# Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique

Vineet Kahlon, Chao Wang, and Aarti Gupta

NEC Laboratories America
4 Independence Way, Princeton, NJ 08540, USA

**Abstract.** We present a new technique called *Monotonic Partial Order Reduction (MPOR)* that effectively combines dynamic partial order reduction with symbolic state space exploration for model checking concurrent software. Our technique hinges on a new characterization of partial orders defined by computations of a concurrent program in terms of *quasi-monotonic sequences* of thread-ids. This characterization, which is of independent interest, can be used both for explicit or symbolic model checking. For symbolic model checking, MPOR works by adding constraints to allow automatic pruning of redundant interleavings in a SAT/SMT solver based search by restricting the interleavings explored to the set of quasi-monotonic sequences. Quasi-monotonicity guarantees both soundness (all necessary interleavings are explored) and optimality (no redundant interleaving is explored) and is, to the best of our knowledge, the only known optimal symbolic POR technique.

## 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. In explicit-state model checking, partial order reduction (POR) techniques [6, 14, 16] have, therefore, been developed to exploit the equivalence of interleavings of independent transitions in order to reduce the search space. Since computing the precise dependency relation between transitions may be as hard as the verification problem itself, existing POR methods often use a conservative statically computed approximation. Dynamic [5] and Cartesian [9] partial order reduction obviate the need to apply static analysis *a priori* by detecting collisions (data dependencies) on-the-fly. These methods can, in general, achieve better reduction due to more accurate collision detection. However, applying these POR methods (which were designed for explicit state space search) to symbolic model checking is a non-trivial task.

A major strength of symbolic state space exploration methods [2] is that *property dependent* and *data dependent* search space reduction is automatically exploited inside modern SAT or SMT (Satisfiability Modulo Theory) solvers, through the addition of conflict clauses and non-chronological backtracking [15]. Symbolic methods are often more efficient in reasoning about variables with large domains. However, combining classic POR methods (e.g., those based on persistent-sets [7]) with symbolic algorithms has proven to be difficult [1, 3, 8, 10, 12]. The difficulty arises from the fact that symbolic methods implicitly manipulate large *sets of states* as opposed to manipulating

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 398–413, 2009.

states individually. Capturing and exploiting transitions that are dynamically independent with respect to a *set of states* is much harder than for individual states.

Consider the example program from [17] shown in Fig. 1 comprised of two concurrent threads accessing a global array $a[\,]$. It is hard to determine statically whether transitions $t_A, t_B$ in thread $T_1$ are dependent with $t_\alpha, t_\beta$ in $T_2$. Similarly, without knowing the points-to locations of $p$ and $q$, we cannot decide whether $t_C$ and $t_\gamma$ are dependent or not. This renders POR methods relying on a static computation of conflicts non-optimal. Indeed, when $i \neq j$ holds in some executions, $t_A, t_B$ and $t_\alpha, t_\beta$ become independent, meaning that the two sequences $t_A; t_B; t_\alpha; t_\beta; t_C; t_\gamma;$ and $t_\alpha; t_\beta; t_A; t_B; t_C; t_\gamma;$ are equivalent. However, none of the existing symbolic partial order reduction methods [1, 3, 8, 10, 12] takes advantage of such information. Among explicit-state POR methods, dynamic partial order reduction [5] and Cartesian partial order reduction [9] are able to achieve some reduction by detecting conflicts on-the-fly; in any individual state $s$, the values of $i$ and $j$ (as well as $p$ and $q$) are fully determined, allowing us to detect conflicts accurately. However, it is not clear how to directly apply these techniques to symbolic model checking, where conflict detection is performed with respect to a set of states. Missing out on these kind of partial-order reductions can be costly since the symbolic model checker needs to exhaustively search among the reduced set of execution sequences.

```
T1

     i = foo() ;
     ...
A    a[i] = 10 ;
B    a[i] = a[i]+20;
C    *p = a[j] ;
```

```
T2

     j = bar() ;
     ...
α    a[j] = 50 ;
β    a[j] = a[j]+100;
γ    *q = a[i] ;
```

**Fig. 1.** $t_A, t_B$ are independent with $t_\alpha, t_\beta$ when $i \neq j$; $t_C$ is independent with $t_\gamma$ when $(i \neq j) \wedge (p \neq q)$

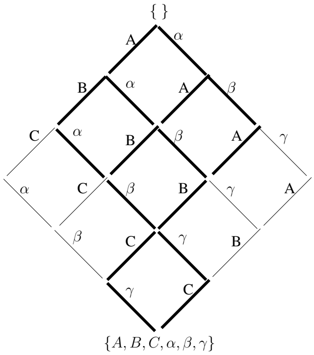


**Fig. 2.** The lattice of interleavings

Recently, a new technique called Peephole Partial Order Reduction (PPOR) [17] has been proposed that allows partial order reduction to be integrated with symbolic state space exploration techniques. The key idea behind PPOR is to place constraints on which processes can be scheduled to execute in the next two steps starting at each global state. If in a global state, transitions $tr$ and $tr'$ such that $tid(tr) < tid(tr')$, where $tid$ denotes thread-id, are enabled and independent then $tr'$ cannot execute immediately before $tr$. It was shown that PPOR is optimal for programs with two threads but non-optimal for programs with more than two. The reason is that in order to achieve optimality for programs with more than two threads, we might need to track dependency chains involving many processes. These chains, which could be spread out over an entire computation, are hard to capture via local scheduling constraints.

We present a new technique called *Monotonic Partial Order Reduction (MPOR)* that exploits a new characterization of partial orders defined by computations of a given concurrent program in terms of *quasi-monotonic sequences* of thread-ids. This characterization, which is of independent interest, can be used both for explicit or symbolic model checking. In this paper, we show that restricting the interleavings explored to the set of quasi-monotonic sequences guarantees both soundness (all necessary interleavings are explored) and optimality (no redundant interleaving is explored). This is accomplished by proving that for each computation there exists a quasi-monotonic sequence that is Mazurkiewicz equivalent[1] [13] to it, and that no two quasi-monotonic sequences can be Mazurkiewicz equivalent. The key intuition behind quasi-monotonicity is that if all transitions enabled at a global state are independent then we need to explore just one interleaving. We choose this interleaving 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 may violate this *natural* monotonic order. In that case, we allow an out-of-order-execution, viz., a transition $tr$ with larger thread-id than that of transition $tr'$ to execute before $tr'$ only if there exists a *dependency chain* from $tr$ to $tr'$, i.e., a sequence of transitions from $tr$ to $tr'$ wherein adjacent transitions are pairwise dependent. Such sequences are called quasi-monotonic.

Note that although our monotonic POR method has the same goal as classic POR methods [6, 14, 16, 5, 9], it does not correspond directly to any existing method. In particular, it is not a symbolic implementation of any of these explicit-state methods. Importantly, our method is optimal for programs with arbitrarily many threads, which, to the best of our knowledge, is not guaranteed by any of the existing symbolic POR techniques [1, 12, 8, 3, 10, 17]. Finally, the proposed encoding scheme is well suited for symbolic search using SAT/SMT solvers.

To summarize, our main contributions are: (1) the notion of quasi-monotonic sequences, which isolates a unique representative for each partial order resulting from the computations of the given program; (2) a new partial order reduction that adds constraints to ensure quasi-monotonicity, along with a symbolic formulation; and (3) the guarantee of removal of all redundant interleavings for programs with an arbitrary number of threads.

## 2 Classical Partial Order Reduction

We start by reviewing standard notions from classical partial order reduction (POR) [11, 7]. Let $T_i$ $(1 \leq i \leq N)$ be a thread with the set $trans_i$ of transitions. Let $trans = \bigcup_{i=1}^{N} trans_i$ be the set of all transitions. Let $V_i$ be the set of local variables of thread $T_i$, and $V_{global}$ the set of global variables of the given concurrent program. For $t_1 \in trans_i$, we denote the thread-id, i.e., $i$, by $tid(t_1)$, and denote the enabling condition by $en_{t_1}$. If $t_1$ is a transition in $T_i$ from control locations $loc_1$ to $loc_2$ and is guarded by $cond$, then $en_{t_1}$ is defined as $(pc_i = loc_1) \wedge cond$. Here $pc_i \in V_i$ is a special variable representing

[1] Intuitively, two computations $x$ and $y$ are said to be Mazurkiewicz equivalent if $x$ can be obtained from $y$ by repeatedly permuting adjacent pairs of independent transitions, and vice versa.

the thread program counter. Let $S$ be the set of global states of the given program. A state $s \in S$ is a valuation of all local and global variables. For two states $s, s' \in S$, $s \xrightarrow{t_1} s'$ denotes a state transition by applying $t_1$, and $s \xrightarrow{t_i \ldots t_j} s'$ denotes a sequence of state transitions.

### 2.1 Independence Relation

Partial-order reduction exploits the fact that computations of concurrent programs are partial orders on operations of threads on communication objects. Thus instead of exploring all interleavings that realize these partial orders it suffices to explore just a few (ideally just one for each partial order). Interleavings which are equivalent, i.e., realize the same partial order, are characterized using the notion of an independence relation over the transitions of threads constituting the given concurrent program.

**Definition 1 (Independence Relation [11, 7]).** *$R \subseteq trans \times trans$ is an independence relation iff for each $\langle t_1, t_2 \rangle \in R$ the following two properties hold for all $s \in S$:*

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 $s'$; and*
2. *if $t_1, t_2$ are enabled in $s$, there is a unique state $s'$ such that $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$.*

In other words, independent transitions can neither disable nor enable each other, and enabled independent transitions commute. As pointed out in [6], this definition has been mainly of semantic use, since it is not practical to check the above two properties for all states to determine which transitions are independent. Instead, traditional collision detection, i.e., identification of dependent transitions, often uses conservative but easy-to-check sufficient conditions. These checks, which are typically carried out statically, over-approximate the collisions leading to exploration of more interleavings than are necessary. Consider, for example, the transitions $t_1{:}a[i] = e_1$ and $t_2{:}a[j] = e_2$. When $i \neq j$, $t_1$ and $t_2$ are independent. However since it is hard to determine statically whether $a[i]$ and $a[j]$ refer to the same array element, $t_1$ and $t_2$ are considered (statically) dependent irrespective of the values of $i$ and $j$. This results in the exploration of more interleavings than are necessary. Such techniques are therefore not guaranteed to be optimal.

In the *conditional* dependence relation [11, 7], which is a refinement of the dependence relation, two transitions are defined as independent with respect to a state $s \in S$ (as opposed to for all states $s \in S$). This extension is geared towards explicit-state model checking, in which persistent sets are computed for individual states. A persistent set at state $s$ is a subset of the enabled transitions that need to be explored from $s$. A transition is added to the persistent set if it may conflict with a future operation of another thread. The main difficulty in persistent set computation lies in detecting future collisions with enough precision due to which these classic definitions of independence are not well suited for symbolic search.

## 3 Optimal Partial Order Reduction

We formulate a new characterization of partial order reduction in terms of quasi monotonic sequences that is easy to incorporate in both explicit and symbolic methods for

```
T1(){                    T2(){                     T3(){
  c1: sh = 1;              c2: sh = sh';             c3: sh' = 2;
}                        }                         }
```

**Fig. 3.** An Example Program

state space search. To motivate our technique, we consider a simple concurrent program $P$ comprised of three threads $T_1, T_2$ and $T_3$ shown in figure 3. Suppose that, to start with $P$ is in the global state $(c_1, c_2, c_3)$ with thread $T_i$ at location $c_i$ (for simplicity, we show only the control locations and not the values of the variables in each global state). Our goal is to add constraints on-the-fly during model checking that restrict the set of interleavings explored in a way such that all necessary interleavings are explored and no two interleavings explored are Mazurkiewicz equivalent. Let $t_i$ denote the program statement at location $c_i$ of thread $T_i$, respectively. In the global state $s = (c_1, c_2, c_3)$, we see that transitions $t_1$ and $t_2$ are dependent as are $t_2$ and $t_3$. However, $t_1$ and $t_3$ are independent with each other. Since $t_1$ and $t_2$ are dependent with each other, we need to explore interleavings wherein $t_1$ is executed before $t_2$, and vice versa.

For convenience, given transitions $t$ and $t'$ executed along a computation $x$ of the given program, we write $t <_x t'$ to denote that $t$ is executed before $t'$ along $x$. Note that the same thread statement (say within a program loop) may be executed multiple times along a computation. Each execution is considered a different transition. Then, using the new notation, we can rephrase the scheduling constraints imposed by dependent transitions as follows: since $t_1$ and $t_2$ are dependent transitions, we need to explore interleavings along which $t_1 < t_2$ and those along which $t_2 < t_1$. Similarly, we need to explore interleavings along which $t_2 < t_3$, and vice versa. However, since $t_1$ and $t_3$ are independent we need to avoid exploring both relative orderings of these transitions wherever possible.

Let the thread-id of transition *tr* executed by thread $T_i$, denoted by $tid(tr)$, be $i$. In general, one would expect that for independent transitions $tr$ and $tr'$ we need not explore interleavings along which $tr < tr'$ as well as those along which $tr' < tr$ and it suffices to pick one relative order, say, $tr < tr'$, where $tid(tr) < tid(tr')$, i.e., force pairwise independent transitions to execute in increasing order of their thread-ids. However, going back to our example, we see that the transitivity of '$<$', might result in ordering constraints on the independent transitions $t_1$ and $t_3$ that force us to explore both relative orderings of the two transitions. Indeed, the ordering constraints $t_3 < t_2$ and $t_2 < t_1$ imply that $t_3 < t_1$. On the other hand, the constraints $t_1 < t_2$ and $t_2 < t_3$ imply that $t_1 < t_3$. Looking at the constraints $t_3 < t_2$ and $t_2 < t_1$ from another perspective, we see that $t_3$ needs to be executed before $t_1$ because there is a sequence of transitions from $t_3$ to $t_1$ (in this case $t_3, t_2, t_1$) wherein adjacent transitions are pairwise dependent. Thus given a pair of independent transitions $tr$ and $tr'$ such that $tid(tr) < tid(tr')$, a modification to the previous strategy would be to explore an interleaving wherein $tr' < tr$ only if there is a sequence of transitions from $tr'$ to $tr$ wherein adjacent transitions are pairwise dependent, i.e., force independent transitions to execute in increasing order of their thread-ids as long as there are no dependency constraints arising from the transitivity of '$<$' that force an *out-of-order* execution.

This strategy, however, might lead to unsatisfiable scheduling constraints. To see that we consider a new example program with a global state $(c_1, c_2, c_3, c_4)$, where for each $i$, local transition $t_i$ of $T_i$ is enabled. Suppose that $t_1$ are $t_4$ dependent only with each other, as are $t_2$ and $t_3$. Consider the set of interleavings satisfying $t_4 < t_1$ and $t_3 < t_2$. Using the facts that (i) $tid(t_1) < tid(t_3)$, and (ii) there cannot be a sequence of transitions leading from $t_3$ to $t_1$ wherein adjacent transitions are pairwise dependent, by the above strategy we would execute $t_1$ before $t_3$ leading to the interleaving $t_4, t_1, t_3, t_2$. However, since $t_2$ and $t_4$ are independent, and there is no sequence of transitions from $t_4$ to $t_2$ wherein adjacent transitions are pairwise dependent, $t_2$ must be executed before $t_4$. This rules out the above interleaving. Using a similar reasoning, one can show that the above strategy will, in fact, rule out all interleavings where $t_4 < t_1$ and $t_3 < t_2$. Essentially, this happens because thread-ids of processes in groups of dependent transitions have opposing orders. In our case, the groups $t_1, t_4$ and $t_2, t_3$ of mutually dependent transitions are such that $tid(t_1) < tid(t_2)$ but $tid(t_4) > tid(t_3)$.

Our strategy to handle the above problem, is to start scheduling the transitions in increasing order of their thread-ids while taking into account the scheduling constraints imposed by the dependencies. Thus in the above example, suppose that we want to explore interleavings satisfying $t_4 < t_1$ and $t_3 < t_2$. Then we start by first trying to schedule $t_1$. However, since $t_4 < t_1$, we have to schedule $t_4$ before $t_1$. Moreover, since there are no scheduling restrictions (even via transitivity) on $t_2$ and $t_3$, vis-a-vis $t_1$ and $t_4$, and since $tid(t_2) > tid(t_1)$ and $tid(t_3) > tid(t_1)$, we schedule both $t_2$ and $t_3$ to execute after $t_1$. Thus we constrain all interleavings satisfying $t_4 < t_1$ and $t_3 < t_2$ to start with the sequence $t_4, t_1$. Next we try to schedule the transition with the lowest thread-id that has not yet been scheduled, i.e., $t_2$. However, since $t_3 < t_2$, we must schedule $t_3$ first and then $t_2$ resulting in the unique interleaving $t_4 t_1 t_3 t_2$.

In general, for independent transitions $t$ and $t'$, where $tid(t) < tid(t')$, we allow $t'$ to be executed before $t$ only if there is a sequence of transitions $t_0, t_1, ..., t_k$, wherein $t_0 = t'$, each pair of adjacent transitions is dependent, and either $t_k = t$ or $tid(t_k) < tid(t)$. This leads to the key concept of a *dependency chain*.

**Definition 2 (Dependency Chain).** *Let $t$ and $t'$ be transitions executed along a computation $x$ such that $t <_x t'$. A dependency chain along $x$ starting at $t$ is a (sub-)sequence of transitions $tr_{i_0}, ..., tr_{i_k}$ executed 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 executed along $x$ between $tr_{i_j}$ and $tr_{i_{j+1}}$ that is dependent with $tr_{i_j}$.*

We use $t \Rightarrow_x t'$ to denote the fact that there is a dependency chain from $t$ to $t'$ along $x$. Then our strategy can be re-phrased as follow: for independent transitions $t$ and $t'$, where $tid(t) < tid(t')$, we allow $t'$ to be executed before $t$ only if either (i) $t' \Rightarrow_x t$, or (ii) there exists transition $t''$, where $tid(t'') < tid(t)$, $t' <_x t'' <_x t$ and $t' \Rightarrow_x t''$. This leads to the notion of a *quasi-monotonic sequence*.

**Definition 3 (Quasi-Monotonic Computation).** *A computation $x$ is said to be quasi-monotonic if and only if 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 $tr' <_x tr'' <_x tr$.*

**MPOR Strategy.** *Restrict the interleavings explored to the set of all quasi-monotonic computations.*

We now show the following:

**Soundness**., i.e., all necessary interleavings are explored.

**Optimality**. i.e., no two interleavings explored are Mazurkiewicz equivalent.

For soundness, we show the following result.

**Theorem 1. (Soundness).** *For each computation $\pi$ there exists a quasi-monotonic interleaving that is Mazurkiewicz equivalent to $\pi$.*

*Proof.* The proof is by induction on the length $n$ of $\pi$. For the base case, i.e., $n = 1$, the path $\pi$ comprises only of one state and is therefore trivially quasi-monotonic.

For the induction step, we assume that the result holds for all paths of length less than or equal to $k$. Consider a path $\pi$ of length $k + 1$. Write $\pi$ as $\pi = \rho.tr$, where $\rho$ is the prefix of $\pi$ of length $k$ and $tr$ is the last transition executed along $\pi$. By the induction hypothesis, these exists a quasi-monotonic path $\rho'$ that is Mazurkiewicz equivalent to $\rho$. Set $\pi' = \rho'.tr$. Let $\pi' = tr_0...tr_{k-1}tr$. Note that we have represented $\pi'$ in terms of the sequence of transitions executed along it as opposed to the states occurring along it. Thus here $tr_i$ represents the $(i + 1)$st transition executed along $\pi'$. Let $tr' = tr_j$ be the last transition executed along $\rho'$ such that $tid(tr') \leq tid(tr)$. Define $T_{dc} = \{tr_l \,|\, l \in [j + 1, k - 1] \textit{ and } tr_l \Rightarrow_{\pi'} tr\}$ and $T_{nc} = \{tr_l \,|\, l \in [j + 1, k - 1] \textit{ and } tr_l \not\in T_{dc}\}$.

Let $\rho'' = tr_0...tr_j.\nu.tr.\zeta$, where $\nu$ is the sequence of all transitions in $T_{dc}$ listed in the relative order in which they were executed along $\pi'$. Similarly, let $\zeta$ be the sequence of transitions of $T_{nc}$ listed in the relative order in which they were executed along $\pi'$. We claim that $\rho''$ is Mazurkiewicz equivalent to $\pi'$. Indeed, the effect of our transformation on $\pi'$ is to migrate the execution of transitions of $T_{nc}$ rightwards. The only way $\rho''$ cannot be Mazurkiewicz equivalent to $\rho'$ is if there exist transitions $t \in T_{nc}$ and $t' \in T_{dc} \cup \{tr\}$ such that $t$ and $t'$ are dependent. However in that case we can show that $t \in T_{dc}$ contradicting our assumption that $t \in T_{nc}$. Indeed, the only case where we cannot move the transition $t \in T_{nc}$ to the right is if there exists a transition $t' \in T_{dc} \cup \{tr\}$ fired after $t$ along $\rho'$ such that $t'$ is dependent with $t$. Since $t' \in T_{dc} \cup \{tr\}$, by definition of $T_{dc}$, $t' \Rightarrow_{\pi'} tr$. However, since $t$ is dependent with $t'$, we have that $t \Rightarrow_{\pi'} tr$ and so $t \in T_{dc}$.

Set $\pi'' = tr_0...tr_j.\nu'.tr.\zeta'$, where $\nu'$ and $\zeta'$ are quasi-monotonic computations that are Mazurkiewicz equivalent to $\nu$ and $\zeta$, respectively. The existence of $\nu'$ and $\zeta'$ follows from the induction hypothesis. Clearly $\pi''$ is a valid computation.

All we need to show now is that $\pi''$ is quasi-monotonic. If possible, suppose that there exists a pair of transitions $t$ and $t'$ such that $tid(t') > tid(t)$ that violate quasi monotonicity. We now carry out a case analysis. Note that since $tr_0, ..., tr_j$ is quasi-monotonic, $t$ and $t'$ cannot both occur along $tr_0, ..., tr_j$. Thus there are two main cases to be considered: (1) $t'$ occurs along $tr_0, ..., tr_j$ and $t$ along $\nu'.tr.\zeta'$, and (2) $t'$ and $t$ both occur along $\nu'.tr.\zeta'$.

First assume that $t'$ and $t$ occur along $tr_0..., tr_j$ and $\nu'.tr.\zeta'$, respectively. We start by observing that from the definition of $j$ it follows that all transitions executed along $\nu'$ and $\zeta'$ have thread-id greater than $tid(tr) \geq tid(tr_j)$. Thus $tid(t) \geq tid(tr_j)$, and so $tid(t') > tid(t) \geq tid(tr_j)$. Since $tr_0, ..., tr_j$ is quasi-monotonic, either (i) $t' \Rightarrow_{tr_0...tr_j}$

$tr_j$, or (ii) there exist a transition $tr_p$, where $p \in [0..j]$, such that $t' \Rightarrow_{tr_0 \ldots tr_j} tr_p$ and $tid(tr_p) < tid(tr_j)$. If $tr_p \Rightarrow_{\pi''} t$ then from $t' \Rightarrow_{tr_0 \ldots tr_j} tr_p$ it follows that $t' \Rightarrow_{\pi''} t$ and so $t$ and $t'$ do not violate quasi-monotonicity. If, on the other hand, $tr_p \not\Rightarrow_{\pi''} t$ we observe that $tid(tr_p) < tid(tr_j) \leq tid(t)$. Also since $t' \Rightarrow_{tr_0 \ldots tr_j} tr_p$ implies that $t' \Rightarrow_{\pi''} tr_p$, we again see that $t$ and $t'$ do not constitute a violation.

Next we consider case 2, i.e., both $t$ and $t'$ occur along $\nu'.tr.\zeta'$. Note that since by our construction, (i) $\nu'$ and $\zeta'$ are quasi-monotonic, and (ii) there is a dependency chain from each transition occurring along $\nu'$ to $tr$, a violation could occur only if $t'$ occurs along $\nu'.tr$ and $t$ along $\zeta'$. Since $t$ occurs along $\zeta'$, we have $tid(t) > tid(tr)$. Moreover, since $t$ occurs along $\nu'$, there is a dependency chain from $t'$ to $tr$ (note that since $\nu$ and $\nu'$ are Mazurkiewicz equivalent they have the same dependency chains). Thus $t$ and $t'$ satisfy the quasi-monotonicity property thereby contradicting our assumption that $\pi''$ is not quasi-monotonic. This completes the induction step and proves the result. □

For optimality, we show the following result.

**Theorem 2. (Optimality).** *No two computations explored are Mazurkiewicz equivalent.*

*Proof.* We prove by contradiction. Assume that $\pi, \pi'$ are two different quasi-monotonic sequences which are (Mazurkiewicz) equivalent. By definition, $\pi$ and $\pi'$ have the same set of transitions, i.e., $\pi'$ is a permutation of $\pi$. Let $tr_1 = \pi'_i$ be the first transition along $\pi'$ that is swapped to be $\pi_j$, where $i \neq j$, along $\pi$. Let $tr_0 = \pi_i$. Note that $i < j$, else the minimality of $i$ will be contradicted. Then $\pi$ and $\pi'$ share a common prefix up to $i$ (Fig. 4). For definiteness, we assume that $tid(tr_1) < tid(tr_0)$, the other case where $tid(tr_1) > tid(tr_0)$ being handled similarly.

Since $\pi$ and $\pi'$ are Mazurkiewicz equivalent and the relative order of execution of $tr_0$ and $tr_1$ is different along the two paths, $tr_0$ and $tr_1$ must be independent. Since $tid(tr_1) < tid(tr_0)$ and $\pi$ is quasi-monotonic, there must exist a transition $tr_2$, such that $tr_0 <_\pi tr_2 <_\pi tr_1$, $tid(tr_2) < tid(tr_1)$ and $tr_0 \Rightarrow_\pi tr_2$ (note that there cannot exist a dependency chain from $tr_0$ to $tr_1$ else $\pi$ and $\pi'$ will not be Mazurkiewicz equivalent). In Fig. 4, the circle on the square bracket corresponding to $tr_2$ along $\pi$ indicates that $tr_2$ lies between $tr_0$ and $tr_1$ along $\pi$.

Since all adjacent transitions along a dependency chain are, by definition, dependent, the relative ordering of the execution of transitions along any dependency chain must be the same along both $\pi$ and $\pi'$ as they are Mazurkiewicz equivalent. It follows then that $tr_0 <_{\pi'} tr_2$. Since $tr_1 <_{\pi'} tr_0$, we have $tr_1 <_{\pi'} tr_2$. Furthermore, it cannot be the case that $tr_1 \Rightarrow_{\pi'} tr_2$ else to preserve Mazurkiewicz equivalence it must be the case that $tr_1 \Rightarrow_\pi tr_2$ and so $tr_1 <_\pi tr_2$ leading to a contradiction. Therefore, since $\pi'$ is quasi-monotonic and $tid(tr_2) < tid(tr_1)$, there must exist a transition $tr_3$, such that $tr_1 <_{\pi'} tr_3 <_{\pi'} tr_2$, $tid(tr_3) < tid(tr_2)$ and $tr_1 \Rightarrow_{\pi'} tr_3$. Again as before since $tr_1 \Rightarrow_{\pi'} tr_3$, we have $tr_1 \Rightarrow_\pi tr_3$. Thus $tr_1 <_\pi tr_3$. Since $tr_2 <_\pi tr_1$, we have $tr_2 <_\pi tr_3$. But $tid(tr_3) < tid(tr_2)$ and we can repeat the above argument. Thus continuing the above process we can obtain a sequence $tr_0, tr_1, ..., tr_k$ of transitions such that $tid(tr_k) < tid(tr_{k-1}) < ... < tid(tr_1) < tid(tr_0)$ and

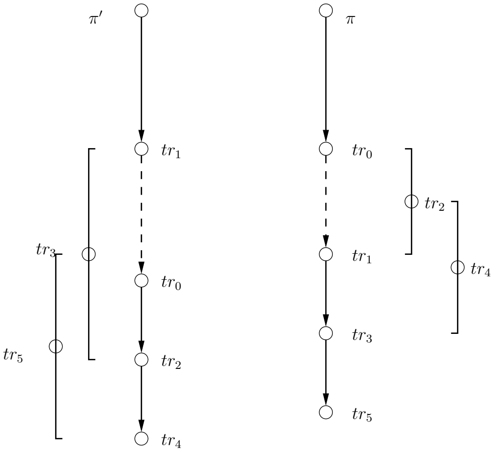


**Fig. 4.** Dependency Chains

1. for each $i \in [0..k-2]$, $tr_i \Rightarrow tr_{i+2}$ ($tr_i \Rightarrow_\pi tr_{i+2}$ and $tr_i \Rightarrow_{\pi'} tr_{i+2}$)
2. for each $i \in [1..k/2]$, $tr_{2i} <_\pi tr_{2i-1}$
3. for each $i \in [0..k/2]$, $tr_{2i+1} <_{\pi'} tr_{2i}$.

Since the thread-ids of the transitions $tr_i$ form a strictly descending sequence, there exists a sequence of transitions of maximum length satisfying the above properties. Assume now that the above sequence is, in fact, maximal. We consider two cases. First assume that $k$ is even. Then there is dependency chain (property 1) from $tr_{k-2}$ to $tr_k$ along $\pi'$. Thus $tr_k$ is executed after $tr_{k-2}$ along both $\pi$ and $\pi'$ and so $tr_{k-2} <_{\pi'} tr_k$. Also, by property 3, $tr_{k-1} <_{\pi'} tr_{k-2}$. By combining the above facts, we have $tr_{k-1} <_{\pi'} tr_{k-2} <_{\pi'} tr_k$. Note also that $tid(tr_k) < tid(tr_{k-1})$. Thus by quasi-monotonicity of $\pi'$ either (i) there exists a dependency chain from $tr_{k-1}$ to $tr_k$, or (ii) there exists a transition $tr_{k+1}$ such that $tr_{k-1} \Rightarrow tr_{k+1}$ and $tr_{k-1} <_{\pi'} tr_{k+1} <_{\pi'} < tr_k$. The second case cannot happen as it would violate the maximality of the sequence $\{tr_i\}$. Thus $tr_{k-1} \Rightarrow tr_k$ which implies that $tr_{k-1} <_\pi tr_k$ (as dependency chains are preserved across Mazurkiewicz equivalent sequences). However by property 2, $tr_k <_\pi tr_{k-1}$ which is absurd. This contradicts our initial assumption that there exist two different Mazurkiewicz equivalent quasi-monotonic sequences. The other case where $k$ is odd can be handled similarly. This completes the proof. □

# 4 Implementation

## 4.1 Bounded Model Checking (BMC)

We start by reviewing the basics of SMT/SAT-based bounded model checking. 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 $\Psi$ such that *$\Psi$ is satisfiable iff there exists a length-$k$ execution that* *violates the property*. The formula is denoted $\Psi = \Phi \wedge \Phi_{prop}$, where $\Phi$ represents all possible executions of the program up to $k$ steps and $\Phi_{prop}$ is the constraint indicating violation of the property (see [2] for more details about $\Phi_{prop}$). In the following, we focus on the formulation of $\Phi$.

Let $V = V_{global} \cup \bigcup V_i$, where $V_{global}$ is the set of global variables and $V_i$ the set of local variables of $T_i$. For all local (global) program variables, we add a state variable for $V_i$ ($V_{global}$). Array and pointer accesses need special handling. For an array access $a[i]$, we add separate variables for the index $i$ and for the content $a[i]$. Similarly, for a pointer access $*p$, we maintain separate state variables for $(*p)$ and $p$. 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.

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

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 partial order reduction, $SCH(V^i) := true$, which means that $sel$ takes all possible values at every step. This default $SCH$ considers all possible interleavings. Partial order reduction can be implemented by adding constraints to $SCH$ to remove redundant interleavings.

We now consider the formulation of $TR_j$. Let $VS_j = V_{global} \cup V_j$ denote the set of variables visible to $T_j$. At the $i$-th time frame, for each $t \in trans_j$ (a transition between control locations $loc_1$ and $loc_2$), we create $tr_t^i$. If $t$ is an assignment $v := e$, then $tr_t^i :=$

$$pc_j^i = loc_1 \wedge pc_j^{i+1} = loc_2 \wedge v^{i+1} = e^i \wedge (VS_j^{i+1} \setminus v^{i+1}) = (VS_j^i \setminus v^i) \;.$$

If $t$ is a branching statement$assume(c)$, as in `if(c)`, then $tr_t^i :=$

$$pc_j^i = loc_1 \wedge pc_j^{i+1} = loc_2 \wedge c^i \wedge VS_j^{i+1} = VS_j^i.$$

Overall, $TR_j^i$ is defined as follows:

$$TR_j^i := \quad \left( sel^i = j \wedge \bigvee_{t \in trans_j} tr_t^i \right) \vee \left( sel^i \neq j \wedge V_j^{i+1} = V_j^i \right)$$

The second term says that if $T_j$ is not selected, variables in $V_j$ do not change values.

## 4.2 Encoding MPOR

In order to implement our technique, we need to track dependency chains in a space efficient manner. Towards that end, the following result is crucial.

**Theorem 3.** *Let transitions $tr$ and $tr'$ executed by processes $T_i$ and $T_j$, respectively, along a computation $x$, constitute a violation of quasi-monotonicity. Suppose that $tr' <_x tr$ and $tid(tr') > tid(tr)$. Then any transition $tr''$ executed by $T_j$ such that $tr' <_x tr'' <_x tr$ also constitutes a violation of quasi-monotonicity with respect to $tr$.*

*Proof.* If possible, suppose that the pair of transitions $tr''$ and $tr$ do not constitute a violation of quasi-monotonicity. Since $tid(tr'') > tid(tr)$ and $tr'' <_x tr$, either (1) there is a dependency chain from $tr''$ to $tr$, or (2) there exists $tr'''$ such that (a) $tr'' <_x tr''' <_x tr$, (b) $tid(tr''') < tid(tr)$, and (c) there is a dependency chain from $tr''$ to $tr'''$. However, since all transitions belonging to the same thread are dependent with each other, we see that $tr'$ is dependent with $tr''$. Thus any dependency chain starting at $tr''$ can be extended backwards to start at $tr'$. As a result we have that either (1) there is a dependency chain from $tr'$ to $tr$, or (2) there exists $tr'''$ such that (a) $tr' <_x tr''' <_x tr$, (b) $tid(tr''') < tid(tr)$, and (c) there is a dependency chain from $tr'$ to $tr'''$. However, in that case transitions $tr'$ and $tr$ do not violate quasi-monotonicity, leading to a contradiction. □

Theorem 3 implies that if there is a violation of quasi-monotonicity involving transitions $tr$ and $tr'$ executed by threads $T_i$ and $T_j$, respectively, such that $tid(tr') > tid(tr)$, then there is also a violation between $tr$ and the last transition executed by $T_j$ before $tr$ along the given computation. This leads to the important observation that in order to ensure that a computation $\pi$ is quasi-monotonic, we need to track dependency chains only from the last transition executed by each process along $\pi$ and not from every transition.

**Tracking Dependency Chains.** To formulate our MPOR encoding, we first show how to track dependency chains. Towards that end, for each pair of threads $T_i$ and $T_j$, we introduce a new variable $DC_{ij}$ defined as follows.

**Definition 4.** *$DC_{il}(k)$ 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 $T_l$ up to time step $k$. If no transition has been executed by $T_i$ till time step $k$, $DC_{il} = 0$.*

**Updating $DC_{ij}$.** 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 dependent with $tr$. To track that we introduce the dependency variables $DEP_{li}$ defined below.

**Definition 5.** *$DEP_{li}(k)$ 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 $DEP_{ii}(k)$ is always true (due to control conflict).*

If $DEP_{li}(k) = true$ and if $DC_{jl}(k-1) = 1$, i.e., there is a dependency chain from the last transition executed by $T_j$ to the last transition executed by $T_l$, then this 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 dependency chains only from the last transition executed by each thread, the dependency chain corresponding to $T_i$ needs to start afresh and so we set $DC_{ij}(k) = -1$ for all $j \neq i$. To sum up, the updates are as follows.

$$
\begin{array}{ll}
DC_{ii}(k) = 1 & \\
DC_{ij}(k) = -1 & \text{when } j \neq i \\
DC_{ji}(k) = 0 & \text{when } j \neq i \text{ and } DC_{jj}(k-1) = 0 \\
DC_{ji}(k) = \bigvee_{l=1}^{n}(DC_{jl}(k-1) = 1 \wedge DEP_{li}(k)) & \text{when } j \neq i \text{ and } DC_{jj}(k-1) \neq 0 \\
DC_{pq}(k) = DC_{pq}(k-1) & \text{when } p \neq i \text{ and } q \neq i
\end{array}
$$

**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 $SCH$ (see subsection 4.1):

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

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

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

**Encoding DEP.** The decoupling of the encoding of the dependency constraints (via the *DEP* variables) from the encoding of quasi-monotonicity has the advantage that it affords us the flexibility to incorporate various notions of dependencies based on the application at hand. These include dependencies arising out of synchronization primitives, memory consistency models like sequential consistency, etc. For our implementation, we have, for now, used only dependencies arising out of shared variable accesses the encoding of which is given below.

We define the following set of variables for each thread $T_i$:

- $pWV_i(k)$, $pRV_i(k)$, $pR^2V_i(k)$ denote the Write-Variable and Read-Variables of the last transition executed by $T_i$ before step $k$. For simplicity, we assume that each assignment has at most three operands: a write variable occurring on the left hand side of the assignment, i.e., $pWV_i(k)$ and up to two read variables occurring on the right hand side of the assignment, i.e., $pRV_i(k)$ and $pR^2V_i(k)$.
- $wv_i(k)$, $wr_i(k)$, $r^2v_i(k)$ denote the Write-Variable and Read-Variables of the transition executed by $T_i$ at step $k$.

We encode $DEP_{ij}(k)$ as follows,

$$\begin{aligned} DEP_{ij}(k) = (\ & pWV_i(k) = wv_i(k) \wedge pWV_i(k) \neq 0 \vee \\ & pWV_i(k) = rv_i(k) \wedge pWV_i(k) \neq 0 \vee \\ & pWV_i(k) = r^2v_i(k) \wedge pWV_i(k) \neq 0 \vee \\ & pRV_i(k) = wv_i(k) \wedge wv_i(k) \neq 0 \vee \\ & pR^2V_i(k) = wv_i(k) \wedge wv_i(k) \neq 0) \end{aligned}$$

**Read and Write Variables.** Let $t_1, \ldots, t_n \in trans_i$ be the set of transitions of $T_i$, and $t_1.writeVar$ be the Write-Variable of the transition $t_1$. Moreover, $en_{t_i}(V^k)$ equals *true* or *false* accordingly as $t_i$ is enabled at time step $k$ or not, respectively.

- We encode $wv_i(k)$ as follows

$$\begin{aligned} wv_i(k) = & (sel^k = i \wedge en_{t_1}(V^k))\ ?\ t_1.writeVar\ : \\ & (sel^k = i \wedge en_{t_2}(V^k))\ ?\ t_2.writeVar\ : \\ & \ldots \\ & (sel^k = i \wedge en_{t_n}(V^k))\ ?\ t_n.writeVar\ :\ 0 \end{aligned}$$

- We encode $pWV_i(k+1)$ as follows (with $pWV_i(0) = 0$)

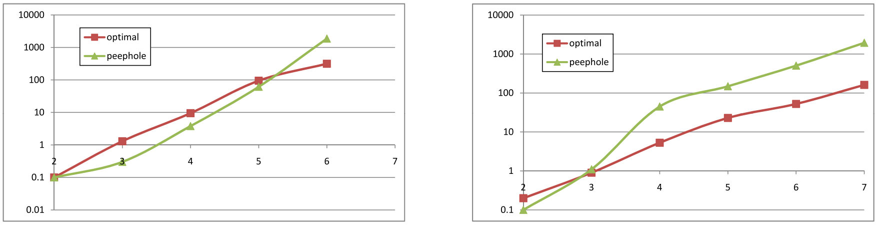
$$\begin{aligned} pWV_i(k+1) = & (sel^k = i \wedge en_{t_1}(V^k))\ ?\ t_1.writeVar\ : \\ & (sel^k = i \wedge en_{t_2}(V^k))\ ?\ t_2.writeVar\ : \\ & \ldots \\ & (sel^k = i \wedge en_{t_n}(V^k))\ ?\ t_n.writeVar\ :\ pWV_i(k) \end{aligned}$$

**Important Optimization.** Note that the last encoding requires an if-then-else chain of length $|trans_i|$. However, we need to detect dependencies only between transitions of threads which access shared objects (as all internal transitions following a shared object access can be executed in one atomic step). Thus, $trans_i$ would now denote the number of transitions of $T_i$ accessing only shared objects which typically is a small fraction of the total number of transitions of $T_i$.

## 5 Experiments

We have implemented the optimal POR methods in an SMT-based bounded model checker using the Yices SMT solver [4]. The experiments were performed with two variants of the optimal POR reduction and a baseline BMC algorithm with no POR. The two variants represent different tradeoffs between the encoding overhead and the amount of achievable reduction. The first one is *PPOR* [17], in which the quasi monotonicity constraints are collected only within a window of two consecutive time frames (and so the reduction is not optimal). The second one is *MPOR*, in which the entire set of quasi-monotonicity constraints are added to ensure quasi monotonicity (the reduction is optimal). Our experiments were conducted on a workstation with 2.8 GHz Xeon processor and 4GB memory running Red Hat Linux 7.2.

We use a parameterized version of *dining philosophers* as our test example. The dining philosopher model we used can guarantee the absence of deadlocks. Each philosopher



(a) With SAT instances (property ’pa’)

(b) With UNSAT instances (property ’pb’)

**Fig. 5.** Comparing runtime performance of (optimal) MPOR and (peephole) PPOR

has its own local state variables, and threads communicate through a shared array of chopsticks. When accessing the global array, threads may have conflicts (data dependency). The first property (pa) we checked is whether all philosophers can eat simultaneously (the answer is no). The second property (pb) is whether it is possible to reach a state in which all philosophers have eaten at least once (the answer is yes).

We set the number of philosophers (threads) to 2, 3, . . ., and compared the runtime performance of the three methods. The results are given in Fig. 5. The $x$-axis represents unroll depth. The $y$-axis is the BMC runtime in seconds, and is in logarithmic scale. The number of variable decisions and conflicts of the SMT solver look similar to the runtime curves and are, therefore, omitted for brevity. When comparing the sizes of the SMT formulae, we found that those produced by the optimal POR encoding typically are twice as large as the plain BMC instances, and those produced by the PPOR encoding are slightly larger than the plain BMC instances.

The detailed results are given in Table 1. In Table 1, Columns 1-3 show the name of the examples, the number of BMC unrolling steps, and whether the property is true or

**Table 1.** Comparing PPOR, MPOR and plain BMC

| Test Program | | | Total CPU Time (s) | | | #Conflicts (k) | | | #Decisions (k) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| name | steps | prop | none | MPOR | PPOR | none | MPOR | PPOR | none | MPOR | PPOR |
| phil2-pa | 15 | unsat | 0.2 | 0.2 | **0.1** | 1 | 1 | 1 | 1 | 1 | 0 |
| phil3-pa | 22 | unsat | 18.2 | **0.9** | 1.1 | 17 | 1 | 1 | 23 | 2 | 3 |
| phil4-pa | 29 | unsat | 49.6 | **5.3** | 44.9 | 39 | 3 | 27 | 53 | 8 | 41 |
| phil5-pa | 36 | unsat | 76.3 | **22.9** | 148.6 | 48 | 6 | 53 | 69 | 17 | 82 |
| phil6-pa | 43 | unsat | 98.4 | **52.3** | 504.4 | 56 | 12 | 92 | 84 | 30 | 147 |
| phil7-pa | 50 | unsat | 502.3 | **161.6** | > 1h | 161 | 16 | - | 238 | 48 | - |
| phil2-pb | 15 | sat | 0.1 | 0.1 | **0.1** | 1 | 1 | 1 | 1 | 1 | 0 |
| phil3-pb | 22 | sat | 1.5 | 1.3 | **0.3** | 2 | 1 | 1 | 4 | 4 | 1 |
| phil4-pb | 29 | sat | 18.3 | 9.5 | **3.8** | 12 | 3 | 3 | 17 | 11 | 6 |
| phil5-pb | 36 | sat | 195.5 | 94.7 | **61.7** | 44 | 9 | 16 | 61 | 26 | 31 |
| phil6-pb | 43 | sat | >1h | **315.4** | 2283 | - | 16 | 122 | - | 52 | 200 |
| phil7-pb | 50 | sat | >1h | **1218** | > 1h | - | 31 | - | - | 85 | - |

not. Columns 4-6 report the runtime of the three methods. Columns 7-9 and Columns 10-12 report the number of backtracks and the number of decisions of the SMT solver.

In general, adding more SAT constraints involves a tradeoff between the state space pruned and the additional overhead in processing these constraints. However, the results in Fig. 5 indicate that the reduction achieved by MPOR more than outweighs its encoding overhead. For programs with two threads, PPOR always outperforms MPOR. This is because PPOR is also optimal for two threads, and it has a significantly smaller encoding overhead. However, as the number of threads increases, percentage-wise, more and more redundant interleavings elude the PPOR constraints. As is shown in Fig. 1, for more than four threads, the overhead of PPOR constraints outweighs the benefit (runtime becomes longer than MPOR).

## 6 Conclusions

We have presented a monotonic partial order reduction method for model checking concurrent systems, based on the new notion of quasi-monotonic sequences. We have also presented a concise symbolic encoding of quasi-monotonic sequences which is well suited for use in SMT/SAT solvers. Finally, our new method is guaranteed optimal, i.e., removes all redundant interleavings.

## References

[1] Alur, R., Brayton, R.K., Henzinger, T.A., Qadeer, S., Rajamani, S.K.: Partial-order reduction in symbolic state-space exploration. Formal Methods in System Design 18(2), 97–116 (2001)

[2] Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, p. 193. Springer, Heidelberg (1999)

[3] Cook, B., Kroening, D., Sharygina, N.: Symbolic model checking for asynchronous boolean programs. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 75–90. Springer, Heidelberg (2005)

[4] Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for dpll(t). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)

[5] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of programming languages (POPL 2005), pp. 110–121 (2005)

[6] Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Springer, Heidelberg (1996)

[7] Godefroid, P., Pirottin, D.: Refining dependencies improves partial-order verification methods. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 438–449. Springer, Heidelberg (1993)

[8] Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. In: POPL (2005)

[9] Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007)

[10] Kahlon, V., Gupta, A., Sinha, N.: Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 286–299. Springer, Heidelberg (2006)

[11] Katz, S., Peled, D.: Defining conditional independence using collapses. Theor. Comput. Sci. 101(2), 337–359 (1992)

[12] Lerda, F., Sinha, N., Theobald, M.: Symbolic model checking of software. Electr. Notes Theor. Comput. Sci. 89(3) (2003)

[13] Mazurkiewicz, A.W.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1987)

[14] Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697. Springer, Heidelberg (1993)

[15] Silva, J.P.M., Sakallah, K.A.: Grasp—a new search algorithm for satisfiability. In: International Conference on Computer-Aided Design, San Jose, CA (1996)

[16] Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)

[17] Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008)