread `T1` first checks if the `referencedColumnMap` is `null`. If not `null`, `T1` will proceed to dereference it. However, `T2` may be concurrently running and set `referencedColumnMap` to `null`. The branch statement "`if(referencedColumnMap!=null)`" is schedule-sensitive, but it is not expected, because if `T2` executes immediately after `T1` executes this branch and before it dereferences `referencedColumnMap`, it will cause `NullPointerException` upon the dereference.

In our study of a large collection of popular multithreaded benchmarks and real programs (which we will show in Section 6), we find that schedule-sensitive branches are a strong indicator of concurrency errors – 50% (17 out of 34) of the schedule-sensitive branches we find in our experiments are resulted from concurrency bugs.

We anticipate that effectively finding schedule-sensitive branches (SSBs) is useful with at least two applications: (1) *program understanding*: if SSBs are not intended by the

programmer, then they represent good starting points for better understanding the program behavior; (2) *detecting and localizing potential concurrency errors*: it is likely that a SSB is caused by bugs.

In this paper, we present an automated technique, called TAME, to precisely identify SSBs in concurrent programs. TAME consists of three steps:

1. observes a fine-grained program execution trace;
2. constructs symbolic constraints for each branch in the trace; and
3. uses an SMT solver to find SSBs by solving the negated branch condition conjuncted with the symbolic constraints.

TAME is able to identify all the SSBs that can be inferred from the observed execution trace. Moreover, for each identified SSB, it generates a corresponding schedule that can enforce the program to execute a different branch choice. This feature also allows TAME to effectively test concurrent programs, as a different branch choice may manifest unexpected behaviors, such as the runtime exceptions in Figure 2.

A main challenge is how to scale to real programs that produce huge traces. TAME leverages concolic execution [11, 25] to delimit the traces to analyse. We further propose a distributed trace partition approach that scopes our analysis to a selected range of schedules, such that the corresponding constraints can be solved within a reasonable time. Although this approach may result in missing certain SSBs, it does not affect the premise of precision, *i.e.*, every identified SSB is truly schedule-sensitive. More importantly, it achieves a much better balance between analysis efficiency and effectiveness compared to a simple window-based trace segmentation approach [27, 14].

Our contributions are summarized below:

- We present a technique to precisely identify schedule-sensitive branches in concurrent programs based on symbolic constraint analysis and concolic execution.
- We present a distributed trace partition approach that scales our technique to real world programs with long running executions.
- We evaluate our technique using both popular benchmarks and real programs and show that it is able to analyze real world program executions containing tens of millions of instructions in a minute.
- We report, for the first time, 34 schedule-sensitive branches found in popular benchmarks and real programs. Among them, 17 are caused by concurrency bugs, and 9 are related to ad hoc synchronizations.

2. OVERVIEW

In this section, we start by illustrating our approach using an artificial example. We then identify the technical challenges and outline how we address them.

Figure 3 (top) shows an overview of our approach, consisting of three components: a tracer, a constraint builder, and an SMT solver. The tracer monitors the execution of a program and produces a trace of fine-grained events. The constraint builder takes the trace as input and constructs a set of constraints for each branch event, and invokes the SMT solver to determine if a branch is schedule-sensitive. Consider an example in Figure 3(a). The program contains two threads (`T1` and `T2`) accessing two shared variables (`x` and `y`) and three branches (marked as ❶ ❷ ❸). Branch ❶ at line 1 is only input-sensitive because Thread `T2` is only started after line 4. For branches ❷ and ❸ (at lines 6 and 11 respectively), their schedule-sensitivity is much harder to see, because in addition to the branch conditions, we also need to reason about thread schedules.

Suppose we run this program once with input `x=0`, `y=0` following a schedule denoted by the line numbers, we will observe `13` critical events (*i.e.*, shared data reads/writes and thread synchronizations) in the execution. Note that lines 2 and 12 are not executed in this schedule because their branch conditions are not satisfied, and line 1 generates two critical events (a read on `x` and a read on `y`). In our constraint model, we give each of these 13 events an order variable and each read access a symbolic value variable. We construct for each branch event a group of constraints over these variables. Let O_i denote the order variable of the event at line i, x_0 and y_0 the input value of x and y, and x_i and y_i the symbolic value of x and y, respectively, at line i. To avoid clutter, we use O_2 and y_2 to denote the order variable and symbolic value variable, respectively, for the read to `y` at line 1.

Figure 3(b) shows the constructed constraints for the three branches. There are two types of constraints. The first type is simple ordering constraints between critical events to ensure sequential consistency[1]. For example, $O_1 < O_2$ means that line 1 must happen before line 2. $O_4 < O_9$ because `T2` is forked at line 4 so its first event should happen after the `fork` event, and $O_5 > O_{14} \vee O_8 < O_9$ because the two `lock` regions cannot overlap. The second type is the constraints corresponding to the branch conditions. We perform dynamic symbolic execution to compute the path condition for each branch event and *negate* its corresponding branch condition. If the negated branch condition can be satisfied with any one valid schedule, then we know the branch is schedule-sensitive.

For shared data reads that propagate to the branch conditions, we use symbolic variables to denote their values, because they may read a value written by the same thread or by a remote write from a different thread. Therefore, the negated constraints for the three branches are written as `x1<=y2`, `x1>y2`$\wedge$`x3+1<=y6`, and `x10+2<=y11`, respectively. We then match each read with a valid write following the semantics of sequential consistency: a read returns the value written by the most recent write. For example, `y11` may either match with `y0` or `y7`. If it matches with `y0`, then either line 7 happens after line 11, or branch ❶ or ❷ is false. Otherwise if it matches with `y7`, branches ❶ and ❷ must be true and line 7 should happen before line 11. Therefore, we write the constraint as $(y_{11} = y_0 \wedge (O_{11} < O_7 \vee x_1 > y_2 \vee x_3 + 1 <= y_6)) \vee (y_{11} = y_7 \wedge (O_7 < O_{11} \wedge x_1 <= y_2 \wedge x_3 + 1 > y_6))$. The constraints for the other symbolic reads can be constructed in a similar way.

Putting all constraints together, we invoke an off-the-shelf SMT solver such as Z3 [6] or Yices [7] to solve them. For branches ❶ and ❷, their corresponding constraints cannot be satisfied, hence we cannot determine if they are schedule-sensitive or not based on the observed trace. For branch ❸,

[1]For a focused presentation, we only discuss sequential consistency in this paper. Nevertheless, our technique is generalizable to relax memory models.

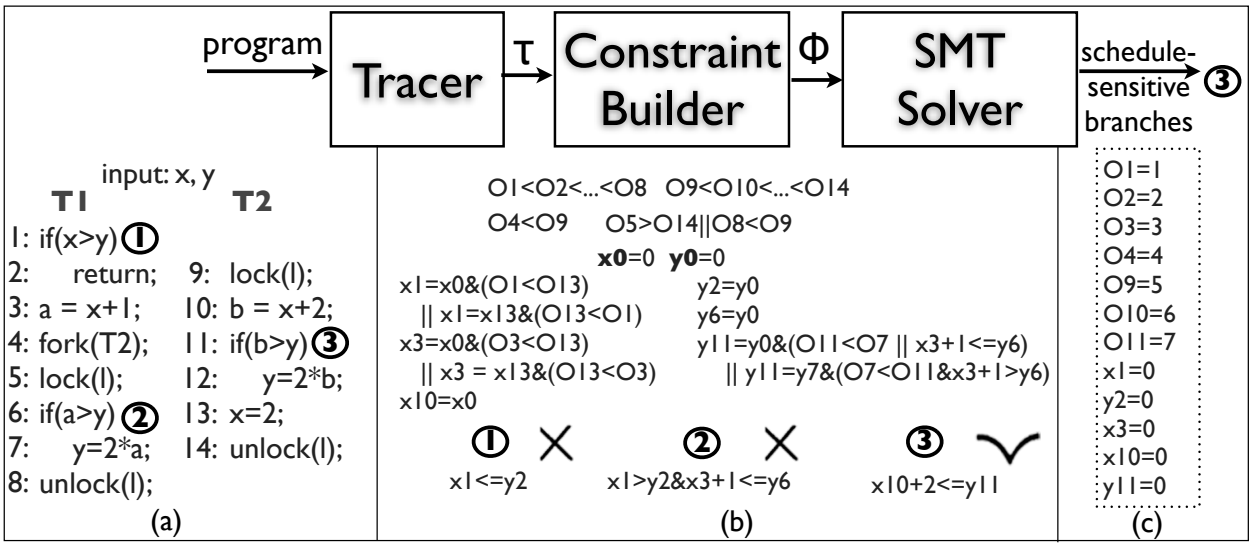

Figure 3: Technical overview of our approach. Branches ❶ ❷ are not schedule-sensitive, while Branch ❸ is.

the solver returns a solution shown in Figure 3(c), which corresponds to a schedule that can enforce branch ❸ to take the else branch upon re-execution of the program. We can then tell that branch ❸ is schedule-sensitive. Note that the solution returned by a solver may not be unique, indicating that there are multiple schedules that can enforce a branch to take a different choice. However, regardless of which one is returned, as long as there exists any solution, the considered branch is schedule-sensitive.

3. PRACTICAL CHALLENGES

Although the overall flow of our approach is easy to follow, there are several tough challenges that we must tackle before the approach can be applied to realistic programs. We next discuss these challenges and describe our solutions.

3.1 Fine Grained Tracing

To obtain the branch conditions, we need to trace not only those critical events, but also thread-local computations on the stack. Though thread-local computations do not directly introduce scheduling non-determinism, they could indirectly transfer the effect of non-deterministic data flow to branches. However, the number of thread-local computations is usually much larger than (*e.g.*, several orders of magnitude of) that of critical events, because real programs often use standard libraries which encapsulate complex thread-local functions. In addition, it is not always possible to trace every computation. Many programs contain native code or calls to external libraries of which the source code is not available or hard to instrument. Furthermore, the runtime overhead (including both memory and time) for tracing all computations can be prohibitive, which makes any tracing technique difficult to scale to long running programs.

Consider a simple (but almost full[2]) Java program in Figure 4. The program starts two threads to compare two different implementations of the SHA-256 hash algorithm on an input string. Thread `T1` uses the `Hashing` class from Google

```
String input = "esec/fse2015";
HashMap map = new HashMap();
Thread T1:
1. String hash1 = DigestUtils.sha256Hex(input);
2. checkHash(hash1);
3. if(T2.isAlive())
4.    System.out.println("done");
Thread T2:
5. String hash2 = Hashing.sha256()
6.           .hashString(input, Charsets.UTF_8)
7.           .toString();
8. checkHash(hash2);
checkHash(String hash){
9.  String hashed = map.get(input);
10. if(hashed == null){
11.    map.put(input, hash);
12. }
13. else if(!hashed.equals(hash))
14.    ERROR;
}
```

Figure 4: Example for fine-grained tracing

Guava[3], and Thread `T2` uses the `DigestUtils` class from Apache Commons Codec[4]. Both `T1` and `T2` call the method `checkHash` to check the computed hash string. In `checkHash`, a shared variable `map` is used to store and retrieve the hashed value. There are two `if` branches, at lines 10 and 13, respectively. Each thread first checks (at branch line 10) if a hash of the input string computed by the other algorithm exists or not. If not, the input hash will be stored into the map. Otherwise, the two hash strings will be compared (at branch line 13), and if they are different an error will be thrown at line 14. Regardless of the implementation correctness of the two hash functions, as long as they are both

[2]Only the main method and the thread creating statements are ignored to simplify the presentation.

[3]`https://github.com/google/guava/`

[4]`http://commons.apache.org/proper/commons-codec/`

```
1. x = new X(y);
2. if(x.f+y>0) doSomething;
```

Figure 5: Example for missing computations

deterministic, the branch at line 13 is not schedule-sensitive. However, the branch at line 10 is schedule-sensitive, because the thread which comes first will find that `hashed` is `null` and take the true branch, and the other thread will find `hashed` not `null` and take the false branch. To illustrate the issue brought by native code, we also add a branch statement at line 3, which checks if `T2` is still alive by calling `Thread.isAlive()`, and prints out a message if not. This branch is schedule-sensitive because the start and termination of `T2` depends on the schedule.

Although the program contains only less than 20 lines of code, a complete trace of its execution contains more than 390K computations if we trace every bytecode instruction (even after excluding the JDK libraries `java.lang.*` and `java.util.*`)[5]. The reason is that the library calls `DigestUtils.sha256Hex()` and `Hashing.sha256().hashString()` involve a myriad of subclasses and other libraries. Moreover, the runtime overhead for tracing all these computations is significant for such a tiny program. Without tracing, this program finishes in 20ms, whereas with tracing it takes more than 2 seconds (100X slower) to execute and generates a log of more than 100KB. We can imagine how serious this issue will be for large programs. Furthermore, it is difficult to trace the function call `Thread.isAlive()` because the code it executes is written in a different language (defined by Java native interface) and is not directly observable by the JVM.

To address the above issues, we propose to exclude tracing certain classes, which can be defined by the user and matched with regular expressions. For example, the user can specify `-exclude=java.*,com.google.*,org.apache.*` in the command line to instruct the instrumentation tool to not trace classes in these packages. This is a standard step used also in many other analysis frameworks [14, 4, 18]. It reduces both the trace size and runtime overhead, and also avoids the problem of tracing native code used in those excluded classes. For instance, after excluding the libraries classes in `java.*,com.google.*,org.apache.*`, the simple program in Figure 4 only generates 744 events and the tracing takes only 100ms. However, this approach raises a new problem: the computation information in the excluded classes is missed. Because it is unknown what is computed in the missing code, it is hard to obtain the branch condition if it is related to the missing computation. For example, in the code in Figure 5, suppose the class `X` is excluded and `y` is a thread-shared variable. At line 2, because the value of the field `x.f` is unknown, it is impossible to compute the branch condition and decide its schedule-sensitiveness.

To tackle this problem, we propose to insert additional value-tracking instrumentation to log the return value of method calls and every data read instruction (including the load operations on both heap and thread-local variables). The value information will help to recover the effect of missing computations in a manner of under-approximation: given the same input value the untracked code always produces the same output value. For example, in Figure 5, we may log at line 1 that the value of `y` is 0 and the constructor of `X` returns `void`, and at line 2 the value of `x.f` is 1 and `y` is 0. When computing the branch condition, we can use the logged value 1 to under-approximate `x.f`, *i.e.*, to conservatively assume that `X(y)` always takes `y==0` and returns `x.f==1`. In this way, despite that `X(y)` is excluded, the obtained branch conditions are sound (though it may limit the detection of SSB as additional constraints are introduced).

[5]In fact, we could not trace classes in these libraries due to limitations of the tracing tool we use (ASM and Java agent).

3.2 Runtime Exception Handling

When a runtime exception (either caught or uncaught) occurs, the program control flow will be transferred to a different point specified by the programmer. This causes two problems in constructing the branch conditions. First, the branch condition must consider the exception condition (*i.e.*, the condition for the exception to occur). If the exception condition is not met, the branch may not even be executed. For example, in the code below, if `x` is 0, a divide by zero exception will occur and hence the branch at line 2 will not be executed.

```
1. r=100/x;
2. if(r>0) doSomething;
```

Second, exceptions clutter the thread stack frame, which, if not properly handled, can fail symbolic execution. Consider an example in Figure 6. The method `m` which returns the quotient of 100 divided by an input integer `i` is called at line 3 with input 0. The divide by zero exception is thrown out of `m` and caught by the `try-catch` block at line 4. As we can see from the trace (shown in the figure, right), there is no indication that an exception has occurred in method `m` and the stack frame of `m` is exited. When symbolically executing the instruction "`ASTORE 1`" (store a reference at the top of the stack into a local variable at index 1), the current stack is still in method `m`, but there is no reference data in the stack! And when symbolically executing the instruction "`ILOAD 0`" (load an int value from a local variable at index 0), the value 0 (which is the input value to `m`) will be loaded from the stack frame of `m`, which is wrong. The correct value is 1, which is stored to `i` at line 1 in the stack frame of the caller of `m`.

```
1. int i = 1;                 ICONST_1
2. try{                       ISTORE 0
3.     i = m(0);              ICONST_0
4. }catch(Exception e){}      INVOKESTATIC m (I)I
5. int j = i;                 BIPUSH 100
                              ILOAD 0
                              IDIV  <-- INVOKE_EXCEPTION
6. static int m(int i){       ASTORE 1
7.     return 100/i;          ILOAD 0
8. }                          ISTORE 1
```

Figure 6: Example for handling exceptions

To address these issues, we propose to add additional events in the trace to recognize the occurrence of runtime exceptions. Specifically, for every method invoke statement, we insert a new statement `INVOKE_END` after it and enclose them within a `try-catch` block. If the method invocation returns normally, an `INVOKE_END` event will be logged. Otherwise if an exception is thrown from the invoked method, we log in the `catch` block an `INVOKE_EXCEPTION` event and

re-throw the exception. When performing symbolic execution on the trace, upon an `INVOKE_END` event, we pop up the stack frame of the invoked method and push its return value onto the new frame; upon `INVOKE_EXCEPTION`, because there is no return value in the invoked method, we pop up its stack frame and push a placeholder object onto the new frame to denote the exception object. In this way, the "`ASTORE`" instruction after the exception can be correctly executed.

In addition, we propose to capture exception conditions as additional branch constraints. Not all exception conditions can be captured (such as JVM internal errors). We currently handle two kinds of runtime exceptions: divide by zero, and array out of bounds, corresponding to bytecodes `{IDIV,LDIV,IREM,LREM}` (dividing an integer or long value) and `*ALOAD` (where `*` $\in$ `{A,B,C,D,F,I,L,S}` representing eight different types of array accesses), respectively. For each instruction in `{IDIV,LDIV,IREM,LREM}`, we mark the divisor as symbolic, say d, and create a branch constraint $d \neq 0$. For `*ALOAD`, we mark the array index value as symbolic, say i, and keep track of the array size, say S, and create a branch constraint $i < S$.

3.3 Data-Induced Control Flow

Besides explicit branches such as `if`, `while`, `switch`, etc, data flow that involves dereferencing a memory location can also implicitly affect control flow. Figure 7 illustrates the cases for object dereferencing and array indexing. At line 2, Thread `T1` calls `o.m()`, where depending on the schedule `o` may reference the object `C1` (created at line 1), `C2` (created at line 3), or `null` set by Thread `T2`. Hence, `o.m()` may execute a different method `m` or even throw a null pointer dereference exception. Similarly, the array access `a[x]=1` by `T1` at line 6 may write to either `a[0]` or `a[1]`, which makes the branch choice at line 8 non-deterministic. Therefore, to construct a sound branch condition, the data-induced control flow must be considered.

```
T1                      T2
1. o = new C1();        3. o = new C2();
2. o.m();               4. o = null;

5. x = 0;               7. x = 1;
6. a[x] = 1;            8. if(a[1]>0) doSomething;
```

Figure 7: Example for data-induced control flow

To capture data-induced control flow, we introduce additional branch events for object dereferencing and array indexing operations. Specifically, for shared non-primitive object dereferences `o.*`, we introduce a symbolic variable s_o (denoting the symbolic value of o) and add a branch condition $s = addr(o)$ where $addr(o)$ is the runtime address of o. For array reads and writes `a[i]`, we introduce a symbolic variable s_i (denoting the value of the index i) and add a branch condition $s_i = v_i$ where v_i is the runtime value of i. In this way, we not only ensure that the path conditions are captured, but can also find schedule-sensitive object dereferences and array accesses.

3.4 Precise Shared Data Identification

In constructing the constraints for matching reads and writes, we can filter out those thread-local or immutable data accesses, because their mapping is fixed. This also reduces the size of constraints and speeds up the solver. However, a caveat is that the address of every data access must be precisely identified. We must not miss any shared data access or match a read with a write on different addresses. Otherwise, the analysis result might be wrong. Consider the code in Figure 8. At line 9 the field `o2.x` is written to 1 by Thread `T2`. If `o2.x` is mis-identified as `o.x`, then the branch `if(o.x>0)` at line 3 will be miss-classified as schedule-sensitive. Similarly, if the write access `o.x=0` at line 8 is mis-identified as a thread-local access or on a different address other than `o.x`, the branch at line 3 will be miss-classified, because then the read of `o.x` at line 3 will be able to match with the write `o.x=1` at line 7.

```
T1                          T2
1. o.x = 0;                 6. synchronized(o){
2. synchronized(o){         7.   o.x = 1;
3.   if(o.x>0)              8.   o.x = 0;
4.     doSomething;         9.   o2.x = 1;
5. }                        10.}
```

Figure 8: Example for precise data identification

To precisely identify shared data accesses, we represent the address of heap access as follows. For array access `a[i]`, we represent its address as $addr(a)$`.i`, where $addr(a)$ is the runtime address of the array object `a`. For field access `o.x`, we represent as $addr(o).fid(x)$, where $addr(o)$ is the runtime address of `o` and $fid(x)$ is the field ID of `x` (a unique integer to the class of `o`). For static field accesses, since `o` is `null`, we set $addr(o)$ to 0.

3.5 Scalable Constraint Solving

There are two challenges associated with constraint solving in this problem: (1) complex branch constraints such as non-linear real arithmetics; (2) long execution traces which generate large number of constraints. For (1), advancements in theorem provers and decision procedures would be needed to efficiently solve such constraints, which is not the focus of this paper. A practical workaround we employ is to under-approximate the behavior of complex computations with the concrete runtime value. For example, for a branch `if(`x^2`+y>1)`, because the solver does not support non-linear arithmetic x^2, we repace x^2`+y>1` by a conjunction of `x =` v_x and v_x^2`+y>1`, where v_x is the recorded concrete value of `x`.

For (2), it affects the scalability of our technique. Although high-performance SMT solvers such as Z3 and Yices are becoming increasingly powerful, in theory they can only solve a limited number of constraints within a limited time. To improve the scalability of our technique, we propose to partition the input trace into smaller chunks such that the constraints generated for each chunk has a reasonable size. However, a strategy proposed in previous race detection work [27, 14] that simply segments the trace into windows of consecutive events (each containing `N` events) will not work well, because different from the trace for race detection which contains only critical events, our trace here contains many more fine-grained events. If `N` is not large enough, most events in a window will be from the same thread, which has little space for schedule exploration. On the other hand, if `N` is too large, the corresponding constraints will be too large to solve efficiently.

Distributed Trace Partition. To achieve a good balance between analysis effectiveness and efficiency, we de-

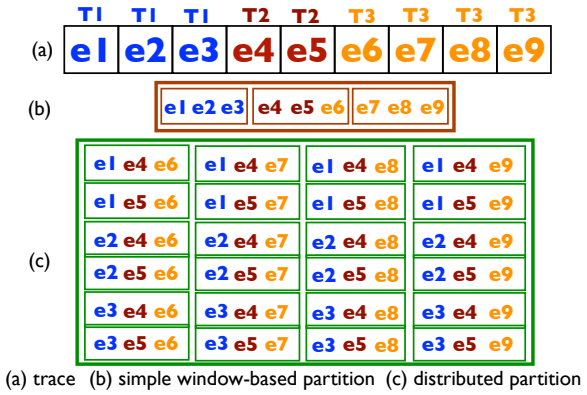

Figure 9: Illustration for trace partition

velop a distributed trace partition algorithm that segments the local trace of each individual thread into consecutive windows (rather than cutting a global trace), and combines windows from different threads to form a chunk. Specifically, for a trace τ, let τ_i denote the events by Thread T_i, and $\tau_i(j)$ the jth window in τ_i, each containing N events. A chunk is a union of $\tau_i(j)$ for all i with any j. There are in total $\prod_{i=1}^{K} M_i$ chunks, where K is the number of threads and M_i the number of windows in the trace of Thread T_i. For example, as illustrated in Figure 9(c), suppose there are three threads (K=3) and each window contains one event, then there are 24 chunks in total. Because each chunk contains an event from each of the three threads, the space of possible thread schedules formed by these three events is much larger than that produced by a global window of three consecutive events as shown in Figure 9(b). The limitation of this approach is that some SSBs may still be missed because schedules among events from different chunks are not explored. Nevertheless, it does not affect the precision: a branch that is determined to be schedule-sensitive is truly schedule-sensitive.

The trace can be partitioned either offline or online. For long running executions, because the full trace can be even too large to store, online partition is preferable. We skip the events by the main thread until the first child thread is created, because none of these events is schedule-sensitive. One additional issue is that the initial state for each chunk is unknown, which can break symbolic execution. When a full trace is available, we can store the final state of the current chunk as the initial state of the next one. However, this approach does not work when only one chunk of events is logged online. Fortunately, recall Section 3.1 that the value of every data read and method return is tracked. Similar to the treatment of untracked computations, we are able to use the logged runtime value to recover the initial states.

4. ALGORITHM

Our algorithm is summarized in Algorithm 1. For each chunk of events, we first perform a linear scan to find all the critical events (including shared heap accesses and synchronizations), and to symbolically compute the branch conditions for each branch event. Then for each branch event, we construct a set of constraints and invoke the solver. If the solver returns a solution, we report that the corresponding branch event is schedule-sensitive.

Algorithm 1 FindScheduleSensitiveBranch(τ)

```
1: Input: τ - a trace of events
2: Data structure: τ^c - τ containing only critical events;
3:    m - a map from branch events to branch conditions.
4: τ^c ← extractCriticalEvents(τ);
5: m ← computeBranchConditions(τ);
6: for b ∈ m.keyset do
7:    Φ_b ← extractNegatedPathCondition(m,b);
8:    τ_b^c ← getRelevantEvents(τ^c,b);
9:    Φ_b^c ← constructConsistencyConstraints(τ_b^c);
10:   if satisfiable(Φ_b ∧ Φ_b^c) then
11:      report b is schedule-sensitive.
```

Constraint Construction. Recall Section 2 that the constraints consist of two parts:

(i) Φ_b: the path condition for the branch event b conjuncted with its negated branch condition.

(ii) Φ_b^c: the consistency constraints among critical events.

For (i), the path condition of b is a conjunction of all preceding branch conditions by the same thread. The only unknown variables in Φ_b are the symbolic value variables introduced for each read access to shared heap locations. For (ii), the consistency constraints are similar to the thread causality constraints developed in previous work [13, 14, 12], except that the value of reads is not constrained, because the control flow consistency is already captured by the branch conditions. Specifically, Φ_b^c consists of the conjunction of three types of constraints: $\Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{rw}$, where Φ_{mhb} denotes the must-happen-before constraints, Φ_{lock} the lock-mutual-exclusion constraints, and Φ_{rw} the read-write constraints over read and write events. For space reasons, we refer the readers to GPredict [13] for detailed description of Φ_{mhb} and Φ_{lock}. We next describe Φ_{rw}, which is different from previous work.

Read-write constraints (Φ_{rw}). Consider the Read and Write events on the same shared data. For a Read, it may read the value written by a Write by the same or a different thread, depending on the order relation between the Writes. Consider a Read r, and let W denote the set of Writes on the same location as that of r, and V_r the value returned by r. Φ_{rw} is written as:

$$\bigvee_{\forall w_i \in W} (V_r = w_i \wedge O_{w_i} < O_r \bigwedge_{\forall w_j \neq w_i} O_{w_j} < O_{w_i} \vee O_{w_j} > O_r)$$

The constraint above states that, if a Read is mapped to a Write, for this Write, its order is smaller than that of the Read, and there is no other Write that is between them. We group all the Reads and Writes by the accessed memory address, and encode Φ_{rw} for each Read.

Constraint Complexity. Let N_r and N_w denote the number of Reads and Writes on a certain shared address, the size of Φ_{rw} is $4N_rN_w^2$, which is cubic in the number of Read/Write events in τ_b^c.

Optimizations. In practice, the size of Φ_{rw} can be significantly reduced by taking the must-happen-before relation $\prec$ into consideration. Consider two Write events w_1 and w_2, and $w_1 \prec w_2 \prec r$. We can ignore w_1 because it is impossible for r to read the value written by w_1. Another optimization is that we do not need to repeatedly check for branch events corresponding to the same branch location. Once a branch

location is determined to be schedule-sensitive, we can skip all th rest branch events on it.

5. IMPLEMENTATION

TAME is implemented based on ASM [1] and Z3 [6] and works for Java programs. The architecture is inspired by CATG [28], a Java concolic unit testing engine. We extend CATG to handle large real concurrent programs. TAME currently traces at the Java bytecode level and supports all opcodes [3] in Java 7 excepts `INVOKEDYNAMIC` (Opcode 186), including array/field accesses, loads/stores to local variables, method invocations, stack operations, etc. For different types of instructions, we log different runtime data such as the Thread ID and the value of loads and stores. For debugging purpose, for all instructions we also maintain a map from a unique instruction ID to its location (class and line number). We next describe the instructions related to critical events:

- Heap accesses: all the field and array loads and stores. For field accesses, there are four different instructions: `GETSTATIC`, `PUTSTATIC`, `GETFIELD`, and `PUTFIELD`. For each field access instruction, we log its intruction ID, class ID, field ID, read/write value, and address of the object if not static. For array access, there are 16 different instructions: `*ASTORE` and `*ALOAD`, where `*` denotes eight different data types including the reference type and seven primitive types. For each array access instruction, we log its intruction ID, address of the array object, and index value.
- Thread synchronizations: Thread `start/join/wait/notify`, and `lock/unlock` events (corresponding to `MONITORENTER/MONITOREXIT` instructions). For synchronized methods, as there is no corresponding `MONITORENTER/MONITOREXIT` bytecode instruction, we log at the beginning and every return instruction of the method to indicate `lock/unlock` events. For each synchronization instruction, we log the ID of the participating threads and address of the lock object.

In addition to events corresponding to the bytecode instructions of the original program execution, the trace also contains the inserted new events for handling runtime exceptions and data-induced control flow as discussed in Section 3, including `INVOKE_END` and `INVOKE_EXCEPTION` to recognize runtime exceptions and two special events: `BRANCH_SPECIAL` and `MARK_SYMBOLIC`. We insert `BRANCH_SPECIAL` after every branch instruction to indicate if the true branch is taken, which is used to build the correct branch conditions. We insert `MARK_SYMBOLIC` before every load access to shared heap locations. This is done by pre-processing the trace before constructing the constraints. We mark a heap load access as symbolic if in the logged trace (or chunk of events) there exists at least one heap store to the same address and by a different thread. When performing symbolic execution, for each `MARK_SYMBOLIC` event, we introduce a new symbolic variable to represent the value returned by the load access.

6. EVALUATION

We have applied TAME on a variety of popular multithreaded benchmarks and several real world large complex programs shown in Tables 1 and 2. All these programs were collected from recent concurrency studies [14, 13, 10, 23], with the total size close to 1MB. The main goal of our evaluation is to answer two research questions:

1. How effective and efficient is our technique for finding schedule-sensitive branches?
2. How scalable is our technique when applied on real programs with long running executions?

For the first question, we ran TAME on a collection of benchmarks that produce traces with a relatively small size after excluding the events in the JDK libraries. We set the bound to 100K events and 1K critical events, to make sure that TAME can finish within a reasonable time. For the second question, we ran TAME on a collection of large multithreaded applications including `Jigsaw-2.2.6`, `Derby-10.3.2.1`, `H2-1.4.178`, `FTPServer`, `Cache4j`, `Log4j`, `Hedc`, `Weblech`, `Pool`, as well as several long running benchmarks. Because the traces of some of these programs contain tens of millions of events, it is challenging to even store and load the whole trace. We hence performed the trace partition strategies online (as explained in Section 3.5) to log only one chunk of events in each run. This turned out to work well in practice because there are often many redundant events across chunks that a single chunk can often reveal much information about the whole trace.

All experiments were conducted on an 8-processor 32-core 3.6GHz Intel i7 Linux with 8GB memory and JDK 1.7. We set the Java heap space to 8GB and Z3 timeout to one minute. All data were averaged over three runs.

Overall results. TAME is effective for finding SSBs, and has good scalability to real world programs and long running executions when our distributed trace partition approach is applied. For the 15 smaller benchmarks, TAME was able to analyze all the traces in less than two minutes and found a total of 20 SSBs with 8 of them related to concurrency errors. For most real programs and larger benchmarks, TAME could not finish analyzing the full trace in a reasonable time. With our distributed partition approach, however, it was able to analyze all these programs in around ten minutes and found a total of 14 SSBs in which 9 are related to concurrency errors. We present these SSBs in Section 6.3.

6.1 Effectiveness and Efficiency

Table 1 summarizes the results on the smaller benchmarks. Columns 3-6 report the trace characteristics (the numbers of threads, events, critical events, and branch events). The branch events include both the explicit branching events and those abstracted from the exception conditions and data-induced control flow. Column 7 reports the online tracing time for each benchmark. Column 8 reports the average size of the constructed constraints for the branch events, and Column 9 the total offline constraint analysis time (including both the constraint construction and solving). Column 10 reports the number of schedule-sensitive branches found in each benchmark and the number of those related to concurrency errors . Note that each reported schedule-sensitive branch has a unique program location. Redundant schedule-sensitive branch events on the same location are filtered out.

TAME is highly effective in finding SSBs in these benchmarks of which the traces have a manageable size. The number of threads in these traces ranges between 2 and 27. The online tracing time ranges from a few milliseconds to 2s. The average constraint size ranges between 1.5KB to 2.2MB, and the total offline trace analysis time ranges from

Table 1: Results on smaller benchmarks. The last column "()" refers to harmful #SSB.

Program	LoC	#Thread	#Event	#Critical	#Branch	Tracing	Avg Cons	Solving	#SSB
Example	37	2	121	15	4	7ms	1.5KB	148ms	**1**
Account	373	3	873	55	13	519ms	9.6KB	2ms	**1(1)**
Airline	136	11	1123	77	21	85ms	125KB	885ms	0(0)
Allocation	348	25	13440	757	503	193ms	89K	1s	0(0)
Critical	63	3	285	27	11	243ms	3.2KB	892ms	**2(1)**
MTList	5979	27	7555	584	480	408ms	152KB	17s	0(0)
MTSet	7086	22	8569	518	394	377ms	97KB	5.5s	0(0)
StringBuffer	1339	3	674	43	22	47ms	5.9KB	287ms	**4(2)**
BufWriter	255	7	7803	246	108	181ms	229KB	30.7s	**1(1)**
LinkedList	425	7	1105	54	32	45ms	7KB	443ms	0(0)
Deadlock	135	4	809	34	19	24ms	3.5KB	277ms	0(0)
Piper	211	5	1238	130	41	46ms	28.5KB	398ms	**4(1)**
Loader	129	11	3634	506	306	2s	293KB	9.3s	**3(2)**
Shop	236	4	4040	306	173	93ms	2.2MB	16.7s	**3(0)**
Sor	7175	3	13478	1029	690	187ms	178KB	15s	**2(0)**
Philo	78	3	1428	151	58	39ms	52KB	696ms	0(0)
Total: bench	24K	140	67K	4532	2875	4.7s	-	98s	**20(8)**

Table 2: Results on real programs and large benchmarks - with distributed trace partition 10K. For benchmarks marked with *, TAME without distributed partition either ran out of memory or timeout in an hour.

Program	LoC	#Thread	#Event	#Critical	#Branch	Tracing	Avg Cons	Solving	#SSB
Pool107	4402	3	1634	58	15	178ms	8.1KB	326ms	**2(2)**
Pool146	2091	2	5163	245	58	287ms	62.5KB	1.4s	0(0)
Pool149	2767	3	1610	87	24	215ms	16.2KB	433ms	**1(0)**
Pool162	2917	2	4219	136	18	668ms	32.8KB	592ms	**1(0)**
Log4j	3792	3	7996	363	248	405ms	6.1KB	2.4s	**2(2)**
Weblech	35K	3	11455	272	166	2.2s	46KB	1.7s	**1(1)**
Cache4j*	1797	2	17064368	60	2	3.3s	85KB	673ms	0(0)
Hedc*	30K	10	910593	235	595	1.6s	54KB	7.5s	0(0)
FTPServer*	32K	11	467707	1668	1202	6.2s	325KB	36s	0(0)
Jigsaw*	381K	13	27369096	537	263	5.5s	137KB	6.5s	**2(2)**
Derby*	302K	3	15908795	106	70	7.1s	144KB	4.1s	**2(2)**
H2*	136K	14	45531806	395	136	6.5s	180KB	5.8s	0(0)
Total: real	932K	69	108M	4162	2797	34.5s	-	67.4s	**11(9)**
Tsp*	444	3	60775151	37	7	1.9s	4KB	282ms	0(0)
Garage*	545	7	1686507	1399	1061	2.1s	475KB	40s	**1(0)**
Elevator*	336	3	65980	1227	664	182ms	245KB	14.6s	0(0)
Moldyn*	2887	2	2835088	640	98	3s	363KB	8.1s	**2(0)**
MonteCarlo*	2887	2	36686043	185	66	6s	211KB	4.6s	0(0)
RayTracer*	2887	2	173191	1355	1302	1.9s	397KB	651s	0(0)
Total: bench	10K	19	102M	4843	3198	15.1s	-	718.6s	**3(0)**

2ms to 30.7s. For most benchmarks, TAME detected 1-4 SSBs in each of them, with the total amount to 20.

6.2 Scalability Results on Real Programs

Table 2 summarizes the results on real programs and those benchmarks that produce large traces. For half of the real programs, the trace size is relatively small that TAME finished the analysis within a few seconds and found 7 SSBs in them. For the other half (marked with "*"), the trace size is much larger (ranging between 467K in `FTPServer` to 45.5M in `H2`), and TAME was not able to finish analyzing the whole trace in a reasonable time (it either ran out of memory or timeout in an hour). To understand the effectiveness of our distributed trace partition approach, we performed online both the distributed partition (with the chunk size of each thread set to 10K) and the simple window-based approach (with the total window size set to 100K) and compared between them. It turned out that our distributed partition algorithm is efficient, and much more effective than the simple window-based approach. Columns 5-10 in Table 2 report the corresponding results for TAME with our distributed partition approach. TAME was able to analyze all these programs within a minute, and detected four SSBs in `Jigsaw` and `Derby`, whereas TAME with the window-based approach did not find any SSBs (though took less time). For the others, although TAME did not detect SSBs based on the logged chunk of events, TAME was able to analyze them within a minute.

For the large benchmarks, their trace size ranges from 66K to 60M and, similar to many real programs, TAME could not finish analyzing the whole trace. However, with distributed online trace partition, TAME was able to analyze them in a

Table 3: A summary of schedule-sensitive branches found. I – bug. II – ad hoc synchronization. III – unclear.

ID	Program	Class	Method	Line	Statement	Type
1	Account	Account	checkResult	78	if(Bank_Total==Total_Balance)	I
2	Critical	Critical	run	59	if(t.turn!=0)	I
3		Critical	run	75	while(t.turn!=1)	II
4	StringBuffer	StringBuffer	getChars	327	if(srcEnd<0 || srcEnd>count))	I
5		StringBuffer	append	446	if(newcount>value.length)	I
6		StringBuffer	delete	662	if(end>count)	III
7		StringBuffer	delete	668	if(len>0)	III
8	BufWriter	BufWriter	main	90	if(res!=0)	I
9	Piper	Piper	fillPlane	46	if((_last+1)%NUM_OF_SEATS==_first)	I
10		Piper	fillPlane	49	_passengers[_last]=name	II
11		Piper	emptyPlane	62	while(_first==_last)	II
12		Piper	emptyPlane	65	name =_passengers[_last]	II
13	Loader	Loader	main	50	while(!NewThread.endd)	II
14		Loader	main	58	if(array[i]>array[i+1])	I
15		NewThread	run	42	if(array[i]>array[i+1])	I
16	Shop	Shop	getItem	26	storage[items]	III
17		Shop	putItem	32	storage[items]	III
19		Shop	isEmpty	62	if(items==-1)	III
19	Sor	CyclicBarrier	doBarrier	227	else if(index==0)	II
20		CyclicBarrier	doBarrier	269	else if(r!=resets_)	II
21	Garage	GarageManager	WaitForManager	457	if(!status.IsManagerArrived())	III
22	Moldyn	TournamentBarrier	DoBarrier	65	while(IsDone[myid+i*spacing]!=donevalue)	II
23		TournamentBarrier	DoBarrier	78	while(IsDone[0]!=donevalue)	II
24	Pool107	CorsorableLinkedList	next	975	return _next	I
25		SimpleFactory	makeObject	162	if(activeCount>maxActive)	I
26	Pool149	GenericObjectPool	allocate	1117	if(isClosed()	III
27	Pool162	GenericObjectPool	allocate	1427	if(isClosed()	III
28	Log4j	Category	callAppenders	202	if(c.iia!=null)	I
29		WriterAppender	checkEntryConditions	181	if(this.layout==null)	I
30	Weblech	Spider	isRunning	103	return (running==0)	I
31	Jigsaw	HttpMessage	getHeaderValue	186	if(d!=null&&d.offset>=0)	I
32		HeaderDescription	getHolder	40	cls.newInstance())	I
33	Derby	TableDescriptor	getObjectName	780	if(referencedColumn Map)	I
34		TableDescriptor	getObjectName	806	if(referencedColumnMap.isSet(...))	I

few minutes and found 3 SSBs in `Garage` and `Moldyn`. The one that TAME took the most time is `RayTracer`. The reason is that the trace of `RayTracer` contains a large number of (1355) critical events and a large number of (1302) branch events. The trace produces many (1302) large constraint files (400KB on average) but none of them is satisfiable.

6.3 Schedule-sensitive Branches Found

In our experiments, TAME found a total of 34 SSBs, 17 of them are indications of concurrency bugs, 9 related to ad-hoc synchronizations, and the rest are either functional requirements or their intended behavior is unclear. We note that all these programs have been frequently studied before [14, 13, 10, 23], but no previous work reported SSBs. Our work is the first to report SSBs in these programs.

Table 3 summarizes these SSBs. Columns 2-5 report for each SSB the class, method, line number, and the signature, respectively. The last column reports the type of the SSB: "I" denotes concurrency bug, "II" ad hoc synchronization, and "III" unclear. We next describe several interesting SSBs.

StringBuffer We found four SSBs in this program, two of them are lated to concurrency bugs. The first one is in method `getChars` at line 327 "`if(srcEnd<0 || srcEnd>count)`". The value of the shared variable `count` can be changed by concurrent threads, which makes the branch choice nondeterministic. This SSB is a direct manifestation of the concurrency bug in this program (the true branch throws `StringIndexOutOfBoundsException`). The second SSB is in method `append` at line 446 "`if(newcount>value.length)`". The value of the local variable `newcount` depends on `count`. This SSB is also an indication of the concurrency bug. The third and fouth SSBs are both in method `delete`, at lines 662 "`if(end>count)`" and 668 "`if(len>0)`" respectively. From the semantics of this method, both SSBs are intended.

Pool107 We found two SSBs in this program, one in class `org.apache.commons.pool.impl.CorsorableLinkedList` at line 975 "`return _next;`", and the other in method `makeObject` of class `pool107$SimpleFactory` at line 162 "`if(activeCount>maxActive)`". In the first SSB, `_next` is a shared variable, and we found that it may return a `null` or non-`null` reference depending on the schedule. The second SSB is actually a manifestation of the concurrency bug in this program, which causes a runtime `IllegalStateException`.

Pool149 and ***Pool162*** We found a SSB "`if(isClosed()`" in each of these two programs. Both SSBs are in method `allocate` of class `org.apache.commons.pool.impl.GenericObjectPool`, but at different lines (1177 in `Pool149` and 1427 in `Pool162`). The method call `isClosed()` can return either true or false depending on the schedule. It is not clear if this behavior is buggy or not, though.

Jigsaw We found two SSBs in `Jigsaw`. One in method `getHeaderValue` in class `org.w3c.www.http.HttpMessage` at line 186 "`if(d!=null&&d.offset>=0)`". The variable `d` references a shared `HeaderDescription` object and its field `offset` may be set to 0 by another thread concurrently. This SSB indicates a vulnerability. The other SSB is in method

`getHolder` of class `org.w3c.www.http.HeaderDescription` at line 40 "`cls.newInstance()`". The shared variable `cls` may be `null` or non-`null` depending on the schedule, which also indicates a concurrency bug.

Derby We found two SSBs in `Derby`, both in method `getObjectName` of class `org.apache.derby.iapi.sql.dictionary.TableDescriptor` at line 780 "`if(referencedColumnMap)`" and line 806 "`if(referencedColumnMap.isSet(...))`". In fact both SSBs reveal the same concurrency bug (which is previously known) that the shared variable `referencedColumnMap` can be set to `null` concurrently.

6.4 Limitations

We note that TAME currently has two limitations that we plan to address in our future work.

Input-sensitivity Being a dynamic trace based approach, TAME may not find all schedule-sensitive branches (SSBs) in the program, but only those that can be inferred from the observed trace information, which is sensitive to the test input. Enhancing TAME with test input generation will help find more SSBs that can only be revealed by different inputs.

Relaxed memory models TAME currently only models sequential consistency, though the Java memory model is not sequentially consistent. We plan to develop weak memory constraints in TAME to find SSBs in systems exhibiting relaxed memory model behaviors.

7. RELATED WORK

To our best knowledge, our work is the first to focus on finding schedule-sensitive branches in concurrent programs. Unlike conventional consistency criteria such as data races [22] and atomicity violation [20] which are based on interleaving patterns, branch schedule-sensitivity is more effect-oriented and may serve as an alternative criterion for concurrency bug detection.

Our technique belongs to the school of predictive trace analysis [30, 13, 15, 17, 14], which has been shown promising for practical concurrency defect analysis. Representative techniques include race detection [5, 14], deadlock detection [8], and finding null pointer dereferences [10]. Our work expands the scope to analyze branch behaviors.

There are two lines of related work: ad hoc synchronization finding and concolic testing of concurrent programs. Ad hoc synchronizations are often witnessed together with schedule-sensitive branches. However, they are not the same. As shown in our experiments, half of the schedule-sensitive branches we found are indications of concurrency errors. In addition, ad hoc synchronizations lack a precise definition and are hard to find because of their behavioral semantics. There exist a few techniques to identify ad hoc synchronizations statically [32] or dynamically [29]. For instance, SyncFinder [32] relies on a few heuristics yet it is imprecise and may report false alarms. To refine our technique for finding concurrency errors, we can integrate with SyncFinder or the other techniques [29] to sift out ad-hoc synchronizations.

Concolic execution, first developed in DART [11] and Cute [25], has been the golden approach to test sequential programs facing complex constraints. Our technique leverages concolic execution to construct branch conditions facing the missing computations in native code or excluded libraries. When a computation is missed, its concrete value is used to construct a sound constraint. The effectiveness of our technique can be further improved by exploring multiple traces driven by concolic execution, which is still under active research [26, 19] to improve efficiency and code coverage.

Several concolic testing techniques [9, 21, 24] have been proposed for generating both inputs and schedules for concurrent programs. For generating schedules, jCute [24] takes a race-direct way that re-orders the events involved in data races. Similar to our technique, both ConCrest [9] and the work [21] encode scheduling constraints and use SMT solving to generate tests. While the work [21] combines data flow constraints from multiple traces, ConCrest focuses on one trace and explores the scheduling space by iteratively expanding the interference scenarios formed by shared data reads and writes. A difference between our technique and ConCrest is that we consider the events by chunks instead of interference scenarios. This reduces the number of invocations to the solver when a branch is not schedule-sensitive. Moreover, because we do not need to generate inputs, our technique achieves much higher scalability than ConCrest.

At the heart of our technique is symbolic trace analysis combined with concolic execution. Many symbolic trace analyses have been proposed before that extract a model from the execution trace in terms of first-order logical constraints and use SMT solving to find concurrency bugs [31, 10, 14], reproduce concurrency failures [16], and test concurrent programs [9, 12]. The construction of symbolic constraints in our work is similar to that in [14, 13], except that reads and writes are matched through the use of branch conditions rather than by the recorded concrete value.

8. CONCLUSION

We have presented a technique, TAME, to precisely identify schedule-sensitive branches in concurrent programs. TAME combines symbolic trace analysis and concolic execution to precisely determine if a branch can make a different decision in any feasible schedule based on the observed execution trace. We have specifically addressed the various challenges for handling real world programs and proposed a novel distributed trace partition approach to achieve good balance between analysis efficiency and effectiveness. Our evaluation on both popular benchmarks and large complex real applications demonstrates that TAME is effective and scales well to large programs. TAME found 34 schedule-sensitive branches in these programs, which were first reported in this paper, with half of them resulting from concurrency errors.

9. ACKNOWLEDGEMENT

We would like to thank the anonymous ESEC/FSE reviewers for their constructive comments. This research is supported by faculty start-up funds from Texas A&M University and a Google Faculty Research Award to Jeff Huang.

10. REFERENCES

[1] ASM bytecode analysis framework. http://asm.ow2.org.

[2] Derby bug #2861. https://issues.apache.org/jira/browse/DERBY-2861.

[3] Java 7 instruction opcodes. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-7.html.

[4] T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/ .

[5] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jPredictor: a predictive runtime analysis tool for Java. In *International Conference on Software Engineering*, pages 211–230, 2008.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[7] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, 2006.

[8] Mahdi Eslamimehr and Jens Palsberg. Sherlock: Scalable deadlock detection for concurrent programs. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 353–365, 2014.

[9] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2013.

[10] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. Predicting null-pointer dereferences in concurrent programs. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 47:1–47:11, 2012.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[12] Jeff. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 165–174, 2015.

[13] Jeff Huang, Qingzhou Luo, and Grigore Rosu. GPredict: Generic Predictive Concurrency Analysis. In *International Conference on Software Engineering*, 2015.

[14] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

[15] Jeff Huang and Charles Zhang. PECAN: Persuasive Prediction of Concurrency Access Anomalies. In *ACM International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

[16] Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–152, 2013.

[17] Jeff Huang, Jinguo Zhou, and Charles Zhang. Scaling predictive analysis of concurrent programs by removing trace redundancy. *ACM Transactions on Software Engineering and Methodology*, 22(1):8:1–8:21, 2013.

[18] Patrick Lam, Eric Bodden, and Laurie Hendren. The soot framework for Java program analysis: a retrospective, 2011.

[19] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. Steering symbolic execution to less traveled paths. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2013.

[20] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[21] Niloofar Razavi, Franjo Ivancic, Vineet Kahlon, and Aarti Gupta. Concurrent test generation using concolic multi-trace analysis. In *APLAS*, 2012.

[22] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.

[23] Koushik Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[24] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, 2006.

[25] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2005.

[26] Hyunmin Seo and Sunghun Kim. How we get there: A context-guided search strategy in concolic testing. In *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2014.

[27] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 387–400, 2012.

[28] Haruto Tanno, Xiaojing Zhang, Hoshino Takashi, and Koushik Sen. TesMa and CATG: Automated test generation tools for models of enterprise applications, 2015.

[29] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *ACM International Symposium on Software Testing and Analysis*, 2008.

[30] Chao Wang, Sudipta Kundu, Malay K. Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *FM*, 2009.

[31] Chao Wang, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 328–342, 2010.

[32] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.