Assertion Guided Symbolic Execution of Multithreaded Programs

Shengjian Guo  
Virginia Tech  
Blacksburg, VA, USA

Markus Kusano  
Virginia Tech  
Blacksburg, VA, USA

Chao Wang  
Virginia Tech  
Blacksburg, VA, USA

Zijiang Yang  
Western Michigan University  
Kalamazoo, MI, USA

Aarti Gupta  
Princeton University  
Princeton, NJ, USA

ABSTRACT

Symbolic execution is a powerful technique for systematic testing of sequential and multithreaded programs. However, its application is limited by the high cost of covering all feasible intra-thread paths and inter-thread interleavings. We propose a new assertion guided pruning framework that identifies executions guaranteed not to lead to an error and removes them during symbolic execution. By summarizing the reasons why previously explored executions cannot reach an error and using the information to prune redundant executions in the future, we can soundly reduce the search space. We also use static concurrent program slicing and heuristic minimization of symbolic constraints to further reduce the computational overhead. We have implemented our method in the Cloud9 symbolic execution tool and evaluated it on a large set of multithreaded C/C++ programs. Our experiments show that the new method can reduce the overall computational cost significantly.

Categories and Subject Descriptors

F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D.2.4 [Software Engineering]: Software/Program Verification

Keywords

Symbolic execution, test generation, concurrency, partial order reduction, weakest precondition

1. INTRODUCTION

The past decade has seen exciting developments on symbolic execution of both sequential [19, 42, 48, 8] and concurrent programs [41, 38, 14, 5]. However, existing methods are still limited in their capability of mitigating the state space explosion. That is, the number of paths in each thread may be exponential to the number of branch conditions, and the number of thread interleavings may be exponential to the number of concurrent operations. Many techniques have been proposed to address this problem, including the use of function summaries [18], interpolation [34, 23, 61], static symbolic execution [19], and coverage metrics [14]. In this paper, we propose a new and complementary method, which is designed specifically for pruning redundant executions in multithreaded programs where the properties under verification are expressed as assertions.

Our assertion guided symbolic execution framework focuses on identifying and eliminating executions that are guaranteed to be redundant for checking assertions. Assertions can be used to model a wide variety of interesting properties, ranging from logic and numerical errors, to memory safety and concurrency errors, and has been the focus of many software verification projects. When semantic errors of the program are modeled as simple code reachability, i.e., the reachability of a bad state guarded by the assertion condition, we can concentrate on exploring potentially failure-inducing executions as opposed to all feasible executions of the program. This is particularly attractive in the presence of concurrency, since it becomes possible to uniformly handle the exploration of both intra-thread execution paths and inter-thread interleavings leading to a simple but more powerful analysis algorithm.

The overall flow of our new method is illustrated in Figure 1: the shaded block represents our addition and the remainder illustrates the classic symbolic execution procedure for multithreaded programs [41]. Specifically, given a program $P$ and some symbolic input variables, the procedure explores the feasible executions of the program systematically, e.g., in a depth-first search order.

Starting with an initial test $(in, sch)$ consisting of inputs and thread schedule, the method first produces a concrete execution followed by a symbolic execution. Then, it tries to generate a new test by flipping a prior decision at either an interleaving pivot point (i-PP) or a local branch pivot point (b-PP). The new test is denoted by either $(in, sch')$ or $(in', sch)$, depending on whether changes are made to the thread schedule (sch') or data input (in'), respectively. The iterative procedure terminates when no new test can be generated.

State explosion occurs because it has to explore the com-
bined space of inputs and thread schedules where each individual execution may be unique, i.e., it leads to a different program state.

We extend the baseline algorithm by adding a constraint-based pruning block shown in Figure 1, which centers around the idea of summarizing the reasons why the bad state is unreachable via previously explored executions, and leveraging such information to avoid similarly futile executions. Specifically, at each global control location \( n \), we use a predicate summary (PS) constraint to capture the weakest preconditions [13] of the assertion condition along all explored executions starting from \( n \). Therefore, \( \text{PS}[n] \) captures the reason why prior executions are not able to violate the assertion. Whenever symbolic state \( n \) is reached again through another execution path, we check if the new path condition is subsumed by \( \text{PS}[n] \). If so, we can safely backtrack from \( n \) since extending the execution beyond \( n \) would never lead to a bad state.

Our method for pruning redundant executions can be viewed as a way of systematically exploring an abstract search space defined by a set of predicates [4] which, in this case, are extracted from the assertion. Although the concrete search space may be arbitrarily large, the abstract search space can be significantly smaller. In this sense, our method is similar to predicate abstraction [21] in model checking except that the latter requires constructing a priori a finite-state model from the actual software code whereas our method directly works on the software code while leveraging the predicates to eliminate redundant executions.

Our method complements partial order reduction (POR) techniques in that it relies on property-specific information to reduce the state space. But, POR techniques typically do not target particular states. We will show through experiments that our new method can indeed eliminate a different class of redundant executions from those eliminated by state-of-the-art POR techniques, such as dynamic partial order reduction (DPOR) [16]. Toward this end, since DPOR is an elegant but delicate algorithm that can easily be made unsound without taking great care in the implementation [59], a main technical challenge in our work is to make sure our new pruning method does not interfere with DPOR or make it less effective.

Our method differs from prior works by Wachter et al. [51], and Chu and Jaffar [9], which extended the framework of lazy abstraction with interpolants [34] to multithreaded programs. One main difference is that our computation of predicate summaries is significantly more general than existing methods, especially at the thread interleaving pivot points, where we merge summaries from multiple execution paths to form a combined summary. Another main difference is in the integration of property specific pruning with partial order reduction. Both existing methods implemented a variant of the symbolic partial order reduction algorithm by Kahlion et al. [26] whereas we integrate our predicate summary-based pruning method with the more scalable DPOR algorithm.

We have implemented our method in Cloud9 [11], a state-of-the-art symbolic execution tool for multithreaded C/C++ programs. We have implemented an inter-procedural static program slicing algorithm [22], executed prior to symbolic execution, to further reduce the search space. We have also implemented heuristic based minimizations of predicate summary constraints during symbolic execution to reduce the computational overhead. In both cases, the main technical challenge is to ensure the overall algorithm remains sound in the presence of such optimizations. We have conducted experiments on a set of standard multithreaded C/C++ applications. Our results show that the new method can reduce the number of explored executions as well as the overall run time significantly.

To sum up, this paper makes the following contributions:

- We implement our method in a state-of-the-art symbolic execution tool while ensuring it does not interfere with the popular DPOR algorithm or make it less effective.
- We demonstrate through experiments that our new method can indeed achieve a significant performance improvement on public benchmarks.

The remainder of this paper is organized as follows. First, we illustrate our new method through examples in Section 2, then establish the notation and review the baseline symbolic execution algorithm in Section 3. We present our method for summarizing explored executions in Section 4 and pruning redundant executions in Section 5. We present optimization techniques in Section 6 and experimental results in Section 7. We review related work in Section 8 and finally give our conclusions in Section 9.

2. MOTIVATING EXAMPLES

In this section, we illustrate the high-level ideas in our method using examples. Consider the example in Figure 2, which has two threads \( T_1 \) and \( T_2 \), a global variable \( x \), and two local variables \( a \) and \( b \). The initial value of \( x \) is a symbolic input which can be any integer value. We want to check if the assertion fails and, if so, compute a failure-inducing test input.

\[
\text{assert( } a=x \leq b \text{ )}
\]

\[
\begin{align*}
\text{Run 1: } & \text{if}(x>10) \text{ return; } \\
& \text{if}(x<=10) x=10; x=20; a=x; b=x;
\end{align*}
\]

\[
\begin{align*}
\text{Run 2: } & \text{if}(x>10) \text{ return; } \\
& \text{if}(x<=10) a=x; x=10; x=20; b=x;
\end{align*}
\]

\[
\begin{align*}
\text{Run 3: } & \text{if}(x>10) \text{ return; } \\
& \text{if}(x<=10) a=x; x=10; x=20; b=x;
\end{align*}
\]

\[
\begin{align*}
\text{Run 4: } & \text{if}(x>10) \text{ return; } \\
& \text{if}(x<=10) a=x; x=10; x=20; b=x;
\end{align*}
\]

\[
\begin{align*}
\text{Run 5: } & \text{if}(x>10) \text{ return; } \\
& \text{if}(x<=10) a=x; x=10; x=20; b=x;
\end{align*}
\]

\[
\begin{align*}
\text{Run 6: } & \text{if}(x>10) \text{ return; } \\
& \text{if}(x<=10) a=x; x=10; x=20; b=x;
\end{align*}
\]

Figure 2: Our method has to explore one full run and four partial runs, as opposed to all six runs by existing methods.
The program has six distinct executions, each leading to a different final state defined by the values of $a$ and $b$. According to the theory of partial order reduction [16], they belong to six different equivalence classes [32], each having a different final state. However, exploring all six executions is not necessary for the purpose of checking the assertion, since some of these executions share the same reason why they cannot reach the bad state. Our new method can reduce the exploration from six executions to one full execution together with four partial executions, as illustrated by the red dotted lines in Figure 2.

Our method first extracts a set of predicates by computing the weakest preconditions of the assertion condition along the explored executions. These predicates are then combined at the merge points (in the graph) to form a succinct summary that captures the reason why the bad state has not been reached via executions starting from these merge points. During subsequent symbolic execution iterations, our method needs to explore only those executions that have not been covered by these predicates, thereby leading to a sound reduction of the search space.

Now, we explain how our method works on this example:

- Run 1 is the first and only execution fully explored, which goes through nodes $n_1, n_2, n_4, n_7$ in the graph in Figure 2 before executing $b = y$ if ($a < b$). Since it does not violate the assertion, we summarize the reason at $n_3$ and $n_2$, respectively, as follows: $PS[n_3] = (a \leq b)$ and $PS[n_7] = (a \leq x)$. That is, as long as ($a \leq x$) holds at node $n_7$, it would be impossible for the execution to reach the bad state.

- Run 2 goes through nodes $n_1, n_2, n_5$ before reaching $n_7$, where its path condition is $pcon[n_5] = (V \leq 10)$ and symbolic memory is $M = (a = 10, x = 20)$. Since $pcon[n_5] \rightarrow PS[n_7]$ under $M$, meaning the set of reachable states falls inside $PS[n_7]$, continuing the execution from $n_7$ would not lead to a bad state. Therefore, we skip the remainder of this execution.

- Run 3 goes through nodes $n_1, n_2, n_5, n_6$ before reaching $n_9$, where its path condition again falls within $PS[n_6]$. We skip the remainder of this execution and update the summary at $n_8$ and $n_9$ as follows: $PS[n_8] = (a \leq b)$ and $PS[n_9] = wp[n_7] \land wp[n_8] = (a \leq 20) \land (a \leq x)$. By conjoining the weakest preconditions along both interleavings $n_5 \rightarrow n_7$ and $n_5 \rightarrow n_8$, we capture the summary common to both interleavings.

- Run 4 goes through nodes $n_1, n_3, n_5$ before reaching $n_8$, with the new path condition $pcon[n_5] = (V \leq 10)$ and symbolic memory $M = (a = v, x = 10)$. Since $pcon[n_5] \rightarrow PS[n_8]$ under $M$, we skip the remainder of this execution, which would have led to Run 4 and Run 5 if it was allowed to continue.

- Run 5 goes through nodes $n_1, n_3, n_5$ before reaching $n_8$, where the new path condition falls within $PS[n_8]$. Therefore, we skip the remainder of this execution.

At this moment, our method has completed the exploration.

Note that we conjoin weakest preconditions from different interleavings at i-PP nodes such as $n_6$, but union them from different thread-local paths at b-PP nodes (see Section 4). Also note that the amount of reduction achieved by our method depends on the program structure as well as the location of the assertion. For example, if we change $if(x > 10)$ to $if(x < 11)$, our method would have to explore Run 5 instead of skipping it because $pcon[n_5] = (V \leq 11)$ would no longer be subsumed by $PS[n_5] = (V \leq 10)$.

This example demonstrates that our method differs from partial order reduction techniques such as DPOR [16] which could not prune away any of the six interleavings. Furthermore, our method also differs from the stateful state space exploration techniques commonly used in model checking, which record the forward reachable states explicitly during exploration to prevent visiting them again. Such methods would not be effective for the example in Figure 2 either because each of the six executions leads to a distinct state. In contrast, our new method can achieve a significant reduction due to its use of property specific information as guidance. In this sense, our new method is a property directed reduction, whereas the POR techniques are property agnostic.

However, it can be tricky to combine our pruning method with the state-of-the-art DPOR algorithm. The main advantage of DPOR over static POR lies in its dynamic update of backtrack sets, which uses runtime information to compute the dependency relation between shared accesses. Without taking any additional measure, pruning redundant executions may interfere with the dynamic update of backtrack sets in DPOR. Consider run 4 in Figure 2 as an example. If the execution is allowed to complete, when $b=x$ is executed, thread $T_2$ will be added to the backtrack set of node $n_3$. However, if run 4 is terminated prematurely at node $n_4$ due to our predicate summary-based pruning, thread $T_2$ would not be added to the backtrack set of node $n_3$ since $b=x$ has been skipped. As a result, the DPOR algorithm would not explore run 6. Therefore, integrating DPOR with property specific pruning is a challenging task. We present our solution to this problem in Section 5.2.

Our computation of predicate summaries at the thread interleaving merge point $n_5$ in Figure 2 shows that it is different from the prior work by Wächter et al. [51], and Chu and Jaffar [9]. Specifically, we combine the summaries from all outgoing edges by conjoining them, whereas existing methods do not merge interpolants at these i-PP nodes. Furthermore, these existing methods implemented symbolic POR whereas our method is integrated with the more scalable DPOR.

Now, we use the example in Figure 3 to demonstrate that our new method has the potential to achieve an exponential reduction. In this contrived example, the interleaving of instructions in $(a=x, x=10)$ is completely independent from $(b=y, y=10)$ and $(c=z, z=10)$. Exploring all feasible executions results in $2^k$ runs, each of which leads to a different final state. However, based on the abstract search space induced by the assertions, our new method can reduce the exploration of eight runs down to one full run together with three partial runs, as marked by the $*$ symbol in Figure 3. To further generalize the example, a program with $k$ independent code segments would have $2^k$ distinct interleavings, which can be reduced by our method to $(k+1)$ executions.
3. PRELIMINARIES

We establish the notation and review the baseline symbolic execution algorithm for multithreaded programs in this section.

3.1 Multithreaded Programs

For ease of presentation, we consider a simple imperative language with integer variables, assignments, and if-else statements only. We elide the details for handling input and output operations such as pointers, recursion, and system calls in symbolic execution since these are orthogonal issues addressed previously by many symbolic execution tools [8, 11]. A multithreaded program $P$ consists of a set of threads $\{T_1 \ldots T_m\}$, where each thread, $T_i$, is a sequential program. Threads share a set of global variables.

Each thread also has a set of local variables.

Let $st$ be an instruction in a thread with the thread index $tid$. Let event $e = (tid, l, st')$ be an execution instance of $st$, where $l$ and $l'$ are locations in the thread before and after executing the instance of $st$. If the same instruction is executed more than once, e.g., when it is in a loop or a recursive function call, we make copies of $l, st, l'$ to make them unique for each event. Conceptually, this corresponds to unrolling loops and recursive calls. A global control state of the multithreaded program is a tuple $s = \langle l_1, \ldots, l_m \rangle$, where each $l_i$ is a location in $T_i$. We regard a global control state as an abstract state implicitly containing all concrete states that have the same thread locations but potentially different values of the local and global variables.

Without loss of generality, we assume that every assertion of the form $\text{assert}(c)$ is transformed to $\text{if}(c)\text{abort}$. We use a special event $\text{abort}$ to denote faulty program termination and $\text{halt}$ to denote normal program termination. Let $v_l$ denote a local variable, $v_g$ denote a global variable, $\text{cond}$ denote a local condition, and $\text{exp}$ denote an local expression. In addition to $\text{abort}$ and $\text{halt}$, each instruction $st$ in an event may have one of the following types:

- $\alpha$-operation, which is a local assignment $v_l := \text{exp}$;
- $\beta$-operation, which is a local branch $\text{assume}(\text{cond})$;
- $\gamma$-operation, which is a global operation defined as follows:
  - $\gamma$-I is a global write $v_g := \text{exp}$, or read $v_l := v_g$;
  - $\gamma$-II is a thread synchronization operations.

For each $\text{if}(c)\text{-else statement}$, we use $\text{assume}(c)$ to denote the execution of then-branch, and $\text{assume}(!c)$ to denote the execution of else-branch. Without loss of generality, we assume that all if-else conditions use only local variables or local copies of global variables [17]. For thread synchronizations, we focus on mutex locks and condition variables since they are frequently used in mainstream multithreaded programming environments such as C, C++, and Java. Specifically, we consider the following types of $\gamma$-II operations: thread creation, thread join, lock, unlock, signal, and wait. If other thread synchronizations or blocking operations are used they can be modeled similarly as $\gamma$-II events.

During the program execution, $\gamma$-operations are thread interleaving points whereas $\beta$-operations are thread-local branching points. Both contribute to the path interleaving explosion. In contrast, $\alpha$-operations are local and thus invisible to other threads; they do not contribute directly to the path interleaving explosion.

A concrete execution of the multithreaded program is characterized by $\pi = (\pi_h, \pi_s)$, where $\pi_h$ is the thread schedule corresponding to the total order of events $e_1 \ldots e_n$. The corresponding symbolic execution is denoted by $(s, \pi_s)$, where the $s$ indicates the data input is kept symbolic and thus may take any value. Each execution of the program $P$ can be represented by a finite word $(\alpha, \beta, \gamma)^* \text{halt} \text{abort}$. If the execution ends with $\text{halt}$ it is a normal execution. If the execution ends with $\text{abort}$ it is a faulty execution.

3.2 Generalized Interleaving Graph (GIG)

The set of all possible executions of a multithreaded program can be captured by a generalized interleaving graph (GIG), where nodes are global control states and edges are events. The root node corresponds to the initial state. Leaf nodes correspond to normal or faulty ends of the execution. Each internal node may have:

- one outgoing edge corresponding to an $\alpha$-operation;
- two outgoing edges corresponding to a $\beta$-operation; or
- $k$ outgoing edges where $k \geq 2$ is the number of enabled $\gamma$-operations from different threads.

We call a node with more than one outgoing edge a pivot point.

- If the pivot point corresponds to $\beta$-operations we call it a branching pivot point (b-PP).
- If the pivot point corresponds to $\gamma$-operations we call it a thread interleaving pivot point (i-PP).

Figure 4 shows a program and its GIG. For simplicity, we assume $a=x++$ is atomic. The root node $(a_1, b_1)$ corresponds to the starting points of the two threads. The terminal node $(a_3, b_3)$ corresponds to the end of the two threads. Nodes such as $(a_1, b_1)$ are i-PP nodes, where we can execute either thread 1 which leads to $(a_2, b_1)$, or thread 2 which leads to $(a_1, b_2)$. In contrast, nodes such as $(a_1, b_1)$ are b-PP nodes, where we can take either the $\text{assume}(a=0)$ branch, leading to the code segment $T_{a1}$, or the $\text{assume}(a \neq 0)$ branch, leading to the code segment $T_{a2}$.

Note that the GIG does not have loop-back edges since the GIG paths represent unrolled executions. Furthermore, pointers, aliasing, and function calls have been resolved as well during execution. However, a GIG may have branches, which makes it significantly

![Figure 4: A two-threaded program and its generalized interleaving graph (GIG). Black edges represent events from thread $T_1$ and blue edges represent events from thread $T_2$.](image-url)
different from the typical thread interleaving graph used in the partial order reduction literature.

As is typical in symbolic execution algorithms, we focus on only a finite set of executions and assume that each execution has a finite length. Typically, the user of a symbolic execution tool needs to construct a proper testing environment that satisfies the above assumption. In KLEE [8] and Cloud9 [11], for example, the user may achieve this by bounding the size of the symbolic input thereby restricting the execution to a fixed number of paths of finite lengths.

3.3 Symbolic Execution

We present the baseline symbolic execution procedure for multithreaded programs in Algorithm 1 following Sen et al. [41]. The recursive procedure EXPLORE is invoked with the symbolic initial state st. Inside the procedure, we differentiate among three scenarios based on whether s, the current state, is an i-PP node, a b-PP node, or a non-branching node.

If s is an i-PP node where multiple γ-operations are enabled, we recursively explore the next γ event from each thread. If s is a b-PP node where multiple sequential branches are feasible, we recursively explore each branch. If s is a non-branching node, we explore the unique next event. The current execution ends if s is a leaf node (normal_end_state, faulty_end_state) or an infeasible_state, at which point we return from EXPLORE(s) by popping the state s from the stack S.

Algorithm 1 Baseline Symbolic Execution.

Initially: Stack S = {s0}; run EXPLORE(s0) with the symbolic initial state s0.

1: EXPLORE(s)
2: S.push(s)
3: if (s is an i-PP node)
4: while (I ∈ (s.enabled \ s.done))
5: s’ ← NEXTSTATE(s, t);
6: EXPLORE(s’);
7: s.done ← s.done ∪ {t};
8: else if (s is a b-PP node)
9: while (I ∈ (s.branch \ s.done))
10: s’ ← NEXTSTATE(s, t);
11: EXPLORE(s’);
12: s.done ← s.done ∪ {t};
13: else if (s is an internal node)
14: t ← s.next;
15: s’ ← NEXTSTATE(s, t);
16: EXPLORE(s’);
17: S.pop();
18: NEXTSTATE(s, t)
19: let s = (pcon, M, enabled, branch, done);
20: if (t is halt)
21: s’ ← normal_end_state;
22: else if (t is abort)
23: s’ ← faulty_end_state;
24: else if (t is assume(c))
25: if (s.pcon is unsatisfiable under M)
26: s’ ← infeasible_state;
27: else
28: s’ ← (pcon ∧ c, M);
29: else if (t is assignment v := exp)
30: s’ ← (pcon, M[exp/v]);
31: return s’;

Each state s ∈ S is a tuple ⟨pcon, M, enabled, branch, done⟩, where pcon is the path condition for the execution to reach s from s0, M is the symbolic memory map, enabled is the set of γ-events when s is an i-PP node, branch is the set of β-events when s is a b-PP node, and done is the set of α or β events already explored from s by the recursive procedure. Initially, s0 is set to ⟨true, Minit⟩, where true means the state is always reachable and Minit represents the initial content of the memory. The execution of each instruction t is carried out by NEXTSTATE(s, t) as follows:

- If t is halt, the execution ends normally.
- If t is abort, and s.pcon is satisfiable under the current memory map s.M, we have found an error.
- If t is v := exp, we need to update the current memory map M by changing the content of v to exp.
- If t is assume(c), we change the path condition to (pcon ∧ c).

At each pivot point (i-PP or b-PP), we try to flip a decision made previously to compute a new execution. Let (in, sch) denote the current execution. By flipping the decision made previously at an i-PP node, we compute a new execution (in, sch’), where sch’ is a permutation of the original thread schedule. In contrast, by flipping the decision made previously at a b-PP node, we compute a new execution (in’, sch), where in’ is a new data input. Note that in both cases, the newly computed execution will be the same as the original execution up to the flipped pivot point. After the flipping, the rest of the execution will be a free run.

As an example, consider the GIG in Figure 4, where the current execution is represented by the dotted line run-i. Flipping at the b-PP node (as1, bs) would lead to the new execution labeled run-ii, whereas flipping at the i-PP node (as3, bs) would lead to the new execution run-iii.

4. SUMMARIZING THE EXPLORED EXECUTIONS

We first present our method for symbolically summarizing the reason why explored executions cannot reach the bad state. In the next section, we will leverage the information to prune away redundant executions.

Our method for summarizing the explored executions is based on the weakest precondition computation [13]. We differentiate the following two scenarios, depending on whether the execution encounters the assert statement or not.

- For each execution that encounters assert(c) and satisfies the condition c, we compute the weakest precondition of the predicate c along this execution.
- For each execution that does not encounter the assert statement at all, we compute the weakest precondition of the predicate true along this execution.

Since the weakest precondition is a form of Craig’s interpolant [34], it provides a succinct explanation as to why the explored execution cannot reach the bad state guarded by ¬c.

**Definition 1.** The weakest precondition of the predicate φ with respect to a sequence of instructions is defined as follows:

- For t: v := exp, WP(t, φ) = φ/exp/v;
- For t: assume(c), WP(t, φ) = φ ∧ c; and
- For sequence t1; t2; WP(t1; t2, φ) = WP(t1, WP(t2, φ)).

In the above definition, φ/exp/v denotes the substitution of variable v in φ with exp. As an example, consider the execution path in the following table, which consists of three branch conditions and three assignments. Column 1 shows the control locations along the current path. Column 2 shows the sequence of instructions executed. Column 3 shows the weakest preconditions computed backwardly starting at l6. Column 4 shows the rules applied during the computation.
4.1 Predicate Summaries at b-PP Nodes

Assume that the baseline symbolic execution procedure traverses the GIG in a depth-first search (DFS) order, meaning that it backtracks s, a branching pivot point (b-PP), only after exploring both outgoing edges \( s \xrightarrow{\text{assume}(c)} s' \) and \( s \xrightarrow{\neg \text{assume}(c)} s'' \). This also includes the entire execution trees starting from these two edges. Let \( wp[s'] \) and \( wp[s''] \) be the weakest preconditions computed from the two out-going executions, respectively.

Following the classic definition of weakest precondition [13], we compute them at the b-PP node \( s \) as follows:

\[
wp[s] := \big( c \land wp[s'] \big) \lor \big( \neg c \land wp[s''] \big).
\]

Then, we use \( wp[s] \) computed from these outgoing edges to update the global predicate summary.

The predicate summary, \( PS[s] \), defined for each global control state \( s \), is the union of all weakest preconditions along the outgoing edges. Recall that each node \( s \) may be visited by \( \text{EXPLORE} \) multiple times, presumably from different execution paths (from \( s_0 \) to \( s \)). Therefore, we maintain a global map \( PS \) and update each predicate summary entry \( PS[s] \) incrementally. Initially \( PS[s] = \text{false} \) for every GIG node \( s \). Then, we merge the newly computed \( wp[s] \) to \( PS[s] \) every time \( \text{EXPLORE} \) backtracks from \( s \).

The detailed method for updating the predicate summary is highlighted in blue in Algorithm 2, which follows the overall flow of Algorithm 1, except for the following two additions:

- We compute \( wp[s] \) before the procedure backtracks from state \( s \). At this moment, \( wp[s] \) captures the set of all explored executions from \( s \) as a continuation of the current execution.
- We update the summary as follows: \( PS[s] = PS[s] \lor wp[s] \).

Here, \( PS[s] \) captures the set of execution trees as a continuation of all explored executions from \( s_0 \) to \( s \), including \( wp[s] \), which represents the newly explored execution tree.

4.2 Predicate Summaries at i-PP Nodes

In contrast to the straightforward computation of weakest precondition at the sequential merge point, the situation at the interleaving merge point is trickier. In fact, to the best of our knowledge, there does not exist a definition of weakest precondition in the literature for thread interleaving points.

A naive extension of Dijkstra’s original definition would be insufficient since it leads to the explicit enumeration of all possible interleavings. For example, assume that an i-PP node has two outgoing edges \( s \xrightarrow{\text{tau}_i} s' \) and \( s \xrightarrow{\text{tau}_j} s'' \), one may attempt to define the weakest precondition at node \( s \) as follows:

\[
\left( (\gamma_1 \prec \text{bb} \gamma_2) \land wp[s'] \right) \lor \left( (\gamma_2 \prec \text{bb} \gamma_1) \land wp[s''] \right),
\]

where \( (\gamma_1 \prec \text{bb} \gamma_2) \) means that we choose to execute \( \gamma_1 \) before \( \gamma_2 \), \( (\gamma_2 \prec \text{bb} \gamma_1) \) means that we choose to execute \( \gamma_2 \) before \( \gamma_1 \), and \( wp[s'] \) and \( wp[s''] \) are the weakest preconditions along the two interleavings, respectively.

Although the above definition serves the purpose of summarizing the weakest preconditions along all explored executions from \( s \), it has a drawback: the size of \( wp[s] \) computed in this way can quickly explode when there are a large number of threads. Recall that in a multithreaded program the number of outgoing edges at an i-PP node equals the number of enabled threads and the number of interleavings of \( k \) concurrent threads can be \( k! \) in the worst case. However, for the purpose of pruning redundant executions, the weakest precondition computation does not have to be precise to be effective. To mitigate the aforementioned interleaving explosion problem, we resort to the following definition, which can be viewed as an under-approximation of the naive definition:

\[
wp[s] := \bigwedge_{1 \leq i \leq k} wp[s_i],
\]

where each \( wp[s_i] \) is the weakest precondition computed along one of the \( k \) outgoing edges of the form \( s \xrightarrow{\text{tau}_i} s' \), such that \( 1 \leq i \leq k \). Consider Figure 2 as an example. We compute the weakest precondition at node \( n_5 \) by conjoining weakest preconditions at the two successor nodes \( n_7 \) and \( n_8 \). That is, \( wp[n_5] = wp[n_7] \land wp[n_8] \) when \( (a \leq 20) \land (a \leq x) \).

Algorithm 2: Assertion Guided Symbolic Execution.

Initially: summary \( PS[n] = \text{false} \) for all node \( n \); stack \( S = \{s_0\} \); run \( \text{EXPLORE}(s_0) \) with initial state \( s_0 \).

1. \( \text{EXPLORE}(s) \)
2. \( S \) push(s);
3. if (s is an i-PP node)
4. \( wp[s] \triangleright \text{true} \);
5. while (3t \in (s.enabled \land s.done))
6. \( s' \leftarrow \text{NEXTSTATE}(s, t) \);
7. \( \text{EXPLORE}(s') \);
8. \( wp[s] \leftarrow wp[s] \land \text{WP} \); \( s, t, s' \);
9. \( s.done \leftarrow s.done \lor \{t\} \);
10. else if (s is a b-PP node)
11. \( wp[s] \leftarrow \text{false} \);
12. while (3t \in (s.branch \land s.done))
13. \( s' \leftarrow \text{NEXTSTATE}(s, t) \);
14. \( \text{EXPLORE}(s') \);
15. \( wp[s] \leftarrow wp[s] \lor \text{WP} \); \( s, t, s' \);
16. \( s.done \leftarrow s.done \lor \{t\} \);
17. else if (s is an internal node)
18. \( t \leftarrow s.next \);
19. \( s' \leftarrow \text{NEXTSTATE}(s, t) \);
20. \( \text{EXPLORE}(s') \);
21. \( wp[s] \leftarrow \text{WP} \); \( s, t, s' \);
22. else if \( s \) is end state
23. \( wp[s] \leftarrow \text{true} \);
24. \( PS[s] := PS[s] \lor wp[s] \);
25. S.pop();
26. \( \text{COMPUTEWP}(s, t, s') \)
27. if (t is \( \text{assume}(c) \))
28. \( \text{return} \ wp[s'] \land c; \)
29. else if (t is assignment \( v := \text{exp} \))
30. \( \text{return substitute}(wp[s'], v, \text{exp}); \)
31. else
32. \( \text{return} \ wp[s'] \);
33. \( \text{NEXTSTATE}(s, t) \)
34. let \( s \) be tuple (pc, M, enabled, branch, done);
35. if (t is \( \text{halt} \))
36. \( s' \leftarrow \text{normal_end_state}; \)
37. else if (t is \( \text{abort} \))
38. \( s' \leftarrow \text{faulty_end_state}; \)
39. else if (t is \( \text{assume}(c) \))
40. if (s.pc is unsatisfiable under M)
41. \( s' \leftarrow \text{infeasible_state}; \)
42. else if (pc \rightarrow \text{PS}[s])
43. \( s' \leftarrow \text{early_termination_state}; \)
44. else
45. \( s' \leftarrow (\text{pc} \land c, M); \)
46. else if (t is assignment \( v := \text{exp} \))
47. \( s' \leftarrow (\text{pc}, M[\text{exp}/v]); \)
48. \( \text{return} \ s' \).

For pruning redundant executions, conjunction of weakest preconditions from different interleavings at i-PP nodes is a sound approximation. Although it may not capture all the explored executions and thus fail to prune certain redundant executions, all the pruned executions are guaranteed to be redundant.

5. PRUNING REDUNDANT EXECUTIONS

We present our method for leveraging the predicate summaries to prune away redundant executions in this section.
5.1 Assertion Guided Pruning

To decide if we can skip executions starting from a global control state $s$, where $s$ has been visited by EXPLORE through some executions from $s_0$ to $s$ but is reached again through a new execution, we check whether the current path condition $s.pcon$ is subsumed by $PS[s]$ under the current memory map $s.M$. Intuitively, the path condition $s.pcon$ represents the set of states reachable along the current execution from $s_0$ to $s$, whereas $PS[s]$ represents the set of states from which it is impossible to reach the bad state.

Within the NextState procedure in Algorithm 2, we check for the pruning condition as follow:

- If $s.pcon \rightarrow PS[s]$ holds under $s.M$, extending the current execution beyond $s$ would not lead to a bad state. Therefore, we backtrack immediately by setting $s'$ as an early termination state.
- Otherwise, there may exist an extension of the current execution beyond $s$ to reach the bad state. In this case, we need to continue the forward symbolic execution as usual.

The validity of $s.pcon \rightarrow PS[s]$ can be decided by checking the satisfiability of $(s.pcon \land \neg PS[s])$ using an SMT solver. That is, $s.pcon \rightarrow PS[s]$ holds if and only if $(s.pcon \land \neg PS[s])$ is unsatisfiable.

Our new pruning method is complementary to partial order reduction techniques. POR is a generic reduction that relies solely on commutativity between concurrent operations. Therefore, two executions are considered equivalent as long as they result in the same program state. Our new method, in contrast, uses assertions to guide the pruning. Therefore, even executions that result in different program states may still be regarded as equivalent.

Consider the GIG in Figure 4, which has 54 feasible executions. To make the presentation simple, we have assumed that $x++$ is atomic in this example. However, note that $a_1$ : $s' \rightarrow x++$ and $b_1$ : $b' \rightarrow x++$ do not commute, because from a state where $x=0$, for instance, executing $a_1;b_1$ leads to $a=0,b=1,x=2$, but executing $b_1;a_1$ leads to $a=1,b=0,x=2$. As shown in Table 1, without applying any reduction technique, the program has a total of 54 distinct runs. Partial order reduction (POR) alone can reduce the 54 runs down to 34 runs. Our new predicate summary-based pruning method alone can reduce the 54 runs down to the 18 runs. Finally, applying both our method and POR can reduce the 54 runs down to 13 runs.

However, since $n_{3, pcon} \rightarrow PS[n_5]$ along run 4, our pruning method would force EXPLORE to backtrack from $n_5$, thereby skipping the remainder of run 4 and run 5. Here, the technical challenge is how to properly update the backtrack set at node $n_3$ before EXPLORE backtracks from $n_5$.

Fortunately, similar problems were encountered during the development of stateful DPOR algorithms [59]. In this work, we follow the solution by Yang et al. [59]. We maintain two global tables, $RVar[s]$ and $WVar[s]$, for each global control state $s$. The $RVar$ table stores the set of global variables that have been read by some thread during previously explored executions starting from $s$. Similarly, the $WVar$ table stores the set of global variables that have been written to by some thread during previously explored executions starting from $s$. These two tables are updated at the same time the global PS table is updated.

For the example in Figure 2, after exploring run 1, run 2, and run 3, we would have $WVar[n_3] = \{(x, T_1)\}$ representing that $x=20$ has previously been executed by thread $T_1$ at some point after $n_3$. Similarly, we have $RVar[n_3] = \{(x, T_2)\}$ representing that $b=x$ has previously been executed by thread $T_2$ at some point after $n_5$.

Whenever EXPLORE decides to skip the execution tree from a node $s$, we can leverage the information stored in $WVar[s]$ and $RVar[s]$ to properly update the backtrack sets for DPOR. For example, the original DPOR algorithm waits until assignment $b=x$ is executed by thread $T_2$ before it can update the backtrack set of $n_3$. Now, using the entry $(x, T_2) \in RVar[n_3]$, it can put thread $T_2$ into the backtrack set of $n_3$, as if $b=x$ has been executed by thread $T_2$ at some point after $n_5$.

The correctness of this solution follows Yang et al. [59] in the context of stateful DPOR, which ensure that DPOR remains sound in the presence of assertion guided pruning. For more information on the dynamic update of backtrack sets, please refer to the original description of DPOR [16].

5.3 Proof of Correctness

Now, we state and prove the correctness of our overall algorithm. Let $SE_{orig}$ be the baseline symbolic execution procedure described in Algorithm 1, and $SE_{new}$ be our new symbolic execution procedure with predicate summary-based pruning, as described in Algorithm 2. We say that $SE_{new}$ is a sound reduction of $SE_{orig}$ if it always reaches the same set of error states as $SE_{orig}$.

**Theorem 1.** Given a program $P$ and an error location $E$. Our new symbolic execution procedure $SE_{new}$ reaches $E$ if and only if the original symbolic execution procedure $SE_{orig}$ reaches $E$.

Proof: We divide the proof into two steps. First, we prove that if $SE_{new}$ reaches $E$, then $SE_{orig}$ also reaches $E$. This is straightforward because $SE_{new}$ explores a subset of the execution paths explored by $SE_{orig}$, as shown by a comparison of the two versions of NextState in Algorithms 1 and 2.

Second, we prove that if $SE_{orig}$ reaches $E$, then $SE_{new}$ reaches $E$. We do this by contradiction. Assume $SE_{orig}$ can reach $E$ along $\pi$ but $SE_{new}$ cannot. Since Lines 42–43 in Algorithm 2 are the only places where $SE_{new}$ can skip a path, there must exist an event $\langle s, t, s' \rangle$ in $\pi$ such that $s.pcon \rightarrow PS[s]$ holds under $s.M$.

- Since path $\pi$ is feasible, the subpath of $\pi$ from $s'$ to $E$ must also be feasible. To skip $\pi$ in $SE_{new}$, the subpath must have been explored and then summarized in $PS[s']$, presumably when $SE_{new}$ first explored the subpath.
- But if $PS[s']$ already includes this common subpath from $s'$ to $E$, by definition, $SE_{new}$ must have reached the error block $E$. This contradicts our assumption that the new symbolic execution procedure $SE_{new}$ cannot reach the error block $E$.

Therefore, our assumption is incorrect. The theorem holds.
6. OPTIMIZATIONS

In our new method, the size of the summary table as well as the size of the logical constraint in each entry may become a performance bottleneck. Since large logic formulas are expensive to compute and store, we would like to reduce the associated computational cost without affecting soundness of the overall procedure. Toward this end, we propose two optimizations.

6.1 Leveraging Static Program Slicing

Our first optimization is to combine our assertion guided pruning with static program slicing to achieve a more significant state space reduction. Given an assertion statement \( st \), we define the slice of \( st \) as the set of all statements in the program that may affect the result of \( st \). The slice is computed based on two dependency relations: the control dependency relation and the data dependency relation. Intuitively, a statement \( st' \) is a control dependency of a statement \( st \) if the execution of \( st' \) determines whether \( st \) can be executed. Whereas a statement \( st'' \) is a data dependency of \( st \) if the execution of \( st'' \) may affect the data used in \( st \).

Consider the example in Figure 5. The write to \( x \) at Line 5 has a control dependency at Line 4, and a data dependency at Line 3. The slice of Line 5 is defined as the transitive closure of its control and data dependencies, which consists of Lines 3–5. In contrast, the branching statement at Line 1 and the write to \( y \) at Line 2 are irrelevant since their execution will not affect the value written to \( x \) at Line 5 nor the reachability of Line 5. Therefore, for checking the assertion at Line 6, which is related to the value of \( x \) at Line 5, we can simply ignore Lines 1–2. In other words, the slice of Line 5 is a control of Line 5, whereas the slice of Line 6 is a data of Line 5 if the execution of Line 6 may affect the data used in Line 5.

![Figure 5: Example for static program slicing.](image)

Consider the example in Figure 5. The write to \( x \) at Line 5 has a control dependency at Line 4, and a data dependency at Line 3. The slice of Line 5 is defined as the transitive closure of its control and data dependencies, which consists of Lines 3–5. In contrast, the branching statement at Line 1 and the write to \( y \) at Line 2 are irrelevant since their execution will not affect the value written to \( x \) at Line 5 nor the reachability of Line 5. Therefore, for checking the assertion at Line 6, which is related to the value of \( x \) at Line 5, we can simply ignore Lines 1–2.

![Figure 6: Using Type A and B nodes outside the slice.](image)

The correctness of this approach directly follows from the definition of slicing. For both Type A and Type B nodes outside the program slice, which outgoing edge to execute does not affect the reachability of the bad state. Due to the relative efficiency of static slicing, the overhead of computing the slice is small compared to the subsequent symbolic execution. However, we will show through experiments that, by leveraging static slicing, we can significantly decrease the number of executions to be explored, thus decreasing the complexity of the overall analysis.

6.2 Approximating the Summary Constraints

Following Theorem 1, we can prove that in general, any kind of underapproximation of \( PS[s] \) may be used in Algorithm 2 to replace \( PS[s] \), while maintaining the soundness of our pruning method. Our optimization is to heuristically reduce the computational cost associated with predicate summaries. Toward this end, we propose the following two underapproximations.

First, we use a global hash table with a fixed number \( N \) of entries to limit the storage cost for \( PS \); that is, two global control locations \( s \) and \( s' \) may be hashed to the same entry. Whenever this happens, instead of storing both in a linked list, we drop one of them. That is, when \( key(s) = key(s') \), we heuristically remove one entry, effectively setting the corresponding predicate summary false.

Second, we use a fixed threshold to bound the size of each individual logical constraint for \( PS[s] \). In other words, when the predicate summary becomes too large, we will stop adding new weakest-preconditions to it, thereby dropping all subsequently explored subpaths. That is,

\[
\text{if } (\text{size}(PS[s]) < \text{bd}) \quad \text{PS}[s] := PS[s] \lor \text{wp}[s] .
\]

This is again an underapproximation of \( PS[s] \).

A main advantage of this on-demand constraint minimization framework is that it allows various forms of underapproximations to be plugged into it without affecting the soundness proof of the overall algorithm. With underapproximations, it is possible that we may no longer be able to prune away all redundant executions, however, we can guarantee that all pruned executions are truly redundant. In particular, the baseline symbolic execution in Algorithm 1 (no pruning) can be viewed as an extreme form of underapproximation, where \( PS[s] \) is underapproximated to false for all global control locations.

7. EXPERIMENTS

We have implemented our method in Cloud9 [11], which in turn builds upon the LLVM compiler [2] and the KLEE symbolic virtual machine [8]. Note that KLEE does not by itself support multithreading, and although Cloud9 has extended KLEE to support a limited number of POSIX thread routines, it does not attempt to cover all feasible thread interleavings. Indeed, Cloud9 allows for context switches only before certain POSIX thread synchronizations but not before shared variable reads/writes. Furthermore, Cloud9 does not support partial order reduction. Instead, it forks a new execution every time a POSIX synchronization is encountered, which can cause the number of executions to explode quickly.

We have extended Cloud9 to implement the baseline symbolic execution in Algorithm 1, which systematically explores both intra-thread paths and thread interleavings. Then, we implemented the...
We have conducted experiments on two sets of benchmarks. The first set consists of multithreaded C programs from the 2014 Software Verification Competition (SV-COMP) benchmark [47] and with a combined total of 40,291 lines of code. Each benchmark program is first transformed into LLVM bitcode using Clang/LLVM, before given to the symbolic execution tool with a set of user annotated variables as symbolic input.

Table 2 summarizes the results of our experimental evaluation. Columns 1–3 show the name, lines of code, and the number of threads for each program. Columns 4–9 compare the performance of three different methods in terms of the number of explored runs and the total run time in seconds. Cloud9 denotes the baseline symbolic execution algorithm in Algorithm 1, +DPOR denotes the baseline algorithm with dynamic partial order reduction, and +DPOR + AG denotes our new method, which augments the baseline algorithm with DPOR and assertion guided pruning. The runtime of +DPOR + AG includes the time to compute the slice. For all tests, we used a maximum time of 30 minutes.

In the remainder of this section, we analyze the experimental results in more detail, to answer the following research questions:

1. How effective is our proposed pruning technique? Is it more effective than DPOR alone?
2. How scalable is our technique? Is it practical in handling realistic C/C++ programs?

First, we show the comparison of Cloud9 and +DPOR + AG in two scatter plots in Figure 7, where the x-axis in each scatter plot represents the number of runs (or time) of the baseline algorithm (Cloud9), and the y-axis represents the number of runs (or time) of our method (+DPOR + AG). Each benchmark program is represented by a dot in the scatter plots; dots below the diagonal lines are winning cases for our method. The results show that our new method can significantly reduce the number of runs explored by symbolic execution as well as the overall execution time. In many cases, the baseline algorithm timed out after 30 minutes while our new method finished in a few seconds.

Next, we show the comparison of +DPOR and +DPOR + AG in the scatter plots in Figure 8. Our goal is to quantify how much of the performance improvement comes from our new assertion guided pruning as opposed to DPOR. Again, dots below the diagonal lines are winning cases for our method (+DPOR + AG) over DPOR. For most of the benchmark programs, our new method demonstrated a significant performance improvement over DPOR. But for some benchmark programs, +DPOR + AG was slightly slower than +DPOR despite that it executed the same, or a smaller, number of runs. This is due to the additional overhead of running the supplementary static slicing algorithm, as well as predicate symbolically-based pruning, which did not provide sufficient performance boost to offset their overhead.

However, it is worth noting that, where our combined optimization of slicing and pruning is able to bring a performance improvement, it often leads to a drastic reduction in the execution time compared to DPOR alone. For example, in nedmalloc (Table 2), our new method was able to identify that the property does not depend on any shared variables. In such cases, it can safely skip exploring the entire interleaved state space and finish in just one run.

We also evaluated the growth trends of the three methods when the complexity of the benchmark program increase. Figure 9 shows the results of comparing the three methods on a parameterized program named reorder2false. In these two figures, the x-axis repre-
sents the number of threads created in the parameterized program, and the y-axis represents, in logarithmic scale, the number of runs explored and the execution time in seconds. As shown by these two figures, the computational overhead of all three methods increases as the complexity of the program increases. However, our new method increases at a significantly reduced rate compared to the two existing methods.

8. RELATED WORK

As we have mentioned earlier, for sequential programs, there is a large body of work on mitigating path explosion in symbolic execution using function summaries [18], may-must abstraction [20], demand-driven refinement [31], state matching [50], state merging [30], and structural coverage [37]. McMillan proposed lazy abstraction with interpolants [33, 34], which has been shown to be effective in model checking sequential software [6]. Jaffar et al. [10] used a similar method in the context of constraint programming to compute resource-constrained shortest paths and worst-case execution time. However, a direct extension of such methods to multithreaded programs would be inefficient since they lead to the naive exploration of all thread interleavings.

Wachter et al. [51] extended McMillan’s lazy abstraction with interpolants [34] to multithreaded programs while combining it with a symbolic implementation of the monotonic partial order reduction algorithm [26, 58]. The idea is to apply interpolant-based reduction to each interleaved execution while applying symbolic POR to reduce the number of interleavings. Chu and Jaffar [9] proposed a similar method, where they improved the symbolic POR by considering not only the standard independence relation but also a new semi-commutativity relation. However, these existing methods [51, 9] differ from our method significantly.

First, we merge predicate summaries at interleaving pivot points whereas the existing methods [51, 9] do not. Second, we leverage static program slicing and heuristic minimization of summary constraints during symbolic execution to further reduce the search space. Finally, our pruning method is designed to work seamlessly with the more scalable DPOR algorithm [16] whereas the existing methods implemented symbolic POR. Neither of these previous methods demonstrated handling C/C++ code with more than a thousand lines of code as in our work.

Kusano and Wang [29] introduced a notion of predicate dependence in the context of dynamic partial order reduction. Wang et al. [57, 52] proposed similar property-driven pruning methods for dynamic model checking. However, all these prior methods were geared toward stateless model checking, which can be viewed as a form of systematic testing with fixed data input, as opposed to symbolic data inputs. Furthermore, these methods relied on control and data dependency relations as opposed to symbolic constraints generated from weakest precondition computation, and therefore were unable to merge non-failing executions reaching different final states. In this sense, our new method is a more general and more accurate version of the prior works. Furthermore, it is orthogonal and complementary to the symmetry-reduction method proposed by Yang et al. [60].

Our sound method for pruning executions differs significantly from various heuristic reduction techniques in concurrency testing that do not guarantee the soundness. For example, Farzan et al. [14] and Razavi et al. [38] proposed heuristic methods for quickly exploring certain subsets of thread interleaving scenarios in symbolic execution of concurrent programs. This type of selective interleaving exploration techniques were also used by Wang et al. [56] to quickly cover certain pairs of dependent operations captured by a history aware predecessor set. Further along this line, there are predictive bug detection methods based on the use of SMT solvers [54, 28, 55, 46, 24, 25, 40, 44, 43, 53], which differ from our method in that they explore only the thread interleavings under fixed program inputs.

The GREEN tool by Visser et al. [49] provides a wrapper around constraint satisfiability solvers to check if the results are already available from prior invocations, and reuse the results if available. As such, they can achieve significant reuse among multiple calls to the same solvers during the symbolic execution of different paths. GREEN achieves this by distilling constraints into their essential parts and then representing them in a canonical form. The reuse achieved by GREEN is at a much lower level, and therefore is complementary to our new pruning method.

Finally, we assume sequential consistency, although our method can be integrated with dynamic partial order reduction methods for relaxed memory models [62, 1]; we leave this for future work.

9. CONCLUSIONS

We have presented a predicate summary-based pruning method for improving symbolic execution of multithreaded program. Our method is designed to work with the popular DPOR algorithm, and has the potential of achieving exponential reduction. We have implemented the method in Cloud9 and demonstrated its effectiveness through experiments on multithreaded C/C++ benchmarks. For future work, we plan to conduct more experiments to identify the sweet spots in using heuristic minimizations of summary constraints to exploit the trade-off between increasing the pruning power and decreasing the computational overhead.

10. ACKNOWLEDGMENTS

This work was primarily supported by the NSF under grants CCF-1149454, CCF-1405697, and CCF-1500024. Partial support was provided by the ONR under grant N00014-13-1-0527. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.
11. REFERENCES


