Assertion Guided Abstraction: A Cooperative Optimization for Dynamic Partial Order Reduction

Markus Kusano
ECE Department
Virginia Tech
Blacksburg, VA 24061, USA
mukusano@vt.edu

Chao Wang
ECE Department
Virginia Tech
Blacksburg, VA 24061, USA
chaowang@vt.edu

ABSTRACT

We propose a new method for reducing the interleaving space during stateless model checking of multithreaded C/C++ programs. The problem is challenging because of the exponential growth of possible interleavings between threads. We have developed a new method, called *assertion guided abstraction*, which leverages both static and dynamic program analyses in a cooperative framework to reduce the interleaving space. Unlike existing methods that consider all interleavings of all conflicting memory accesses in a program, our new method relies on a new notion of *predicate dependence* based on which we can soundly abstract the interleaving space to only those conflicting memory accesses that may cause assertion violations and/or deadlocks. Our experimental evaluation of assertion guided abstraction on open source benchmarks shows that it is capable of achieving a significant reduction, thereby allowing for the verification of programs that were previously too complex for existing algorithms to handle.

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

General Terms

Algorithm, Verification, Reliability

Keywords

Stateless model checking, partial order reduction, predicate dependence, assertion guided abstraction, cooperative analysis.

1. INTRODUCTION

Analyzing the behavior of a multithreaded program remains a difficult task despite the large body of existing work on both static and dynamic program analysis techniques. The main reason is that the number of thread interleavings is often exponential in the program size, which means that the naive approach of explicitly checking all possible interleavings is practically infeasible. Due to this well-known *interleaving explosion* problem, existing methods based on static analysis often lack *accuracy* as a result of having to track all possible executions of the program simultaneously. In contrast, dynamic analysis methods can be made significantly more accurate since they only have to focus on a single execution trace at a time. However, without a global view of the program behavior, dynamic analysis methods often lack *foresight*. For example, a dynamic analysis may have difficulty computing even simple facts of the program such as control and data dependencies.

We present in this paper a new cooperative analysis framework for multithreaded programs to allow static and dynamic analysis methods to share information between each other with the goal of increasing the accuracy and speed of the analysis as compared to using each method alone. Specifically, we show that the new static–dynamic analysis framework can be leveraged to efficiently check embedded assertions in a multithreaded program. In this particular application, the static analysis is a control and data dependency analysis, and the dynamic analysis is a stateless model checking procedure augmented with partial order reduction. We shall demonstrate through experiments that the use of both static and dynamic analyses in our cooperative framework can significantly outperform each individual method.

From a static analysis standpoint, assertions in a multithreaded program can be checked with a concurrent data flow analysis. However, carrying out a precise whole-program static analysis that is capable of resolving all embedded assertions is challenging in practice, due to complex language constructs such as loops, recursive function calls, and heap allocated data structures. From a dynamic analysis standpoint, assertions can be checked by using a stateless model checker [15] that systematically executes the program under all possible thread schedules. However, in the presence of *interleaving explosion*, dynamic analysis alone is not sufficient for solving the problem. Our new cooperative static–dynamic approach, in contrast, reduces the verification problem into two significantly simpler subproblems: first statically computing the approximate dependence between statements and then dynamically pruning away the redundant interleavings based on the precomputed dependence.

Toward this end, we introduce the new notion of *predicate dependence* over concurrent operations. Dependence is always at the heart of static and dynamic analysis methods for concurrent programs. For example, dynamic partial order reduction (DPOR [13]) relies on *conflict dependence*. Two operations are conflict dependent if they are from different threads, access the same memory location, and at least one of them is a write operation. DPOR groups execution traces into various equivalence classes and then picks a representative from each equivalence class to check. According to the trace theory by Mazurkiewicz [26], which is the foundation of partial order reduction methods, two traces are equivalent if they can be transformed into each other by repeatedly swapping the *adjacent, independent* transitions. For example, a read of shared variable `x` in `if(x)` and a write in `x:=1` would be considered

ASE'14, September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642998 .

```
#define NUM_THREADS 12
#define SIZE  128
#define MAX   4
int table[SIZE];
int *thread_routine(int *arg) {
  int tid = ((int*)arg);
  int m = 0, w, h;
  while(1) {
    if (m < MAX){
      w = (++m) * 11 + tid;
    }else {
      thread_exit(0);
    }
    h = (w * 7) % SIZE;
    if (h < 0) {
      assert(0);
    }
    while (cas(table, h, 0, w) == 0) {
      h = (h+1) % SIZE;
    }
  }
}
int main() {
  for (int i = 0; i < NUM_THREADS; ++i)
    thread_create(thread_routine(i));
  ...
}
```

Figure 1. Example from SV-COMP 2014 (`IndexerSafe`) where multiple threads share a hash table. The assertion checks if a thread reads past the array bound. `cas` is an atomic compare-and-swap which modifies the state of the table at the passed index.

as conflict-dependent, whereas `if(x)` and `y:=1` would not since they access different memory locations.

However, the definition of conflict dependence is overly restrictive in many cases and does not allow redundant interleavings to be pruned away. For example, the two write operations in `x:=10` and `x:=10` are conflict-dependent and yet their relative execution order is immaterial for property verification. One can imagine extending conflict dependence as follows: two conflict-dependent operations are said to be *view-dependent* if the two different execution orders of them lead to different program states. In other words, `x:=10` and `x:=10` would not be view-dependent but `x:=10` and `x:=20` would be. However, even view dependence would not be able to prune away many redundant interleavings.

The most general extension along this direction is *predicate dependence*. For example, assume that the only place where the values written to `x` are used subsequently in the program is to control the branching condition in `if(x>5)`. In this case, it is actually immaterial whether we execute `x:=10` before or after `x:=20`, because both (10>5) and (20>5) are true. For the purpose of checking reachability properties in a multithreaded program, all we care about is whether the relative execution of two operations affects the reachability of a bad state, e.g., a state where an assertion fails.

Consider the example in Figure 1, which is a variant of the running example used to illustrate the DPOR algorithm in [13]. The program has a set of threads concurrently accessing data items in a shared hash table. The assertion in this program checks if the generated hash table key in each thread is out-of-bounds of the array. When the number of threads is below 12, all possible interleavings of the programs belong to the same equivalence class, meaning that only one representative needs to be checked. However, when the number of threads reaches or goes beyond 12, according to the classic partial order reduction methods, the number of equivalence classes goes up exponentially, which quickly makes existing dynamic analysis methods intractable.

Figure 2 shows the performance of the DPOR algorithm as well as our new method (Pred-DPOR). The x-axis is the number of threads in the test program, and the y-axis is the execution time. Although DPOR performs well when the number of threads is below 12, it suffers from the interleaving explosion when the number

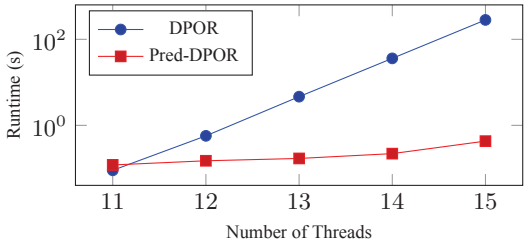

Figure 2. Comparing baseline DPOR with our new Pred-DPOR method on `IndexerSafe` with different number of threads.

of threads is above 12. However, our observation is that, for the purpose of checking assertions in the program, there is no need to explore the exponentially many thread interleavings. Since the hash key, `h`, relies on `w` and `m`, none of which are interfered by other threads, the assertion cannot be violated due to thread scheduling. Where a traditional dynamic analysis method would have to check all possible thread interleavings of the program, with the help of a conservative static analysis, we can tell after *one* run that this program never violates the assertions due to thread scheduling.

We have implemented our new methods by leveraging the LLVM platform to implement the new static analysis and a modified version of *Inspect* [42] to implement the new dynamic analysis. Figure 3 shows the overall flow of our method, which takes a multithreaded C/C++ program as input and determines if there are assertion violations or deadlocks. First, we parse the input program and instrument it to add logging/control capabilities for dynamic analysis. Then, we perform a conservative static analysis in subprocedure named *Dependence Calculate* to compute the predicate dependency relations between potentially concurrent operations. In the subsequent dynamic analysis, which extends the *DPOR* algorithm, we leverage the precomputed dependency relations to prune away redundant thread interleavings.

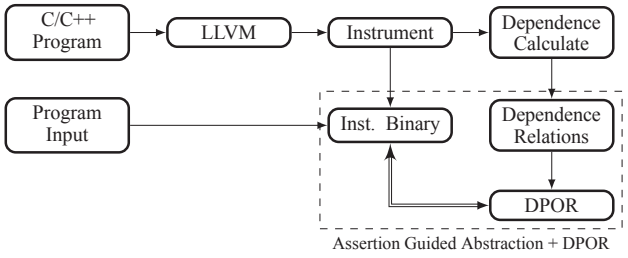

Figure 3. Overall flow of our new method.

We have evaluated our new method on a set of open source benchmarks and benchmarks from SV-COMP 2014. Our experiments show that the cooperative analysis framework can greatly reduce the search efforts by focusing on the subset of the conflict-dependent operations that actually affect the validity of the properties at hand. In fact, our results show that, when compared to the default DPOR algorithm, our new method can quickly verify the assertion properties of many programs that were previously intractable.

In summary, this paper makes the following contributions:

- We introduce the new notion of predicate dependence and propose a new static analysis method for computing the dependence conservatively.
- We propose a new cooperative static–dynamic analysis framework that leverages the new dependency relation to reduce the interleaving space at run time.

- We also propose two new optimizations for the DPOR algorithm to further improve the performance.
- We implement the new methods and demonstrate their effectiveness through experiments using open source benchmarks.

The remainder of this paper is organized as follows. We will establish notation in Section 2 before introducing predicate dependence in Section 3. We will present our new cooperative static–dynamic analysis framework in Section 4, and the two additional optimizations for DPOR in Section 5. Our experimental results will be presented in Section 6. We will review related work in Section 7, and finally, give our conclusions in Section 8.

2. PRELIMINARIES

This section provides the background information on existing dynamic analysis methods for multithreaded programs.

2.1 Concurrent System

A concurrent system is composed of a finite number of threads and a finite set of *communication objects*. Individually, each thread executes a sequence of operations of a sequential program. During dynamic analysis, operations on communication objects are considered *visible* whereas all other operations are considered *invisible* – they are not monitored during the program execution. We also assume that each visible operation is *atomic*, i.e., it can be executed on one communication object at a time without interference from the other threads. An operation is *blocking* if it cannot currently be executed. For example, a thread waiting for a mutex lock to be released by another thread is said to be blocked.

A *global state* is reached whenever the next operation for each thread is a visible operation. We assume that there exists at least one visible operation for each thread and that there is a unique initial state s_0. A transition from one state to another is the execution of a visible operation followed by any finite number of invisible operations by the same thread, ending just before another visible operation. The *state space* of a concurrent program is simply all the global states reachable from s_0 and all the transitions between these states. Following the notation used by Godefroid [15], we combine local (invisible) operations with the previous visible operation into one transition. Therefore, the state space is reduced by avoiding unnecessary interleavings of local operations. From here on, we will use the term *state* to mean global state.

Formally, a concurrent system can be modeled as a *transition system* $A_G = (S, \Delta, s_0)$ where S is the set of states for the system, $\Delta \subseteq S \times S$ is the *transition relation*, and s_0 is the initial state. Let $\mathcal{T}$ be the set of all transitions for the system, and $\mathcal{T}^*$ be the set of all finite words (all sequences of transitions) that can be created from $\mathcal{T}$. We use $s \xrightarrow{t} s'$ to mean that executing $t \in \mathcal{T}$ from s leads to state s'. We use $s \overset{w}{\Rightarrow} s'$ to mean that executing the finite sequence of transitions $w \in \mathcal{T}^*$ leads from s to s'. A state s' is said to be *reachable* from s if there exists some w such that $s \overset{w}{\Rightarrow} s'$.

A transition is *disabled* in the state s if its visible operation is blocking, implying that the transition cannot currently be executed. If a transition is not disabled, it is *enabled* in s. Two transitions are *co-enabled* if there exists some state where they are both enabled.

2.2 Stateless Model Checking

The state space of a concurrent system can be fully explored by using a *stateless model checking* [15] procedure, which systematically executes the program under all possible thread schedules. Different from the classic model checkers [7] which are typically *stateful*, here the search is carried out without explicitly storing any system state. Instead, a system state is uniquely identified by the sequence of transitions executed, starting from the initial state s_0. In other words, instead of exploring the reachable states of the system, the procedure systematically explores the set of execution traces. When dealing with full-fledged programming languages such as C/C++/Java, where the system state consists of values for all memory locations that can be accessed by a thread, stateless model checking is a far more practical method.

To exhaustively explore the interleaving space, we must check at least one representative thread interleaving from each *equivalence class* of interleavings, also called the Mazurkiewicz trace [26]. This can be regarded as the theoretical foundation of partial-order reduction (POR). More formally, Mazurkiewicz traces [26], in the context of concurrent systems, are defined as equivalence classes of sequences of transitions. Let $\mathcal{D} \subseteq \mathcal{T} \times \mathcal{T}$ be a valid dependency relation between transitions. Two sequences of transitions over $\mathcal{T}$ are equivalent if the two traces can be obtained from each other by successively exchanging adjacent independent transitions. Consider two transitions t and t'. If t and t' are independent, then the sequences containing tt' and $t't$, respectively, are in the same equivalence class. This implies that the concurrent program will end up in the same state regardless of the execution order of t and t'.

Computing the equivalence classes rests on the concept of a dependency relation. In classic POR methods such as DPOR, the dependency relation is typically defined with respect to the concurrent system itself, without considering the properties to be checked. Godefroid [15] formalizes the general requirement for a relation over concurrent operations to be a dependency relation as follows:

Definition 1. Let $\mathcal{T}$ be the set of transitions and $\mathcal{D} \subseteq \mathcal{T} \times \mathcal{T}$ be a binary, reflexive, and symmetric relation. $\mathcal{D}$ is a *valid dependency relation* for the system iff for all $t_1 t_2 \in \mathcal{T}, (t_1, t_2) \notin \mathcal{D}$ (t_1 and t_2 are independent) implies that the two following properties hold for all states s in the state space A_G of the system:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s' (independent transitions can neither disable nor enable each other), and
2. if t_1 and t_2 are enabled in s such that $s \xRightarrow{t_1 t_2} s'$ and $s \xRightarrow{t_2 t_1} s''$, then s' must be the same as s'' (commutativity of enabled independent transitions).

A stricter definition of the dependency relation would result in fewer equivalence classes, which in turn corresponds to fewer executions to be explored. However, accurately computing the dependence can be difficult; the DPOR algorithm uses *conflict dependence* mainly because it is easy to compute.

A subset T of the transitions enabled at a state s is said to be *persistent* in s if each transition not in T does not interact with T. It has been proved [15] that exploring only transitions in the persistent set of each state guarantees detection of all deadlock and assertion violations. Below is a formal definition of persistent sets.

Definition 2. A set T of transitions enabled in a state s is *persistent* in s iff for all nonempty sequences of transitions

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

from s in A_G and including only transitions $t_i \notin T, 1 \leq i \leq n$, t_n is independent with all transitions in T.

2.3 Dynamic Partial Order Reduction

Early partial order reduction algorithms (such as in [15]) statically computed the persistent set, but limitations such as imprecise pointer alias information often caused the persistent set to over-approximate dependence of transitions, causing equivalent traces to be explored. Dynamic partial order reduction [13] addressed this issue by focusing on one execution trace at a time, where precise alias information can be obtained, and by computing the necessary transitions to explore dynamically using *backtrack sets*.

Algorithm 1 shows the pseudocode of the DPOR algorithm. The algorithm performs a depth-first search through the state space of A_G starting with the initial state s_0. The stack represents a finite *transition sequence* $S \in \mathcal{T}^*$, $t_1 t_2 \dots t_n$, from the states $s_1 \dots s_n$ such that $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \dots \xrightarrow{t_n} s_{n+1}$. Here, $dom(S)$ means the set $\{1, \dots, n\}$, $pre(S, i)$ for $i \in dom(S)$ refers to the state s_i, $last(S)$ refers to s_{n+1}, $next(s, p)$ is the unique transition to be executed by process p in state s, and $proc(t)$ is the thread that executed the transition t. A happens-before relation on a sequence S, $i \rightarrow_S p$, for $i \in dom(S)$ and process p is a relation indicating causality between the transition executed at i and p.

Each state s has a backtrack set, denoted $backtrack(s)$. The backtrack set of a state s represents the set of processes at s with enabled transitions that still need to be explored from s.

The DPOR algorithm starts by calling $Explore()$ with an empty stack. The stack represents the set of transitions executed to reach $last(S)$ (Line 2). Line 4 examines the next transition of each process from s ($next(s, p)$). The algorithm then examines S to find the last transition (if it exists) that is

1. dependent with $next(s, p)$,
2. may be co-enabled with $next(s, p)$, and
3. $i \not\rightarrow_S p$.

Step 1 uses the conflict dependency relation. Upon finding a conflicting transition i in S, a backtrack point is inserted in the state $pre(S, i)$. Which process to add into the backtrack set is determined on Line 5. Here, the algorithm attempts to find the set E of enabled processes in $pre(S, i)$ that happen-before $next(s, p)$ in the current sequence. The happens-before relation signifies causality; executing a transition of a process in E will cause the transition $next(s, p)$ to be executed. If such a causality relationship is not found (E is empty), then the algorithm over-approximates by adding all the enabled transitions in $pre(S, i)$ to the backtrack set.

Overall, the goal of Lines 5–10 is to first identify two dependent transitions in the current sequence and then insert a backtrack point to potentially reverse the order of execution of the dependent transitions in a future execution. The algorithm, on Lines 13 through 20, continues the depth-first search by exploring each non-explored transition from a state's backtrack set. The authors of [13] proved that the backtrack sets explore a set of transitions from each state s which is persistent in s. As a result, they are able to leverage the theorems from [15] to ensure that DPOR will find all deadlock and assertion violations in an acyclic concurrent program.

However, neither DPOR nor any other existing POR method considered the properties to be checked while computing the dependency relation. We shall show in the next section that, by taking the properties into consideration, we can often obtain a more refined dependency relation, which leads to a drastic reduction in the number of equivalence classes.

3. ASSERTION GUIDED ABSTRACTION

In this section, we introduce the new notion of *predicate dependence* to soundly reduce the number of thread interleavings. We refer to this method as *assertion guided abstraction*.

3.1 Predicate Dependency Relation

We modify the general requirement for a relation over concurrent operations to be a valid dependency relation by considering the influence on the outcome of assertion checking. The new requirement, as compared to Definition 1, is given as follows:

Definition 3. Let $\mathcal{T}$ be the set of transitions and $\mathcal{D} \subseteq \mathcal{T} \times \mathcal{T}$ be a binary, reflexive, and symmetric relation. $\mathcal{D}$ is a *valid dependency relation* for the system iff for all $t_1 t_2 \in \mathcal{T}$, $(t_1, t_2) \notin \mathcal{D}$ (t_1 and t_2 are independent) implies that the two following properties hold for all states s in the state space A_G of the system:

```
if (cond)
  x = 1;
```

```
x = 7;
x = 5;
y = x + 2;
```

Figure 4. Examples for control (left) and data dependency (right).

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s' (independent transitions can neither disable nor enable each other), and
2. if t_1 and t_2 are enabled in s such that $s \xRightarrow{t_1 t_2} s'$ and $s \xRightarrow{t_2 t_1} s''$, then s' can lead to a bad state iff s'' can lead to the same bad state (commutativity of enabled independent transitions is predicated on property checking).

In other words, two transitions t_1 and t_2 are *predicate-dependent* if and only if the relative execution order of t_1 and t_2 can affect whether an error state is reached or not. Otherwise, they are considered to be *predicate-independent*. An error state is any state where a property is violated. We focus on two type of errors in this work: assertion violations and deadlocks.

Definition 3 induces an abstraction of the system's interleaving space to ignore operations that are conflict-dependent and yet unrelated to the validity of the properties. Alternatively, the abstraction is transforming the original program into a simpler program, containing only those program statements that can cause an error. We call these program statements *essential statements*.

In the next subsection, we explain that by using predicate dependence to replace conflict dependence in DPOR we can obtain a new dynamic analysis method that is more efficient and at the same time guarantees that no error states are missed. Here, the *error state* is a state where the assertion fails or a deadlock occurs. The *essential statements* are all the statements which could affect assert and lock/unlock calls. Throughout this section, we shall focus our discussion on dealing with a single assertion statement. The case for multiple assertions and/or lock calls will be similar.

3.2 Correctness of the Reduction

First, we introduce the concept of control and data dependencies. Figure 4 (left) shows an example of the control dependency. The statement on Line 1 determines if the statement on Line 2 is executed. In other words, Line 2 is control dependent on Line 1. In general, a statement b is said to be *control dependent* on another statement a if and only if there exists a path from a to b such that every statement $c \neq a$ in the path is post-dominated by b and a is not post-dominated by b.

Figure 4 (right) shows an example of a data dependency. The value of y on Line 2 is dependent on the value of x on Line 1. A statement a is said to be data dependent on another statement b if and only if both statements access the same memory location and at least one of them stores into it, and there is a feasible run-time execution path from a to b.

Furthermore, the order in which statements are executed determines their dependency relation. For example, Line 3 of Figure 4 (right) is data dependent on Line 2 but not Line 1. This is because the write on Line 2 to `x` overwrites the write on Line 1. The same idea applies if all three lines of Figure 4 (right) were executed by three different threads. Line 3 would be dependent on either Line 1 or Line 2 depending on the thread scheduling. We will use this concept of changes in dependencies to prove our method is sound.

To prove that predicate dependence based reduction is correct, we use Theorem 2.2 from [25] as a lemma, which states:

LEMMA 1. *Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.*

Algorithm 1 Classic dynamic partial order reduction algorithm.

```
    Initially: Explore(∅)
 1: procedure EXPLORE(S)
 2:     s ← last(S)
 3:     for all processes p do
 4:         if ∃i = max({i ∈ dom(S) | S_i is dependent and may be co-enabled with next(s,p) and i ↛_S p}) then
 5:             E ← {q ∈ enabled(pre(S,i)) | q = p ∨ ∃j ∈ dom(S) : j > i ∧ q = proc(S_j) ∧ j →_S p}
 6:             if E ≠ ∅ then
 7:                 add any q ∈ E to backtrack(pre(S,i))
 8:             else
 9:                 add all q ∈ enabled(pre(S,i)) to backtrack(pre(S,i))
10:             end if
11:         end if
12:     end for
13:     if ∃p ∈ enabled(s) then
14:         backtrack(s) ← {p}
15:         done ← ∅
16:         while ∃p ∈ (backtrack(s) \ done) do
17:             add p to done
18:             Explore(S.next(s,p))
19:         end while
20:     end if
21: end procedure
```

The proof for Lemma 1 in [25] was based on the fact that a *single* statement will produce a different result if and only if its dependencies change. Thus, if every statement has its dependencies preserved, the program will not produce a different result.

Definitions such as conflict dependence use Lemma 1 to test all possible outcomes of the program caused by concurrent nondeterminism. Each reordering performed by conflict dependence is a change in the dependencies of the program. However, this is often unnecessarily strong. We are not interested in how reordering affects the entire program but only essential statements. We present this idea in the following corollary:

COROLLARY 1. *Any reordering transformation that preserves every dependence of a statement will not affect the outcome of that statement.*

Using Corollary 1, we can prove that using predicate dependence in DPOR to replace the conflict dependence will result in a sound reduction, which is stated formally as follows:

THEOREM 1. *Predicate dependence, as defined in Definition 3, will not cause any error state to be missed.*

PROOF. First, two transitions, t_1 and t_2, affect the reachability of an error state only if they are control or data dependent with the essential statements (e.g., assertions, lock/unlock calls). If they are neither control nor data dependent with any essential statement, then based on the definitions of control/data dependence and essential statements, they cannot affect the reachability of the error state.

Now we prove the theorem by contradiction. Assume that t_1 and t_2 are not predicate dependent, but one of their two execution orders can result in an error state s_{err} being missed.

- Since one of the two execution orders leads to s_{err} being missed, by Corollary 1 this means that the order of t_1 and t_2 does not preserve the dependencies of the essential statements.
- However, if the dependencies of the essential statements are not preserved, then by definition, t_1 and t_2 are predicate dependent, which contradicts our assumption (that they are not predicate dependent).

Therefore, our assumption is not correct; the theorem is proved. □

However, there is difficulty in using Theorem 1 during a purely dynamic analysis, because checking whether t_1 and t_2 are control/-data dependent at run time is not an easy task. Although in theory, we could have pre-computed the control/data dependency relation between all pairs of potentially concurrent operations in a purely static manner before starting the dynamic analysis, it would be computationally expensive, and at the same time, difficult to obtain accurate results due to the limitations in a static inter-procedural, inter-thread, dependency analysis. Instead, we propose a new cooperative static–dynamic framework. The idea is to get the best of both worlds, since static analysis is able to get an approximation of the entire program while dynamic analysis is able to provide information on aliasing and feasible executions.

4. THE COOPERATIVE ANALYSIS

In this section, we provide an overview of our new method shown in Algorithm 2. The input is the program under test (*P*) together with data input (*I*). Subprocedure *Instrument* adds monitoring and control capabilities to the program to prepare it for dynamic analysis. Subprocedure *IfConvert* converts all assertion statements of the form `assert(c)` into `if(!c) ERROR`. Subprocedure *DependenceCalculate* analyzes the program and returns the dependency relationships. Subprocedure *exec* runs the program-under-test with our scheduler. Subprocedure *Explore* takes the dependency relations and performs an exploration of the interleaving space.

Algorithm 2 High level overview of our new analysis method.

```
1: P ← program under test
2: I ← program inputs
3: P ← Instrument(P)
4: P ← IfConvert(P)
5: dep ← DependenceCalculate(P)
6: exec(P, I)
7: Explore(∅, dep)                          ▷ Algorithm 4
```

The bulk of the static analysis takes place in *DependenceCalculate*. This procedure collects all the interprocedural control and data dependencies of each statement of interest (e.g., assert, lock-/unlock calls) while ignoring aliasing. We will explain how we deal with aliasing in subsection 4.2. The process of collecting interpro-

cedural control and data dependencies is the generation of the interprocedural slice of each statement. We will explain our method for computing the interprocedural slice in subsection 4.1. For a more comprehensive description of existing methods for creating the interprocedural slice, please refer to [21] and [41].

4.1 Control and Data Dependency Analysis

In our cooperative framework, we divide our approach to inter-thread control and data dependency analysis into three steps:

1. intraprocedural control and data dependency computations,
2. interprocedural control and data dependency computations,
3. inter-thread alias computations.

Steps 1 and 2 are purely static and they are carried out only once, before the dynamic analysis procedure starts, whereas Step 3 is dynamic – the alias information is updated incrementally while the DPOR algorithm is running.

The static analysis methods used in this application are required to be over-approximated. That is, as long as the execution order of two concurrent operations t_1 and t_2 *may* affect the property, the static analysis method must ensure that t_1 and t_2 are dependent.

Our method uses an intermediate program representation known as the *program dependence graph* (PDG) [12]. For a given procedure, a PDG explicitly provides both the control and data dependencies. The transitive closure of control dependences is the control dependence graph; a node in the control dependence graph is a statement and an edge from node x to node y indicates that y is control dependent on x. The data dependence graph can be created similarly. When combined for a single procedure, the control and data dependence graphs are the two subgraphs of the program dependence graph [12].

The system dependence graph [21] is an interprocedural version of the program dependence graph. It consists of the program dependence graphs for each procedure as well as additional edges to include (1) direct dependencies between call site and called procedure, and (2) transitive dependencies due to calls. As an example, consider the program in Figure 5 (top) and its system dependence graph (bottom). Control dependencies between statements are represented as edges with a diamond head, edges with an arrow head are data dependencies, and dashed edges are dependencies due to function calls and parameter passing.

The parameter inputs of function *add* are represented in the nodes a_{in} and b_{in} and the output in the node a_{out}. To handle parameter passing, two additional nodes in *main* are added ($a_{in} = sum$ and $b_{in} = i$) and connected to the *add* procedure. These extra nodes can be thought of as stack frames to handle pass-by-value semantics. The program dependence graph of *main* and *add* are the nodes reachable from the nodes *Enter main* and *Enter add* excluding function calls and parameter passing (dashed edges).

For a given node s (statement) in the system dependence graph, an interprocedural *slice* is the graph containing all nodes that can reach s. The usefulness of a slice on s is that it contains *all* possible statements that could influence s (Corollary 1).

4.2 Computing Alias Information on the Fly

For languages such as C/C++, aliasing becomes an issue for creating slices. Consider a slice S on program statement s ignoring aliasing. S contains only the statements which could directly affect either the execution of s (control dependencies) or the value used by s (data dependencies). In the presence of aliasing, S also needs to contain any other statement in the program which could alias to any statement already in S. These are additional data dependencies caused by aliasing. Together, they represent the complete set of all statements that could influence s.

To see the effects of aliasing, consider the example program in Figure 6. A thread is accessing the first six elements of a shared array of 12 elements. Similarly, `main` takes an index as input from

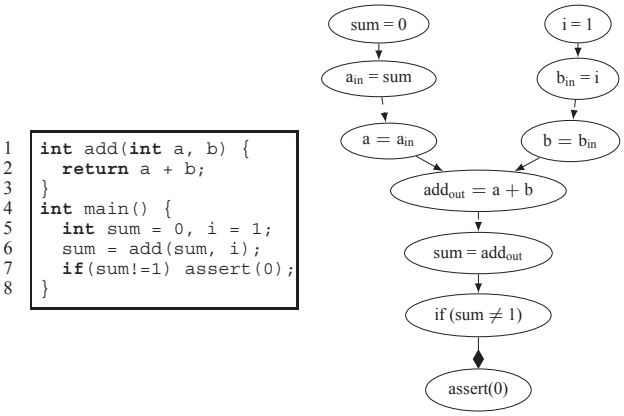

Figure 5. A program (top) and its system dependence graph (bottom). Diamond headed edges are control dependencies while arrow headed edges are data dependencies. Dashed lines represent function calls and parameter passing.

```
1   int array[12];
2   void thread1() {
3     for (int i = 0; i < 6; ++i) {
4       array[i] = array[i] + 1;
5     }
6   }
7   int main(int argc, char *argv[]) {
8     int idx = atoi(argv[1]) % 12;
9     thread_create(thread1);
10    array[idx] = array[idx] + 1;
11  }
```

Figure 6. A example program showing the effects of aliasing.

the user (Line 8) and increments the array at that index. Aliasing could occur between the two accesses to the array (Lines 4 and 10). The situation is complicated because the array index accessed by `main` is based on user input; a conservative static analysis would assume that the user could pass anything. Thus, the statements on Line 4 and 10 would always alias. We will show in this subsection how our cooperative static–dynamic approach avoids this problem.

The program representation shared between the static and dynamic analysis frameworks are program statement IDs. Specifically, each statement in a program is given a unique integer ID. Intuitively, this provides a method of communication between the two frameworks. In our cooperative analysis method, the output of the static analysis is a set of program statement IDs representing the slice on each erroneous statement ignoring aliasing. The goal of our dynamic analysis is to extend the slices with inter-thread dependency information. Dynamic partial order reduction (DPOR) fits this task perfectly; the goal of DPOR is to dynamically enumerate all relevant traces of a concurrent program.

The static analysis is made simpler since it no longer has to reason about complex thread interactions (such as mutex locks) or inter-thread aliasing. Also, the issue of calculating dependencies becomes not only simple for a dynamic analysis but it is also accurate; it is guaranteed that any possible dependencies observed by the dynamic analysis are ones which could possibly occur in the program (there are no false positives).

DPOR generates a set of sequences of program transitions each corresponding to an execution of the program. The entire set of sequences produced by DPOR contains at least one sequence from each equivalence class. The transitions are dynamic instances of each program statement. Due to their dynamic nature, the sequences of transitions contain the memory address used in every memory

```
1  int a;
2  void thread1(void) {
3    a = 0;
4  }
5  void thread2(void) {
6    a = 1;
7    if (a != 1)
8      assert(0);
9  }
```

Figure 7. Example to show static–dynamic slice creation.

read and write; in the context of static analysis, this means that we have complete alias information for the entire sequence.

Algorithm 3 shows the pseudocode for *UpdateSlice*, which takes as input a transition sequence (ρ) generated from DPOR and a set of statements on a slice *Sli*. It updates *Sli* to contain the inter-thread dependencies observed in ρ. The for-loop on Lines 5–11 checks each pair of transitions accessing the same object in ρ to see if they should be added to the slice. We use *obj(t)* to represent the object accessed by *t*, *Sli.contains(t)* to return true if the statement executed by transition t is on the slice and *Sli.insert(t)* inserts the statement executed by t to the slice. Note that $\oplus$ denotes XOR.

Statements should be added to the slice if one of the statements is already on the slice and they are accessing the same object. This is the situation when a statement on the slice aliases to one not on the slice. The entire procedure is a fixpoint computation until the slice is no longer updated. The fixpoint is required because whenever a transition t is added to the slice all the statements not on the slice which are dependent with t also need to be added.

Algorithm 3 Procedure to update the slice Sli with the dependence information observed in the sequence of transitions ρ.

```
1:  procedure UpdateSlice(ρ, Sli)
2:      SliceUpdated ← true
3:      while SliceUpdated do
4:          SliceUpdated ← false
5:          for all t1, t2 ∈ ρ such that obj(t1) = obj(t2) do
6:              if Sli.contains(t1) ⊕ Sli.contains(t2) then
7:                  SliceUpdated ← true
8:                  Sli.insert(t1)
9:                  Sli.insert(t2)
10:             end if
11:         end for
12:     end while
13: end procedure
```

The combination of dynamically calculated inter-thread aliasing using $UpdateSlice$ and statically calculated control and data dependencies completes the slicing algorithm. In the next section, we show how this algorithm is combined with DPOR to implement predicate dependence. Note that intra-thread aliasing is handled statically during the control and data dependency phase.

Example. Consider the multithreaded program in Figure 7, where two threads access a shared variable `a`. Assume that the line number of each statement represents the statement ID. First, we generate the slice on the assertion ignoring aliasing. The slice contains Lines 8 (the assertion itself), 7 (a control dependency) and 6 (a data dependency). Notice that the slice is missing a crucial component, the aliased write to `a` by the first thread on Line 3.

DPOR generates three sequences of transitions for this example. They are: $S_1 = 6, 7, 3$, $S_2 = 6, 3, 7, 8$ and $S_3 = 3, 6, 7$. Next, we run $UpdateSlice$ on each sequence; the results are summarized for S_1 in Table 1. The table shows how the slice is updated for each pair examined in S_1 (i.e., the table is the first iteration of the while-loop in Algorithm 3). Column 3 is the value of *SliceUpdated* in the fixpoint computation for S_1. While examining S_1, the slice is updated; Column 4 stays true until after the first iteration of the fixpoint computation. During the next iteration on S_1, no updates are made to the slice. Examining the remaining two sequences shows that the final slice contains $\{ 8, 7, 6, 3 \}$; the update to `a` by Thread 1 is now included.

Sequence	Pair	Slice	Slice Updated?
Initially:		$\{ 8, 7, 6 \}$	
S_1	$(3, 6)$	$\{ 8, 7, 6, 3 \}$	true
S_1	$(3, 7)$	$\{ 8, 7, 6, 3 \}$	true
S_1	$(6, 7)$	$\{ 8, 7, 6, 3 \}$	true

Table 1. Example of running $UpdateSlice$ (Algorithm 3) on a sequence of transitions generated by DPOR from the program in Figure 7. The procedure continues to run on the remaining sequences but no modifications are made to the slice

4.3 DPOR based on Predicate Dependence

Now, we explain how DPOR (Algorithm 1) can be modified to include predicate dependence. The only additional input is a set of program statements on the slice (ignoring aliasing) of every essential statement. We incorporate the *UpdateSlice* algorithm to dynamically calculate aliasing from the previous subsection.

The major change to Algorithm 1 is that backtrack set is computed at the end of each execution. This is required because we need to examine the entire sequence of transitions of an execution in order to dynamically update the statements contained in the slice.

Algorithm 4 introduces an additional notation to describe a sequence of transitions S and a state s. Here, $predom(S, s)$ is the set from $\{ 1, ..., n \}$ where n is the number of transitions that have occurred before the state s. This the same as $dom(S)$ if S only contained the sequences of transitions leading up to s.

Lines 13–24 implement updating the backtrack set (similar to Lines 3–11 in Algorithm 1) after the entire execution is completed. Line 12 is the fixpoint computation, Algorithm 3, to dynamically expand the slice to include alias information from the current sequence. The dependency relation (Line 15) is from Definition 3. Other than these modifications, Algorithm 4 is the same as Algorithm 1.

5. OPTIMIZING PREDICATED DPOR

In this section, we introduce two new optimizations in DPOR. One optimization, called critical section peeking, is applicable to all properties, whereas the other optimization, called write–write pruning, is applicable to checking assertions.

5.1 Critical Section Peeking

Definitions such as predicate dependence and conflict dependence consider mutex lock calls to be dependent if they are locking the same mutex and never consider the items in the critical section of the mutex. However, they can be unnecessarily inefficient in many cases. As motivation, consider the program in Figure 8. Two threads are incrementing values in an array of 16 integers. Thread 1 is incrementing items 0–7 and thread 2 is incrementing items 8–15. A standard DPOR implementation requires 12,870 runs to test this program even though the two threads will never access the same memory location. In contrast, our new automated optimizations can reduce the the exploration down to one run.

We define *critical section peeking* as follows. Let m_1 and m_2 be two mutex lock calls and cs_1 and cs_2 be the statements in the critical sections protected by m_1 and m_2, respectively. Two mutex lock calls are dependent iff condition 1 is true and either condition 2 or condition 3 is true:

1. The two lock calls are to the same mutex

Algorithm 4 Predicated dynamic partial order reduction algorithm.

1: $Slices \leftarrow$ slice of every essential statement, ignoring aliasing
   Initially: Explore($\varnothing, Slices$)
2: **procedure** EXPLORE($S, Slices$)
3: $s \leftarrow last(S)$
4: **if** $\exists p \in enabled(s)$ **then**
5: $backtrack(s) \leftarrow \{p\}$
6: $done \leftarrow \varnothing$
7: **while** $\exists p \in (backtrack(s) \setminus done)$ **do**
8: add p to $done$
9: $Explore(S.next(s,p), Slices)$
10: **end while**
11: **end if**
12: $UpdateSlice(S, Slices)$ ▷ Algorithm 3
13: **for all** $States\ s'$ in S **do**
14: **for all** $processes\ p$ **do**
15: **if** $\exists i = max(\{i \in predom(S,s') \mid S_i$ is predicate dependent and may be co-enabled with $next(sp)$ and $i \not\rightarrow_S p\}$) **then**
16: $E \leftarrow \{q \in enabled(pre(S,i)) \mid q = p \vee \exists j \in predom(S,S') : j > i \wedge q = proc(S_j) \wedge j \rightarrow_S p\}$
17: **if** $E \neq \varnothing$ **then**
18: add any $q \in E$ to $backtrack(pre(S,i))$
19: **else**
20: add all $q \in enabled(pre(S,i))$ to $backtrack(pre(S,i))$
21: **end if**
22: **end if**
23: **end for**
24: **end for**
25: **end procedure**

```
mutex array_lock;
int array[16];
void thread_1() {
  for (int i = 0; i < 8; ++i) {
    lock(array_lock);
    array[i] = array[i] + 1;
    unlock(array_lock);
  }
}
void thread_2() {
  for (int i = 8; i < 16; ++i) {
    lock(array_lock);
    array[i] = array[i] + 1;
    unlock(array_lock);
  }
}
```

Figure 8. A motivating example for critical section peeking. A single mutex protects an entire array. This causes a DPOR algorithm to test all orderings of array accesses even if two threads are accessing different items of the array.

2. $\exists s \in \{\ cs_1 \cup cs_2\ \} \mid s$ is not a memory accessing transition
3. $\exists s_1 \in cs_1 \mid \exists s_2 \in cs_2 \mid s_1$ is dependent with s_2

In DPOR, when two mutex lock calls are reordered the effective result is that *all* of the transitions in the critical sections are reordered. In essence, critical section peeking only reorders critical sections when it is necessary. Item 2 prevents deadlocks from being missed; two critical sections containing additional mutex lock calls could, when called in a certain order, result in a deadlock even if they are not dependent on each other. Item 3 results in the significant reduction for programs such as in Figure 8; the mutexes only need to reordered if they are protecting memory accesses which could interfere with each other. When using critical section peeking, none of the mutex lock calls in Figure 8 will be dependent; this results in only *one* run of DPOR to be required. Critical section peeking is implemented dynamically by examining the statements in the critical section of two lock calls to the same mutex using the dependence rules defined previously.

5.2 Write–Write Pruning

There are three combinations of shared memory access events between two threads: read–read, read–write, and write–write. Two read accesses, even from two different threads, can never affect each other regardless of their order of execution. Thus, in this section, we focus on read–write and write–write, and show how supplemental information can generate fewer Mazurkiewicz traces.

```
int a = 0;
void t1_main(){
  a = 7;
  a = 6;
}
void t2_main() {
  a = 0;
  a = 1;
}
int main(int argc, char *argv[]) {
  thread_create(t1_main);
  thread_create(t2_main);
  thread_join(t1_main);
  thread_join(t2_main);
  assert(a != 7);
  return 0;
}
```

Figure 9. A read–(write–write) conflict between three threads.

We divide write–write conflicts into two new categories: write–write and read–(write–write) conflicts.

THEOREM 2. *If two writes to the same shared variable, x, by different threads, t_1 and t_2, are adjacent in a trace, then regardless of the order of the writes to x by t_1 and t_2, no assertion violations local to t_1 and t_2 will be missed.*

The key restriction to Theorem 2 is that assertions inside the two threads are never violated. There may be assertions in other threads that could be violated due to the order of writes by the two threads. This can be thought of as a read–(write–write) dependency; the

order of two writes affects the value read by another thread. In DPOR, this can only happen when there is a third thread, t_3, which reads the same location in memory as is written to by t_1 and t_2, and t_3 may not be co-enabled with t_1 and t_2 while they are writing to x. We formalize this concept in the following theorem.

THEOREM 3. *If two different threads, t_1 and t_2, are executing a sequence of operations, S_1 and S_2 respectively, both including some number of writes to the same shared memory location, x, then the order of execution of t_1 and t_2 writing to x will only affect assertion violations if both of the following hold:*

1. *there exists a shared memory read of x by a distinct third thread t_3 which cannot be co-enabled with either t_1 or t_2 during S_1 or S_2,*
2. *the write to x in S_1 by t_1 is the last write to x by t_1 before t_3 is enabled, and*
3. *the write to x in S_2 by t_2 is the last write to x by t_2 before t_3 is enabled.*

The case when t_3 can be co-enabled during either S_1 or S_2, violating condition one of Theorem 3, changes the problem from a read–(write–write) conflict to simply a read–write conflict. Consider the example of a write–write conflict between two threads and a single assertion check in Figure 9. Furthermore, `main` creates and joins the two threads. The key insight is that `main` can never be co-enabled with any of the writes. This situation is captured by Theorem 3; the read by `main` will only be able to read the last value written by each thread (either 6 or 1). DPOR would require 6 runs but in reality only 2 runs are necessary.

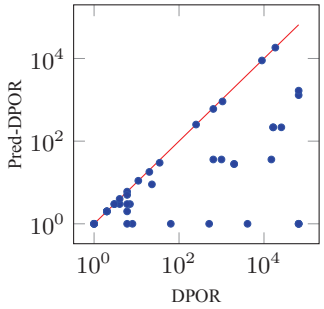

Figure 10. Results: comparing the number of runs of predicated DPOR and baseline DPOR on 46 benchmark examples.

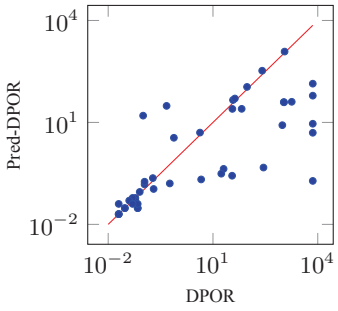

Figure 11. Results: comparing the execution time of predicated DPOR and baseline DPOR on 46 benchmark examples.

6. EXPERIMENTAL RESULTS

We have implemented our new method in a tool called *Käse* based on the LLVM platform for static analysis and code instrumentation and on a modified version of *Inspect* [42] for systematic exploration of a concurrent program. Our tool runs both with and without the proposed optimizations and can handle unmodified C/C++ code using PThreads. We use the DPOR implementation in *Inspect* as a baseline for comparison.

Our experiments were designed to answer the following research questions: (1) How effective is our new method? In practice, is it able to show significant improvement over DPOR? (2) How scalable is our new method? Can it handle realistic programs?

We evaluated our tool on 46 benchmarks from two groups. The first group are a set of small programs from the Software Verification Competition (SV-COMP) [34] as well as two of our own synthetic examples. The second group is a set of real-world open source programs: `nbds` [28] is a C implementation of several non-blocking data structures. `nedmalloc` [29] is a thread-caching malloc implementation. `pfscan` is a parallel directory file scanner. When possible, we used tests and inputs provided by the authors.

Figures 10 and 11 summarize the results of all our tests. The graphs show a comparison of our methods versus DPOR in terms of number of runs and runtime required to test a program respectively. Figure 10 shows that our method is always capable of testing a program in the same number of runs as DPOR and in some cases can offer significant reduction. Figure 11 shows similar results for the runtime. In most cases, our tool incurs a minimal overhead and can results in significant reduction in runtime. The cases where the runtime was significantly higher occurred when a reduction in runs occurred but the static analysis took longer than the saving incurred during dynamic analysis.

Table 2 shows the statistics from tests where a reduction in runs occurred. Column 1 shows the name of each benchmark. Column 2 shows the number of lines of code in the benchmark. Column 3 shows the number of assertions in the benchmark. Column 4 shows the maximum number of threads in the benchmark. Columns 5–7 show the time required to test the program for DPOR, predicate dependence, and predicate dependence with optimizations, respectively. Similarly, columns 8–10 show the number of runs for each method. We allotted a maximum of two hours for each test; an × in columns 5–10 indicates that the method exceeded two hours. All tests were run on a machine with a 2.60 GHz Intel Core i5-3230M processor with 8 GB RAM and a 64-bit Linux OS.

First, the results show that our method is more efficient than DPOR. On the first set of benchmarks, both DPOR and our method can complete and the difference in runtime is small. However, on large programs, such as `nbds`, `pfscan` and `nedmalloc`, DPOR could not finish whereas our method was able to finish in a reasonable amount of time. For some experiments, our new optimizations were required to have good performance, since the programs make heavy use of mutexes.

Second, as a measure of the scalability of our method, we conducted tests on two parameterized programs: `IndexerSafe` and `nbds-hashtable`. `IndexerSafe` is an implementation of Figure 1. We varied the number of threads from 11 to 15; the results are summarized in Figure 2. The number of runs required for DPOR grows exponentially with the number of threads while optimized predicate dependence stays at a constant one run. Critical section peeking was required for this benchmark since the compare-and-swap operations were implemented using mutex locks. The program `nbds-hashtable` was parameterized by the number of compare-and-swap operations used by two threads. We varied the number of operations from four to eight. The results are summarized in Figure 12. Predicate dependence both takes a lower number of runs and has slower growth when compared to DPOR. Once 8 operations are performed, DPOR exceeds the two hour time limit while predicate dependence is able to finish in just over two minutes. Experiments where predicated DPOR finishes in one run are possible because our method concludes that the property does not depend on concurrent non-determinism.

Table 2. Experimental results for a subset of the test programs to illustrate the impact of predicated DPOR and the two optimizations. *LOC* is the number of lines of code. *Assert* is the number of assertions in the file. *Thread* is the maximum number of threads running in the program. Results are given for different levels of optimization: DPOR is the original DPOR implementation. Pred-DPOR is the predicated DPOR. Pred-DPOR-opt is the predicated DPOR with optimizations. For the Pred-DPOR columns, the time includes the static analysis required to create the slice. The maximum testing time was two hours; tests exceeding this time are marked with an ×.

				Time (s)			Runs		
Name	LOC	Assert	Thread	DPOR	Pred-DPOR	Pred-DPOR-opt	DPOR	Pred-DPOR	Pred-DPOR-opt
AccountBad	60	1	4	0.05	0.06	0.05	4	4	3
BluetoothBad	88	1	2	0.19	0.11	0.11	23	9	9
ReadReadWrite	50	1	3	0.07	0.04	0.03	7	3	3
ReadWriteLock	55	1	5	22.03	0.49	0.41	1983	28	28
Stateful	54	1	3	0.02	0.03	0.02	6	6	2
IndexerSafe12	92	1	12	0.57	0.69	0.15	8	8	1
IndexerSafe13	92	1	13	4.64	5.28	0.17	64	64	1
IndexerSafe14	92	1	14	36.03	43.07	0.22	512	512	1
IndexerSafe15	92	1	15	282.06	357.24	0.43	4096	4096	1
nbds-list	1887	1	3	×	0.16	0.19	×	1	1
nbds-hashtable4	2375	24	3	37.11	25.04	25.02	641	36	36
nbds-hashtable5	2375	24	3	68.05	25.35	25.30	999	36	36
nbds-hashtable6	2375	24	3	1082.00	38.76	37.74	16441	216	216
nbds-hashtable7	2375	24	3	1828.59	74.32	73.08	25623	216	216
nbds-hashtable8	2375	24	3	×	140.28	138.04	×	1296	1296
nbds-hashw01	2322	1	3	39.36	45.50	46.69	641	601	601
nbds-hashw02	2322	1	3	1173.25	1199.29	1211.83	16441	14425	14425
nbds-hashw03	2234	1	3	942.99	8.29	7.66	14923	36	36
nbds-skiplistU1	1942	16	3	1.05	3.13	3.14	35	30	30
nbds-skiplistU2	1942	16	3	43.67	49.13	49.40	1057	913	913
nbds-skiplist	1994	1	4	×	0.21	0.21	×	1	1
nedmalloc	6303	9	5	×	9.148	9.138	×	1	1
pfscan	934	1	3	×	×	61.24	×	×	1666

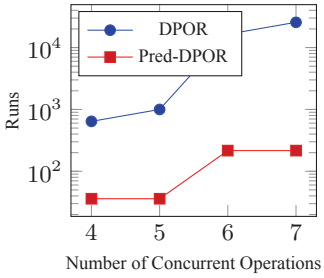

Figure 12. The number of runs versus the number of CAS operations per thread in test program nbds-hashtable.

7. RELATED WORK

There is a large body of work in the model checking literature on soundly reducing the state space of a concurrent system, including persistent sets [16], stubborn sets [36], ample sets [32], sleep sets [14, 17], wakeup trees [1], symmetry [43], and property driven pruning [39, 37]. There are also POR methods for SAT/SMT based bounded model checking [40, 23, 3]. However, they do not exploit the synergy between static and dynamic analysis in a cooperative framework. Godefroid and Pirottin [16] introduced additional dependence relations compared to conflict dependence to refining operations for variables of certain types. Also, they introduce the idea of conditional dependencies which are valid only at specific states in a concurrent program as opposed to all states. However, they do not perform property driven reduction.

Coverage guided approaches to reducing the interleaving space consider the space fully explored when a certain coverage condition is met. These methods include preemption bounds [8, 27], fair bounds [8], delay bounding [11], HaPSet [38], variable bounds [5], and thread bounds [5]. For example, a preemption bound of n means that all sequences of transitions at which no more than n preemptive context switches occur will be explored. The goal of these methods is not verification but accelerated bug detection. Our method of assertion guided abstraction can work along side coverage metrics to potentially provide further reduction.

There are non-systematic techniques for testing concurrent programs as well. Recent empirical studies of these algorithms can be found in [35, 20, 19]. For example, ConTest [10] inserts delays at synchronization points to attempt to increase contention and force deadlocks during testing. Two stage analysis systems such as CTrigger [31], CalFuzzer [22], PENELOPE [33], and Maple [44] operate by first statically or dynamically analyzing a program to identify potentially buggy interleavings. Then, the tools take control of the scheduler and attempt to force the buggy interleavings. While these methods scale well, our approach differs in that we guarantee not to produce any false negatives.

There is a large body of work on dynamic slicing [2], which use a similar method of examining dynamic sequences of transitions for a given program input to build dependence information. However, their focus was primarily on slices of single executions. Our work expands on theirs to handle concurrent non-determinism across multiple executions. Zhang *et al.* [45] expand on the early work to create more precise and efficient dynamic slicing algorithms. Additional work [4, 6, 9, 24, 30, 18] has been done to limit the number of dynamic instrumentation points to reduce the overhead of dynamic slicing. They are orthogonal to the method proposed in this paper.

8. CONCLUSIONS

We have presented a new cooperative static–dynamic analysis method for reducing the interleaving space of multithreaded C/C++ programs. We have also presented two optimizations for DPOR to provide further reduction. We have implemented our new methods and evaluated them on open source benchmarks. Our experimental evaluation shows that the proposed methods can result in significant speedup over DPOR alone. For future work, we plan to increase the accuracy and efficiency of our static analysis method.

9. ACKNOWLEDGMENT

This work was primarily supported by the NSF under grant CCF-1149454 (Markus Kusano). Partial support was provided by the ONR under grant N00014-13-1-0527 (Chao Wang).

10. REFERENCES

[1] P. A. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal dynamic partial order reduction. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 373–384, 2014.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, 1990.

[3] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *International Conference on Computer Aided Verification*, pages 141–157, 2013.

[4] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994.

[5] S. Bindal, S. Bansal, and A. Lal. Variable and thread bounding for systematic testing of multithreaded programs. In *International Symposium on Software Testing and Analysis*, pages 145–155, 2013.

[6] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, Oct. 1991.

[7] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[8] K. E. Coons, M. Musuvathi, and K. S. McKinley. Bounded partial-order reduction. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 833–848, 2013.

[9] E. Duesterwald, R. Gupta, and M. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Languages and Compilers for Parallel Computing*, pages 497–511. ACM, 1993.

[10] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Syst. J.*, 41(1):111–125, Jan. 2002.

[11] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 411–422, 2011.

[12] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[13] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 110–121, 2005.

[14] P. Godefroid. Using partial orders to improve automatic verification methods. In *International Conference on Computer Aided Verification*, pages 176–185, 1991.

[15] P. Godefroid. Model checking for programming languages using verisoft. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[16] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *International Conference on Computer Aided Verification*, pages 438–449, 1993.

[17] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *International Conference on Computer Aided Verification*, pages 332–342, 1992.

[18] R. Gupta, M. L. Soffa, and J. Howard. Hybrid slicing: Integrating dynamic information with static analysis. *ACM Trans. Softw. Eng. Methodol.*, 6(4):370–397, Oct. 1997.

[19] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *International Symposium on Software Testing and Analysis*, pages 210–220, 2012.

[20] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel. The impact of concurrent coverage metrics on testing effectiveness. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 232–241, 2013.

[21] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 1988.

[22] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *International Conference on Computer Aided Verification*, pages 675–681, 2009.

[23] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *International Conference on Computer Aided Verification*, pages 398–413, 2009.

[24] M. Kamkar, P. Fritzson, and N. Shahmehri. Three approaches to interprocedural dynamic slicing. *Microprocessing and Microprogramming*, 38(1):625–636, 1993.

[25] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[26] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[27] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455, 2007.

[28] Non-blocking data structures. URL: https://code.google.com/p/nbds/.

[29] Thread-caching malloc implementation. URL: http://www.nedprod.com/programs/portable/nedmalloc/.

[30] R. H. B. Netzer and M. H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–325, 1994.

[31] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, 2009.

[32] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *International Conference on Computer Aided Verification*, pages 377–390, 1994.

[33] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 37–46, 2010.

[34] 2013 software verification competition. URL: http://sv-comp.sosy-lab.org/2013/.

[35] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: An empirical study. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 15–28, 2014.

[36] A. Valmari. Stubborn sets for reduced state space generation. In *International Conference on Applications and Theory of Petri Nets*, pages 491–515, 1991.

[37] C. Wang, S. Chaudhuri, A. Gupta, and Y. Yang. Symbolic pruning of concurrent program executions. In *ACM*

SIGSOFT Symposium on Foundations of Software Engineering, pages 23–32, 2009.

[38] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.

[39] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 126–140, 2008.

[40] C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 382–396, 2008.

[41] M. Weiser. Program slicing. In *International Conference on Software Engineering*, pages 439–449, 1981.

[42] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *International SPIN workshop on Model Checking Software*, pages 288–305, 2008.

[43] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *International SPIN workshop on Model Checking Software*, pages 279–295, 2009. LNCS 5578.

[44] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 485–502, 2012.

[45] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *International Conference on Software Engineering*, pages 319–329, May 2003.