Exploiting Schedule Insensitivity for Enhancing Concurrent Program Verification

Vineet Kahlon, NEC Labs, Princeton, USA.

Abstract. The key to making program analysis practical for large concurrent programs is to isolate a small set of interleavings to be explored without losing precision of the analysis at hand. The state-of-the-art in restricting the set of interleavings while guaranteeing soundness is partial order reduction (POR). The main idea behind POR is to partition all interleavings of the given program into equivalence classes based on the partial orders they induce on shared objects. Then for each partial order only one interleaving need be explored. POR classifies two interleavings as non-equivalent if executing them leads to different values of shared variables. However, some of the most common properties about concurrent programs like detection of data races, deadlocks and atomicity as well as assertion violations reduce to control state reachability. We exploit the key observation that even though different interleavings may lead to different values of program variables, they may induce the same control behavior. Hence these interleavings, which induce different partial orders, can in fact be treated as being equivalent. Since in most concurrent programs threads are *loosely coupled*, i.e., the values of shared variables typically flow into a small number of conditional statements of threads, we show that classifying interleavings based on the control behaviors rather than the partial orders they induce, drastically reduces the number of interleavings that need be explored. In order to exploit this loose coupling we leverage the use of dataflow analysis for concurrent programs, specifically numerical domains. This, in turn, greatly enhances the scalability of concurrent program analysis.

1 Introduction

Verification of concurrent programs is a hard problem. A key reason for this is the behavioral complexity resulting from the large number of interleavings of transitions of different threads. While there is a substantial body of work devoted to addressing the resulting state explosion problem, a weakness of existing techniques is that they do not fully exploit structural patterns in real-life concurrent code. Indeed, in a typical concurrent program threads are *loosely coupled* in that there is limited interaction between values of shared objects and control flow in threads. For instance, data values written to or read from a shared file typically do not flow into conditional statements in the file system code. What conditional statements may track, for instance, are values of status bits for various files, e.g., whether a file is currently being accessed, etc. However, such status bits affect control flow in very limited and simplistic ways.

One of the main reasons why programmers opt for limited interaction between shared data and control in threads is the fundamental fact that concurrency is complex. A deep interaction between shared data and control would greatly complicate the debugging process. Secondly, the most common goal when creating concurrent programs is to exploit parallelism. Allowing shared data values to flow into conditional statements would require extensive use of synchronization primitives like locks to prevent errors like data races thereby killing parallelism and adversely affecting program efficiency.

An important consequence of this loose coupling of threads is that even though different interleavings of threads may results in different values of shared variables, they may not induce different program behaviors in that the control paths executed may remain unchanged. Moreover, for commonly occurring correctness properties like absence of data races, deadlocks and atomicity violations, we are interested only in the control behavior of concurrent programs. Indeed, data race detection in concurrent programs reduces to deciding the temporal property $\mathsf{EF}(c_1 \wedge c_2)$, where c_1 and c_2 are control locations in two different threads where the same shared variable is accessed and disjoint set of locks are held. Similarly, checking an assertion violation involving an expression *expr* over control locations as well as program variables, can be reduced to control state reachability of a special location *loc* resulting via introduction of a program statement of the form `if(expr) GOTO loc;` . Thus structural patterns in real-life programs as well as in commonly occurring properties are best exploited via reduction techniques that preserve control behaviors of programs rather than the actual behavior defined in terms of program states.

The state-of-the-art in state space reduction for concurrent program analysis is Partial Order Reduction (POR) [3, 8, 9]. The main idea behind POR is to partition all interleavings of the given program into equivalence classes based on the partial orders they induce on shared objects. Then for each partial order only one interleaving need be explored. However, a key observation that we exploit is that because of loose coupling of threads even if different interleavings result in different values of shared (and local) variables, they may not induce different control behaviors. In order to capture how different interleavings may lead to different program behaviors, we introduce the notion of *schedule sensitive* transitions. Intuitively, we say that dependent transitions t and t' are schedule sensitive if executing them in different relative orders affects the behavior of the concurrent program, i.e., changes the valuation of some conditional statement that is dependent on t and t'. POR would explore both relative orders of t and t' irrespective of whether they induce different control behaviors or not whereas our new technique explores different relative orders of t and t' only if they induce different control behaviors. In other words, POR classifies interleavings with respect to global states, i.e., control locations *as well as* the values of program variables, as opposed to just control behavior. However, classifying computations based solely on control behaviors raises the level of abstraction at which partial orders are defined which results in the collapse of several different (state defined) partial orders, i.e., those inducing the same control behavior. This can result in drastic state space reduction.

The key challenge in exploiting the above observations for state space reduction is that deducing schedule insensitivity requires us to reason about program semantics, i.e., whether different interleavings could affect valuations of conditional statements. In order to carry out these checks statically, precisely and in a tractable fashion we leverage the use of dataflow flow analysis for concurrent programs. We show that schedule insensitivity can be deduced in a scalable fashion via the use of numerical invariants like ranges, octagons and polyhedra [7, 2]. Then by exploiting the semantic notion of schedule insensitivity we show that we can drastically reduce the set of interleavings that need be explored over and above POR.

2 Motivation

Consider a concurrent program $\mathcal{P}$ comprised of the two threads T_1 and T_2 shown in fig 1(a) accessing shared variable *sh*. Suppose that we are interested in the reachability

of the global control state (a_4, b_4). Since all transitions write to the same shared variable, i.e., *sh*, each of the transitions a_1, a_2 and a_3 is dependent with each of b_1, b_2 and b_3 except for the pair (a_3, b_3) both of which are read operations. As a result, in applying POR we would need to explore all interleavings of the local transitions of the two threads except a_3 and b_3. This results in the transition digram shown in fig. 1(b) where a pair of the form (c_1, c_2) indicates that thread T_i is at location c_i but hasn't executed the statement at c_i. A downward arrow to the left (right) signifies a move by T_1 (T_2).

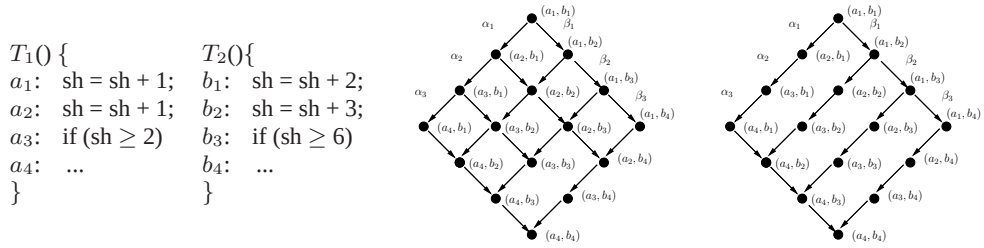

Fig. 1. (a) (b) (c)

However, if we track the values of the shared variable *sh* (assuming it was initialized to 0), we see that at global states (a_3, b_1), (a_3, b_2), (a_3, b_3) and (a_3, b_4), $sh \geq 2$ as a result of which the if-condition at location a_3 of T_1 always evaluates to *true*. This leads to the key observation that even though the statements a_i and b_j, where $i \neq 3$ and $j \neq 3$, are dependent and executing them in different order results in different values of *sh*, it does not affect the valuation of the conditional statement at a_3. Thus with respect to a_3 we need not explore different interleavings of the operations of T_1 and T_2. In fact it suffices to explore just one interleaving, i.e., a_1, a_2, a_3, b_1, b_2 and b_3. Consider now the conditional statement *sh* $\geq$ *6* at b_3. The guard evaluates to *false* in state (a_1, b_3) but evaluates to *true* in each of the states (a_2, b_3) and (a_3, b_3). Starting from state (a_1, b_1), we see that we can reach the global state (a_2, b_3) where *sh* $\geq$ *6* and the state (a_1, b_3) where *sh* $<$ *6*. Thus at (a_1, b_1), we need to explore paths starting with the transition $(a_1, b_1) \rightarrow (a_2, b_1)$ as well as those starting with $(a_1, b_1) \rightarrow (a_1, b_2)$. This is because executing one of these transitions may result in the conditional statement b_3 evaluating to *true* and executing the other may result in it evaluating to *false*. Similarly, from state (a_1, b_2) we need to explore paths starting via both its out-going transitions.

On reaching state (a_2, b_1), however, we see that all interleavings lead either to (a_2, b_3) or to (a_3, b_3) and at both of these states $sh \geq 6$, i.e., the conditional statement at b_3 evaluates to *true*. In other words, starting at state (a_2, b_1) the precise interleaving that is executed does not matter with respect to the valuation of b_3. We would therefore like to explore just one of these interleavings leading to (a_4, b_4). Hence starting at global state (a_2, b_1) we explore just one successor. We choose to explore the one resulting from the transition fired by T_1. Using a similar reasoning, we can see that it suffices to allow only T_1 to execute in each of the states (a_2, b_2) and (a_3, b_2). Furthermore, at the states (a_4, b_1), (a_4, b_2) we have no choice but to execute T_2. Similarly, at the states (a_1, b_4) and (a_2, b_4) we have no choice but to execute T_1. This leads to the transition graph shown in fig. 1(c) clearly demonstrating the reduction (as compared to fig. 1(b)) in the set of interleavings that need be explored.

In order to exploit the above observations, we need to determine for each state (a_i, b_j) in the transaction graph and each conditional statement *con* reachable from (a_i, b_j), whether *con* either evaluates to *true* along all interleavings starting at (a_i, b_j) or evaluates to *false* along all such interleavings. In general, this is an undecidable problem. On the other hand, in order for our technique to be successful our method needs to be scalable to real-life programs. Dataflow analysis is ideally suited for this purpose. Indeed, in our example if we carry out range analysis, i.e., track the possible range of values that *sh* can take, we can deduce that at the locations (a_3, b_1), (a_3, b_2) and (a_3, b_3), *sh* lies in the ranges $[2, 2]$, $[4, 4]$ and $[7, 7]$, respectively. From this it follows easily that the conditional statement at a_3 always evaluates to *true*. It has recently been demonstrated that not only ranges but even more powerful numerical invariants like octagons [7] and polyhedra [2] can be computed efficiently for concurrent programs all of which can be leveraged to deduce schedule insensitivity. A key point is that exploiting numerical invariants to falsify or validate conditional statements offers a good trade-off between precision and scalability. This allows us to filter out interleavings efficiently which can, in turn, be leveraged to make model checking more tractable.

3 System Model

We consider concurrent systems comprised of a finite number of processes or threads where each thread is a deterministic sequential program written in a language such as C. Threads interact with each other using communication/synchronization objects like shared variables, locks and semaphores.

Formally, we define a concurrent program $\mathcal{CP}$ as a tuple $(\mathcal{T}, \mathcal{V}, \mathcal{R}, s_0)$, where $\mathcal{T} = \{T_1, ..., T_n\}$ denotes a finite set of threads, $\mathcal{V} = \{v_1, ..., v_m\}$ a finite set of shared variables and synchronization objects with v_i taking on values from the set V_i, $\mathcal{R}$ the transition relation and s_0 the initial state of $\mathcal{CP}$. Each thread T_i is represented by the control flow graph of the sequential program it executes, and is denoted by the pair (C_i, R_i), where C_i denotes the set of control locations of T_i and R_i its transition relation. A global state s of $\mathcal{CP}$ is a tuple $(s[1], ..., s[n], v[1], ..., v[m]) \in \mathcal{S} = C_1 \times ... \times C_n \times V_1 \times ... \times V_m$, where $s[i]$ represents the current control location of thread T_i and $v[j]$ the current value of variable v_j. The global state transition digram of $\mathcal{CP}$ is defined to be the standard interleaved composition of the transition diagrams of the individual threads. Thus each global transition of $\mathcal{CP}$ results by firing a local transition t of the form (a_i, g, u, b_i), where a_i and b_i are control locations of some thread $T_i = (C_i, R_i)$ with $(a_i, b_i) \in R_i$; g is a guard which is a Boolean-valued expression on the values of local variables of T_i and global variables in $\mathcal{V}$; and u is a set of operations on the set of shared and local variables of T_i that encodes how the value of these variables are modified. Formally, an operation op on variable v is a partial function of the form $IN \times V \rightarrow OUT \times V$, where V is the set of possible values of v and IN and OUT are, respectively, the set of possible input and output values of the operation. The notation $op(in, v_1) \rightarrow (out, v_2)$ denotes execution of operation *op* of v with input value in yielding output *out* while changing the value of v from v_1 to v_2. Given a transition (a_i, g, u, b_i), the set of operations appearing in g and u are said to be *used* by t and are denoted by $used(t)$. Also, for transition $t : (a_i, g, u, b_i)$, we use $pre(t)$ and $post(t)$ to denote control locations a_i and b_j, respectively. A transition $t = (a_i, g, u, b_i)$ of thread T_i is enabled in state s iff $s[i] = a_i$ and guard g evaluates to true in s. If $s[i] = a_i$ but g need not be true in s, then we simply say that t is *scheduled* in s. We write

$s \xrightarrow{t} s'$ to mean that the execution of t leads from state s to s'. Given a transition $t \in \mathcal{T}$, we use $proc(t)$ to denote the process executing t. Finally, we note that each concurrent program $\mathcal{CP}$ with a global state space $\mathcal{S}$ defines the global transition system $A_G = (\mathcal{S}, \Delta, s_0)$, where $\Delta \subseteq \mathcal{S} \times \mathcal{S}$ is the *transition relation* defined by $(s, s') \in \Delta$ iff $\exists t \in \mathcal{T} : s \xrightarrow{t} s'$; and s_0 is the initial state of $\mathcal{CP}$. For ease of exposition, in this paper we consider concurrent programs with only two threads although our techniques extend easily to programs with multiple threads.

4 Schedule Insensitivity Reduction

The state-of-the-art in state space reduction for concurrent program analysis is Partial Order Reduction (POR) [3, 8, 9]. POR classifies computations based solely on the partial orders they induce. These partial orders are defined with respect to global states, i.e., control locations *as well as* the values of program variables, as opposed to just control behavior. However, classifying computations based solely on control behavior raises the level of abstraction at which partial orders are defined which results in the collapse of several different (state defined) partial orders, i.e., those inducing the same control behavior. Whereas (ideally) POR would explore only one computation per partial order, the goal of our new reduction is to explore only one computation for all these collapsed partial orders. This can result in drastic state space reduction.

Concurrent Def-Use Chains and Control Dependency. Control flow within a thread is governed by valuations of conditional statements. However, executing thread transitions accessing shared objects in different orders may result in different values of these shared objects resulting in different valuations of conditional statements of threads and hence different control paths being executed. Note that the valuation of a conditional statement *cond* will be so affected only if the value of a shared variable *flows* into *cond*. This dependency is captured using the standard notion of a *def-use chain*. A *definition* of a variable v is taken to mean an assignment (either syntactic or semantic, e.g., via a pointer) to v. A *definition-use chain (def-use chain)* consists of a definition of a variable in a thread T and all the uses, i.e., read accesses, reachable from that definition in (a possibly different) thread T' without any other intervening definitions. Note that due to the presence of shared variables a def-use chain may, depending on the scheduling of thread operations, span multiple threads. Thus different interleavings can affect the valuation of a conditional statement *cond* only if there is a def-use chain starting from an operation writing to a shared variable sh and leading to *cond*. This is formalized using the notion of *control dependency*.

Definition. (Control Dependency). *We say that a conditional statement cond at location loc of thread T is control dependent on an assignment statement st of thread T' (possibly different from T) if there exists a computation x of the given concurrent program leading a global state with T at location loc such that there is a def-use chain from st to cond along x.*

Schedule Insensitivity. In order to capture how different interleavings may lead to different program behaviors, we introduce the notion of schedule sensitive (or equivalently schedule insensitive) transitions. Intuitively, we say that transitions t and t' of two different threads are *schedule sensitive* if executing them in different relative orders affects the behavior of the concurrent program, i.e., changes the valuation of some conditional statement that is control dependent on t and t'. Formally,

Definition (Schedule Sensitive Operations). *Let OP be the set of operations on variable var and V the set of possible values of var. Then* $\mathsf{Sen} \subseteq OP \times OP \times \mathcal{S}$ *is a schedule sensitivity relation for var in state $s \in \mathcal{S}$ if for all operations op_1, $op_2 \in OP$, $(op_1, op_2, s) \notin \mathsf{Sen}$ (op_1 and op_2 are schedule insensitive for v) implies that $(op_2, op_1, s) \notin \mathsf{Sen}$, where v is the value of var in s, and the following holds for all possible inputs in_1 and in_2:*

- *if $op_1(in_1, v)$ is defined and $op_1(in_1, v) \rightarrow (out_1, v_1')$, then $op_2(in_2, v)$ is defined if and only if $op_2(in_2, v_1')$ is defined; and*
- *if $op_1(in_1, v)$ and $op_2(in_2, v)$ are defined, then each conditional statement cond that is control dependent on op_1 or op_2 and is scheduled in state $t \in \mathcal{S}$ either evaluates to true along all paths of the given concurrent program leading from s to t or it evaluates to false along all such paths.*

Definition (Schedule Insensitive Transitions). *Two transitions t_1 and t_2 are schedule insensitive in state s if*

- *the threads executing t_1 and t_2 are different, and*
- *either t_1 and t_2 are independent in s, or for all $op_1 \in used(t_1)$ and $op_2 \in used(t_2)$, if op_1 and op_2 are operations on the same shared object then op_1 and op_2 are schedule insensitive in s.*

In the above definition we use the standard notion of (in)dependence of transitions as used in the theory of partial order reduction (see [3]). The motivation behind defining schedule insensitive transitions is that if in a global state s, transitions t_1 and t_2 of threads T_1 and T_2, respectively, are dependent then we need to consider interleavings where t_1 and t_2 are executed in different relative orders only if there exists a conditional statement *cond* such that *cond* is control dependent on both t_1 and t_2 and its valuation is affected by executing t_1 and t_2 in different relative orders, i.e., t_1 and t_2 are schedule sensitive with respect to s.

We next define the notion of *control equivalent* computations which is the analogue of Mazurkiewicz equivalent computations for schedule sensitive transitions.

Definition (Control Equivalent Computations). *Two computations x and y are said to be control equivalent if x can be obtained from y by repeatedly permuting adjacent pairs of schedule insensitive transitions, and vice versa.*

Note that control equivalence is a coarser notion of equivalence than Mazurkiewicz equivalence in that Mazurkiewicz equivalence implies control equivalence but the reverse need not be true. That is precisely what we need for more effective state space reduction than POR.

5 Deducing Schedule Insensitivity

In order to exploit schedule insensitivity for state space reduction we need to provide an effective, i.e., automatic and lightweight, procedure for deciding schedule insensitivity of a pair of transitions. By definition, in order to infer whether t_1 and t_2 are schedule sensitive, we have to check whether there exists a conditional statement *cond* satisfying the following: (i) **Control Dependence:** of *cond* on t_1 and t_2, (ii) **Reachability:** *cond*

is enabled in a state t reachable from s, and (iii) **Schedule Sensitivity:** there exist interleavings from s leading to states with different valuations of *cond*.

In order to carry out these checks statically, precisely and in a tractable fashion we leverage the use of dataflow flow analysis for concurrent programs. As was shown in the motivation section, by using range analysis, we were able to deduce schedule insensitivity of the local states (a_i, b_j), where $i \in [2..3]$ and $j \in [1..3]$ which enabled us to explore only one transition from each of them. We can, in fact, leverage even more powerful numerical invariants like octagons [7] and polyhedra [2].

Transaction Graph. In order to deduce control dependence, reachability and schedule sensitivity, we exploit the notion of a *transaction graph* which has previously been used for dataflow analysis of concurrent programs (see [4]). The main motivation behind the notion of a transaction graph is to capture thread interference, i.e., how threads could affect dataflow facts at each others locations. This is because, in practice, concurrent programs usually do not allow unrestricted interleavings of local operations of threads. Typically, synchronization primitives like locks and Java-style wait/notifies, are used in order to control accesses to shared data or introduce causality constraints. Additionally, the values of shared variables may affect valuations of conditional statements which, in turn, may restrict the allowed set of interleavings. The allowed set of interleavings in a concurrent program are determined by control locations in threads where context switches occur. In order to identify these locations the technique presented in [4] delineates *transactions*. A transaction of a thread is a maximal atomically executable piece of code, where a sequence of consecutive statements in a given thread T are *atomically executable* if executing them without any context switch does not affect the outcome of the dataflow analysis at hand. Once transactions have been delineated, the thread locations where context switches need to happen can be identified as the start and end points of transactions. The transactions of a concurrent program are encoded in the form of a *transaction graph* the definition of which is recalled below.

Definition (Transaction graph) [4] *Let $\mathcal{CP}$ be a concurrent program comprised of threads T_1,..., T_n and let C_i and R_i be the set of control locations and transitions of the CFG of T_i, respectively. A transaction graph $\Pi_{\mathcal{CP}}$ of $\mathcal{CP}$ is defined as $\Pi_{\mathcal{CP}} = (C_{\mathcal{CP}}, R_{\mathcal{CP}})$, where $C_{\mathcal{CP}} \subseteq C_1 \times ... \times C_n$ and $R_{\mathcal{CP}} \subseteq (C_1, ..., C_n) \times (C_1, ..., C_n)$. Each edge of $\Pi_{\mathcal{CP}}$ represents the execution of a transaction by a thread T_i, say, and is of the form $(l_1, \ldots, l_i, \ldots, l_n) \rightarrow (m_1, \ldots, m_i, \ldots, m_n)$ where (a) starting at the global state $(l_1, ..., l_n)$, there is an atomically executable sequence of statements of T_i from l_i to m_i, and (b) for all $j \neq i$, $l_j = m_j$.*

Note that this definition of transactions is quite general, and allows transactions to be inter-procedural, i.e., begin and end in different procedures, or even begin and end inside loops. Also, transactions are not only program but also analysis dependent.

Our use of transaction graphs for deducing schedule insensitivity, is motivated by several reasons. First, transaction graphs allow us to carry out dataflow analysis for the concurrent program at hand which is crucial in reasoning about schedule insensitivity. Secondly, transaction graphs already encode reachability information gotten by exploiting scheduling constraints imposed by both synchronization primitives as well as shared variables. Finally, the transaction graph encodes concurrent def-use chains which we use in inferring control dependency. In other words, transaction graphs encodes all the necessary information that allows us to readily decide schedule sensitivity.

Transaction graph Construction. We now recall the transaction graph construction [4] which is an iterative refinement procedure that goes hand-in-hand with the computation of numerical invariants (steps 1-9 of alg. 1). In other words, the transaction graph construction and computation of numerical invariants are carried out simultaneously via the same procedure.

First, an initial set of (coarse) transactions are identified by using scheduling constraints imposed by synchronization primitives like locks and wait/notify and ignoring the effects of shared variables (step 3-7 of alg. 1). This step is essentially classical POR carried out over the product of the control flow graphs of the given threads. This initial synchronization-based transaction delineation acts as a bootstrapping step for the entire transaction delineation process. These transactions are used to compute the initial set of numerical (ranges/octagonal/polyhedral) invariants. Note that once a (possibly coarse) transaction graph is generated dataflow analysis can be carried out exactly as for sequential programs. However, based on these sound invariants, it may be possible to falsify conditional statements that enable us to prune away unreachable parts of the program (Step 8) (see [4] for examples). We use this sliced program, to re-compute (via steps 3-7) transactions based on synchronization constraints which may yield larger transactions. This, in turn, may lead to sharper invariants (step 8). The process of progressively refining transactions by leveraging synchronization constraints and sound invariants in a dovetailed fashion continues till we reach a fix-point.

Deducing Schedule Insensitivity. The transaction graph as constructed via the algorithm described in [4] encodes transactions or context switch points as delineated via a refinement loop that dovetails classical POR and slicing induced by numerical invariants. In order to incorporate the effects of schedule insensitivity we refine this transaction delineation procedure to avoid context switches induced by pairs of transitions of different threads that are dependent yet schedule insensitive.

The procedure for schedule insensitive transaction graph construction is formalized as alg. 1. Steps 1-9 of alg. 1 are from the original transaction delineation procedure given in [4]. In order to collapse partial orders by exploiting schedule insensitivity, we introduce the additional steps 10-32. We observe that given a state (l_1, l_2) of the transaction graph, a context switch is required at location l_1 of thread T_1 if there exists a global state (l_1, m_2) reachable from (l_1, l_2) such that l_1 and m_2 are schedule sensitive. This is because executing l_1 and m_2 in different orders may lead to different program behaviors. Since a precise computation of the schedule sensitivity relation is as hard as the verification problem, in order to determine schedule insensitivity of (l_1, m_2), we use a static over-approximation of the schedule sensitivity relation defined as follows:

Definition (Static Schedule Sensitivity). *Transitions t_1 and t_2 scheduled at control locations n_1 and n_2 of threads T_1 and T_2, respectively, are schedule insensitive at state (n_1, n_2) of the transaction graph if for each conditional statement* cond *such that*

- cond *is reachable from (n_1, n_2) in the transaction graph (***Reachability***),*
- *there are concurrent def-use chains in the transaction graph from both n_1 and n_2 to* cond *(***Control Dependence***),*
- cond *either evaluates to* true *along all paths of the transaction graph from (n_1, n_2) to* cond *or it evaluates to* false *along all such paths (***Schedule Insensitivity***).*

Using dataflow analysis, these checks can be carried out in a scalable fashion.

Algorithm 1 Construction of Schedule Insensitive Transaction Graph

1: **repeat**
2: Initialize $W = \{(in_1, in_2)\}$, where in_j is the initial state of thread T_j.
3: **repeat**
4: Remove a state (l_1, l_2) from W and add it to $Processed$
5: Compute the set *Succ* of successors of (l_1, l_2) via POR by exploiting synchronization constraints **(Synchronization Constraints)**
6: Add all states of $Succ$ not in $Processed$ to W.
7: **until** W is empty
8: Compute numerical invariants on the resulting synchronization skeleton to slice away unreachable parts of the program **(Shared Variable Constraints)**
9: **until** transactions cannot be refined further
10: **repeat**
11: **for** each state (l_1, l_2) of Π **do**
12: $control\_oblivious = true$
13: **for** each global state (l_1, m_2) where m_2 is dependent with l_1 **do**
14: **for** each conditional state *cond* scheduled at state (r_1, r_2), say, **do**
15: **if** (r_1, r_2) is reachable from (l_1, m_2) **then**
16: **if** *cond* is control dependent with l_1 and m_2 **then**
17: **if** $inv_{(r_1,r_2)}$ is the invariant at location (r_1, r_2) and $\neg((inv_{(r_1,r_2)} \Rightarrow cond) \vee (inv_{(r_1,r_2)} \wedge cond = false))$ **then**
18: $control\_oblivious = false$
19: **end if**
20: **end if**
21: **end if**
22: **end for**
23: **end for**
24: **if** $control\_oblivious$ **then**
25: **for** each predecessor (k_1, l_2) of (l_1, l_2) in Π **do**
26: **for** each successor (n_1, l_2) of (l_1, l_2) in Π **do**
27: remove (l_1, l_2) as a successor of (k_1, l_2) and add (n_1, l_2) as a successor.
28: **end for**
29: **end for**
30: **end if**
31: **end for**
32: **until** no more states can be sliced

Checking Reachability and Control Dependency. For our reduction to be precise it is important that while inferring schedule insensitivity we only consider conditional statements *cond* that are reachable from (l_1, m_2). As discussed before, reachability of global states is governed both by synchronization primitives and shared variable values and by using numerical invariants we can infer (un)reachability efficiently and with high precision. Importantly, this reachability information is already encoded in the transition relation of the transaction graph. In order to check control dependence of *cond* on l_1 and m_2, we need to check whether there are def-use chains from a shared variable v written to at locations l_1 and m_2 to a variable u accessed in the conditional statement *cond* at location r_1 or r_2, where state (r_1, r_2) of the transaction graph is reachable from (l_1, l_2). Note that all states that have been deduced as unreachable via the use of numerical invariants and synchronization constraints have already been sliced away via step 8 of

alg. 1. Thus it suffices to track def-use chains along the remaining paths (step 14) in the transaction graph starting at (l_1, l_2) (step 15). This can be accomplished in exactly the same way as in sequential programs - the only difference being that we do it along paths in the transaction graph so that def-use chains can span multiple threads.

Checking Schedule Insensitivity. Next, in order to deduce that a conditional statement *cond* scheduled in state (r_1, r_2) either evaluates to *true* along all paths from (l_1, m_2) to (r_1, r_2) or evaluates to false along all such paths, we leverage numerical invariants computed in step 8 of alg. 1. Let $inv_{(r_1,r_2)}$ be the (range, octagonal, polyhedral) invariant computed at (r_1, r_2). Then if *cond* is either falsified, i.e., *cond*$\wedge inv_{(r_1,r_2)} =$ *false* or *cond* is validated, i.e., $inv_{(r_1,r_2)} \Rightarrow$ *cond*, the valuation of conditional statements in (r_1, r_2) are independent of the path from (l_1, m_2) to (r_1, r_2) (step 16). In order to check schedule-insensitivity of (l_1, m_2), we need to carry out the above check for every conditional statement that is reachable from (l_1, m_2) and has a def-use chain from both l_1 and m_2 to *cond*. If all there exists no such conditional statement then we can avoid a context switch at location l_1 of thread T_1 (steps 24-30) thereby collapsing partial orders in the transaction graph.

Scalability Issues. A key concern in using transactions graphs for deducing schedule insensitivity is the state explosion resulting from the product construction. However, in practice, the transaction graph construction is very efficient due to three main reasons. First, in building the transaction graph we take the product over control locations and not local states of threads. Thus for k threads the size of the transaction graph is at most n^k, where n is the maximum number of lines of code in any thread. Secondly, when computing numerical invariants we use the standard technique of variable clustering wherein two variables u and v occur in a common cluster if there exists a def-use chain along which both u and v occur. Then it suffices to build the transaction graph for each cluster separately. Moreover, for clusters that contains only local thread variables there is no need to build the transaction graph as such variables do not produce thread dependencies. Thus cluster induced slicing can drastically cut down on the statements that need to be considered for each cluster and, as a result, the transaction graph size. Finally, since each cluster typically has few shared variables, POR (step 5) further ensures that the size of the transaction graph for each cluster is small. Finally, it is worth keeping in mind that the end goal of schedule insensitivity reduction is to help model checking scale better and in this context any transaction graph construction will likely be orders of magnitude faster than model checking which remains the key bottleneck.

6 Enhancing Symbolic Model Checking via Schedule Insensitivity

We show how to exploit schedule insensitivity for scaling symbolic model checking.

Schedule Insensitivity versus Partial Order Reduction. In order to illustrate the advantage of schedule insensitivity reduction we start by briefly recalling monotonic partial order reduction, a provably optimal symbolic partial order reduction technique. The technique is optimal in that it ensures that exactly one interleaving is explored for every partial order induced by computations of the given program. Using schedule insensitivity we show how to enhance monotonic POR by further collapsing partial orders over and above those obtained via MPOR.

The intuition behind MPOR is that if all transitions enabled at a global state are independent then we need to explore just one interleaving. This interleaving is chosen to be the one in which transitions are executed in increasing (monotonic) order of their thread-ids. If, however, some of the transitions enabled at a global state are dependent than we need to explore interleavings that exercise both relative orders of these transitions which may violate the *natural* monotonic order. In that case, we allow an out-of-order-execution, viz., a transition tr' with larger thread-id than tr and dependent with tr to execute before tr.

Example. Consider the example in fig. 1. If we ignore dependencies between local transitions of threads T_1 and T_2 then MPOR would explore only one interleaving namely the one wherein all transitions of T_1 are executed before all transitions of T_2, i.e., the interleaving $\alpha_1\alpha_2\alpha_3\beta_1\beta_2\beta_3$ (see fig. 1(b)). Consider now the pair of dependent operations (a_1, b_1) accessing the same shared variable sh. We need to explore interleavings wherein a_1 is executed before b_1, and vice versa, which causes, for example, the out-of-order execution $\beta_1\alpha_1\alpha_2\alpha_3\beta_2\beta_3$ where transition β_1 of thread T_2 is executed before transition α_1 of thread T_1 even though the thread-id of β_1 is greater than the thread-id of α_1. MPOR guarantees that exactly one interleaving is explored for each partial order generated by dependent transitions.

When exploiting schedule insensitivity, starting at a global control state (c_1, c_2) an out-of-order execution involving transitions tr_1 and tr_2 of thread T_1 and T_2, respectively, is enforced only when (i) tr_1 and tr_2 are dependent, *and* (ii) tr_1 and tr_2 are schedule independent starting at (c_1, c_2). Note that the extra condition (ii) makes the criterion for out-of-order execution stricter. This causes fewer out-of-order executions and further restricts the set of partial orders that will be explored over and above MPOR.

Going back to our example, we see that starting at global control state (a_2, b_2), transitions a_2 and b_2 are dependent as they access the same shared variable. Thus MPOR would explore interleavings wherein a_2 is executed before b_2 ($\alpha_1\beta_1\alpha_2\alpha_3\beta_2\beta_3$) and vice versa ($\alpha_1\beta_1\beta_2\alpha_2\alpha_3\beta_3$). However as shown in sec. 2, a_2 and b_2 are schedule insensitive and so executing a_2 and b_2 in different relative orders does not generate any new behavior. Thus we only explore one of these orders, i.e., a_2 executing before b_2 as *thread-id*$(a_2) = 1 < 2 =$*thread-id*(b_2). Thus after applying SIR, we see that starting at (a_2, b_2) only one interleaving, i.e., $\alpha_2\alpha_3\beta_2\beta_3$, is explored.

Implementation Strategy. Our strategy for implementing SIR is as follows:

1. We start by reviewing the basics of SAT/SMT-based bounded model checking.

2. Next we review the MPOR implementation wherein the scheduler is constrained so that it does not explore all enabled transitions as in the naive approach but only those that lead to the exploration of new partial orders via a monotonic ordering strategy as discussed above.

3. Finally we show how to implement SIR by further restricting the scheduler to explore only those partial orders that are generated by schedule sensitive dependent transitions. This is accomplished via the same strategy as in MPOR - the only difference being that we allow out-of-order executions between transitions that are not just dependent but also schedule sensitive.

Bounded Model Checking (BMC). Given a multi-threaded program and a reachability property, BMC can check the property on all execution paths of the program up to a fixed depth K. For each step $0 \leq k \leq K$, BMC builds a formula Ψ such that *Ψ is satisfiable iff there exists a length-k execution that violates the property*. The formula

is denoted $\Psi = \Phi \wedge \Phi_{prop}$, where Φ represents all possible executions of the program up to k steps and Φ_{prop} is the constraint indicating violation of the property (see [1] for more details about Φ_{prop}). In the following, we focus on the formulation of Φ. Let $V = V_{global} \cup \bigcup V_i$, where V_{global} are global variables and V_i are local variables in T_i. For every local (global) program variable, we add a state variable to V_i (V_{global}). We add a pc_i variable for each thread T_i to represent its current program counter. To model nondeterminism in the scheduler, we add a variable sel whose domain is the set of thread indices $\{1, 2, \ldots, N\}$. A transition in T_i is executed only when $sel = i$.

At every time frame, we add a fresh copy of the set of state variables. Let $v^i \in V^i$ denote the copy of $v \in V$ at the i-th time frame. To represent all possible length-k interleavings, we first encode the transition relations of individual threads and the scheduler, and unfold the composed system exactly k time frames.

$$\Phi := \ I(V^0) \wedge \bigwedge_{i=0}^{k} (SCH(V^i) \wedge \bigwedge_{j=1}^{N} TR_j(V^i, V^{i+1}))$$

where $I(V^0)$ represents the set of initial states, SCH represents the constraint on the scheduler, and TR_j represents the transition relation of thread T_j. Without any reduction, $SCH(V^i) := true$, which means that sel takes all possible values at every step. This default SCH considers all possible interleavings. SIR can be implemented by adding constraints to SCH to remove redundant interleavings.

MPOR Strategy. As discussed before, the broad intuition behind MPOR is to execute location transitions of threads in increasing orders of their thread-ids unless dependencies force an out-of-order execution. In order to characterize situations where we need to force an out-of-order execution we use the notion of a *dependency chain*.

Definition (Dependency Chain) *Let t and t' be transitions such that $t <_x t'$, i.e., t is executed before t' along computation x. A dependency chain along x starting at t is a (sub-)sequence of transitions $tr_{i_0}, ..., tr_{i_k}$ fired along x, where (a) $i_0 < i_1 < ... < i_k$, (b) for each $j \in [0..k-1]$, tr_{i_j} is dependent with $tr_{i_{j+1}}$, and (c) there does not exist a transition fired along x between tr_{i_j} and $tr_{i_{j+1}}$ that is dependent with tr_{i_j}.*

For transitions t and t' fired along x, we use $t \Rightarrow_x t'$ to denote that the fact that there is a dependency chain from t to t' along x. Then the MPOR strategy is as follows:

MPOR Strategy. *Explore only those computation x such that for each pair of transitions tr and tr' such that $tr' <_x tr$ we have $tid(tr') > tid(tr)$ only if either (i) $tr' \Rightarrow_x tr$, or (ii) there exists a transition tr'' such that $tid(tr'') < tid(tr)$, $tr' \Rightarrow_x tr''$ and $tr' <_x tr'' <_x tr$.*

Schedule Insensitivity Reduction. For implementing SIR, we only need to consider partial orders induced by those pairs of conflicting transitions that are schedule sensitive. This motivates the following definition.

Definition (Schedule-Dependency Chain) *Let t and t' be transitions fired along a computation x such that $t <_x t'$. A schedule-dependency chain along x starting at t is a (sub-)sequence of transitions $tr_{i_0}, ..., tr_{i_k}$ fired along x, where (a) $i_0 < i_1 < ... < i_k$, (b) for each $j \in [0..k-1]$, tr_{i_j} is schedule-dependent with $tr_{i_{j+1}}$, and (c) there does not exist a transition fired along x between tr_{i_j} and $tr_{i_{j+1}}$ that is schedule-dependent with tr_{i_j}.*

For transitions t and t' fired along x, we use $t \Rightarrow^s_x t'$ to denote that the fact that there is a schedule-dependency chain from t to t' along x. Note that the difference between the above definition and that of a Dependency chain is that the above definition is more restrictive as it only consider chains over dependent transitions only if they are schedule-dependent. As a result is leads to exploration of fewer partial orders which in turn enhances scalability of state space exploration. Then the SIR strategy is as follows:

SIR. *Explore only those computations such that for each pair of transitions tr and tr' such that $tr' <_x tr$ we have $tid(tr') > tid(tr)$ only if either (i) $tr' \Rightarrow^s_x tr$, or (ii) there exists a transition tr'' such that $tid(tr'') < tid(tr)$, $tr' \Rightarrow^s_x tr''$ and $tr' <_x tr'' <_x tr$.*

Encoding SIR. In order to implement our technique, we need to track schedule dependency chains in a space efficient manner. Our encoding to track schedule dependency chains is similar to the one for tracking dependency chains in MPOR except that we consider schedule sensitivity as opposed to dependency of transitions in building these chains. In order to track schedule dependency chains, for each pair of threads T_i and T_j, we introduce a new variable SDC_{ij} defined as follows.

Definition. $SDC_{il}(k)$ *is* 1 *or* -1 *accordingly as there is a dependency chain or not, respectively, from the last transition executed by T_i to the last transition executed by T_l at or before time step k. If no transition has been executed by T_i up to time step k, $SDC_{il} = 0$.*

Updating SDC_{ij}. If at time step k thread T_i is executing transition tr, then for each thread T_l, we check whether the last transition executed by T_l is schedule sensitive with tr. To track that we introduce the dependency variables DEP_{li} defined below.

Definition. $DEP_{li}(k)$ *is* true *or* false *accordingly as the transition being executed by thread T_i at time step k is dependent with the last transition executed by T_l, or not. Note that $DEP_{ii}(k) = 1$ always holds (due to control conflict).*

For MPOR these dependency variables are enough to track dependency chains. However even if two transitions are dependent they might still be schedule insensitive. To carry out this additional check, we introduce the schedule sensitivity variables

Definition. $SS_{li}(k)$ *is* true *or* false *accordingly as the transition of thread T_i being executed at time step k is schedule sensitive with the last transition executed by T_l, or not. Note that $SS_{ii}(k)$ always holds true.*

We now show how the SDC variables are updated. If $(DEP_{li}(k) = 1) \wedge (SS_{li}(k) = true)$ and if $SDC_{jl}(k-1) = 1$, i.e., there is a schedule dependency chain from the last transition executed by T_j to the last transition executed by T_l, then this schedule dependency chain can be extended to the last transition executed by T_i, i.e., tr. In that case, we set $DC_{ji}(k) = 1$. Also, since we track schedule dependency chains only from the last transition executed by each thread, the schedule dependency chain corresponding to T_i needs to start afresh and so we set $SDC_{ij}(k) = -1$ for all $j \neq i$. To sum up, the updates are as follows.

$SDC_{ii}(k) = 1$	
$SDC_{ij}(k) = -1$	when $j \neq i$
$SDC_{ji}(k) = 0$	when $j \neq i$ and $SDC_{jj}(k-1) = 0$
$SDC_{ji}(k) = \bigvee_{l=1}^{n}(SDC_{jl}(k-1) = 1 \wedge DEP_{li}(k) \wedge SS_{li}(k))$	when $j \neq i$ and $SDC_{jj}(k-1) \neq 0$
$SDC_{pq}(k) = SDC_{pq}(k-1)$	when $p \neq i$ and $q \neq i$

Scheduling Constraint. Next we introduce the scheduling constraints variables S_i, where $S_i(k)$ is *true* or *false* based on whether thread T_i can be scheduled to execute or not, respectively, at time step k in order to ensure quasi-monotonicity. Then we conjoin the following constraint to SCH:

$$\bigwedge_{i=1}^{n}(sel^k = i \Rightarrow S_i(k))$$

We encode $S_i(k)$ (where $1 \leq i \leq n$) as follows:
$S_i(0) = true$ and
for $k > 0$, $S_i(k) = \bigwedge_{j>i}(SDC_{ji}(k) \neq -1 \vee \bigvee_{l<i} SDC_{jl}(k-1) = 1)$

In the above formula, $SDC_{ji}(k) \neq -1$ encodes the condition that either a transition by thread T_j, where $j > i$, hasn't been executed up to time k, i.e., $SDC_{ji}(k) = 0$, or if it has then there is a schedule-dependency chain from the last transition executed by T_j to the transition of T_i enabled at time step k, i.e., $SDC_{ji}(k) = 1$. If these two cases don't hold and there exists a transition tr' fired by T_j before the transition tr of T_i enabled at time step k, then in order for quasi-monotonicity to hold, there must exist a transition *tr"* fired by thread T_l, where $l < i$, after tr' and before tr such that there is a schedule-dependency chain from tr' to tr'' which is encoded as $\bigvee_{l<i} SDC_{jl}(k-1) = 1$.

All we need to show now is how to encode the DEP and SS variables. The dependency variables are encoded exactly as in MPOR (see [5] for details). Thus as a final step we show how to encode the SS variables.

Encoding SS. For encoding SS variables we use the schedule insensitive transaction graph constructed in sec 5. In order to decide whether transitions $c_i \rightarrow d_i$ and $c_j \rightarrow d_j$ of threads T_i and T_j are schedule sensitive it suffices to check whether there exist paths in the transaction graph wherein c_i is executed before c_j along one and vice versa along the other. Note that since SIR allows context switching only at locations where shared variables are accessed, we can restrict ourselves to locations c_i and c_j satisfying this property. Moreover since we are interested only in the schedule (in)sensitivity of dependent transitions we can further assume that the statements at c_i and c_j are dependent.

To encode SS_{ij} we first compute the set *SS-Pairs*$_{ij}$ of all pairs (c_1, c_2) such that (i) c_1 and c_2 belong to threads T_i and T_j, (ii) there exists a pair of dependent transitions of the form $tr_1 : c_1 \rightarrow d_1$ and $tr_2 : c_2 \rightarrow d_2$, and (iii) there exist paths in the schedule insensitive transaction graph wherein c_1 is executed before c_2 along one and vice versa along the other. The sets *SS-Pairs*$_{ij}$ can be enumerated via a single traversal of the transaction graph. Then $SS_{ij} = \bigvee_{(c,d)\in SS-Pairs_{ij}}((pc_i = c) \wedge (pc_j = d))$

7 Implementation and Experimental Results

In previous work [6] we used static analysis to produce data race warnings for a suite of Linux device drivers downloaded from the Linux Kernel Archives. Each warning produced via static analysis is a pair (l_1, l_2) of control locations in different threads where the same shared variable is accessed with at least one of the access being a write operation and disjoint sets of locks are held. In order to decide whether (l_1, l_2) is a true date race we have to decide whether there exists a reachable global state of the given program with thread T_i at control location l_i.

Witness #	Shared Vars	Relevant Sh. Vars	Transaction Graph	MPOR		SIR	
				Time	Mem	Time	Mem
jfs_dmap : 1	6	1	0.01	0.02	59	0.01	12
ctrace : 1	19	12	10	2	62	1	43
ctrace : 2	19	12	14	10 hr	1.2G	3hr	0.5G
ctrace : 3	19	12	12	2303	733	1800	560
autofs : 1	7	2	0.05	1.14	60	0.5	30
autofs : 2	7	2	0.07	128	144	43	85
ptrace : 1	3	1	20	844	249	502	191

Witness #	Shared Vars	Relevant Sh. Vars	Transaction Graph	MPOR		SIR	
				Time	Mem	Time	Mem
raid : 1	6	0	-	26.13	75	7.1	21
raid : 2	6	0	-	179	156	20	41
raid : 3	6	0	-	32.19	87	5	29
raid : 4	6	0	-	4.15	61	3	19
raid : 5	6	0	-	9.30	59	2	24
raid : 6	6	0	-	70	116	12	23
ipoib : 1	10	2	0.02	0.1	58	0.1	58
ipoib : 2	10	2	0.02	0.1	59	0.1	59
ipoib : 3	10	2	0.04	0.1	58	0.1	57
ipoib : 4	10	2	0.03	0.3	59	0.3	59

Table 1. Model Checking Data Race Warnings (Timings are in seconds and memory in MBs).

We compare the time taken and memory used for MPOR [5] and SIR. For each of the six drivers, the property checked is reachability of control locations corresponding to data race warnings. Columns 1 and 2 report the total number and the number of relevant shared variables, respectively. Here a shared variable is said to relevant is there is a def-use chain starting at some write of v and leading to a conditional statement of some thread. Clearly we need to consider conflicts only for the relevant shared variables. Note that typically, the number of relevant shared variables is considerably less than the total number of shared variables thereby pointing to the utility of SIR. Column 3 gives the time taken for transaction graph construction using our new SIR algorithm. Note that the overhead of this step is small. Also, for examples that contain no relevant shared variables, e.g., *raid*, this step is unnecessary as we know a priori that only one interleaving need be explored. The model checking statistics for MPOR and SIR are shown in columns 4-5 and 6-7, respectively. Clearly, both the time taken and memory used when applying SIR is significantly less than when MPOR is used. Our experiments were conducted on a workstation with 2.8 GHz Xeon processor and 4GB memory.

References

1. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.
2. Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among the variables of a program. In *ACM POPL*, pages 84–97, January 1978.
3. P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. LNCS 1032. Springer-Verlag, 1996.
4. V. Kahlon, S. Sankaranarayan, and A. Gupta. Semantic reduction of thread interleavings for concurrent programs. In *TACAS*, 2009.
5. V. Kahlon, C. Wang, and A. Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *CAV*, 2009.
6. V. Kahlon, Y. Yang, S. Sankaranarayan, and A. Gupta. Fast and Accurate Static Data-Race Detection for Concurrent Programs. In *CAV*, 2007.
7. Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO II*, volume 2053, pages 155–172, May 2001.
8. D. Peled. Combining partial order reductions with on-the-fly model checking. In *Formal Aspects of Computing*, volume 8, pages 39–64, 1996.
9. Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4), 1992.