Cartesian Partial-Order Reduction

Guy Gueta[1], Cormac Flanagan[2], Eran Yahav[3], and Mooly Sagiv[1]

[1] Tel Aviv University
{guygueta,msagiv}@post.tau.ac.il
[2] University of California at Santa Cruz
cormac@soe.ucsc.edu
[3] IBM T.J. Watson Research Center
eyahav@us.ibm.com

Abstract. Verifying concurrent programs is challenging since the number of thread interleavings that need to be explored can be huge even for moderate programs. We present a *cartesian semantics* that reduces the amount of non-determinism in concurrent programs by delaying unnecessary context switches. Using this semantics, we construct a novel dynamic partial-order reduction algorithm. We have implemented our algorithm and evaluate it on a small set of benchmarks. Our preliminary experimental results show a significant potential saving in the number of explored states and transitions.

1 Introduction

This paper addresses the problem of proving the correctness of a concurrent program, *i.e.*, of showing that all possible program traces satisfy certain correctness properties. We define a *cartesian* partial order reduction technique that allows to safely consider only a subset of these program traces. Our technique can be combined with existing finite state model checkers to yield new algorithms for finite state systems. It can also be combined with abstract interpretation [4] to yield new conservative algorithms for infinite systems. In both cases we expect to obtain significant speedups without sacrificing soundness or completeness. We have implemented a model checker based on cartesian partial order reduction, and provide preliminary experimental results that show a significant reduction in the number of states and transitions explored. Our experiments also compare the performance of our algorithm to the partial order reduction techniques of SPIN [12], and the recent technique of [6]. Compared to these techniques, cartesian partial order reduction saves more states and transitions on most of our example programs.

1.1 Partial Order Reduction

Partial order reduction techniques [8,14,17] combat state explosion by only exploring a representative subset of all possible program traces. In general, however, verifying that a subset of all traces is representative may be as hard as solving the underlying verification problem. Therefore, existing partial order reduction techniques mostly focus on two special cases: "sleep sets" [8, pp. 75] and "persistent sets" [8, pp. 41]. In particular, a transition is established as persistent by checking for its potential collisions with an infinite future of another thread. Such collisions are traditionally detected via static analysis (e.g., [5]), which may yield coarse results for complicated or pointer-rich code. Alternatively, dynamic partial order reduction [6] infers persistent sets dynamically as part of a stateless search, but is applicable only to cycle-free systems. The algorithm of [5] also infers persistent sets dynamically, but only for thread-local and lock-protected data.

1.2 Main Results

In this paper, we present a new approach for partial order reduction. This approach identifies and exploits a different kind of redundancy than either sleep sets or persistent sets. The strength of our approach stems from the fact that, unlike in persistent sets, where a transition must be checked for conflicts with an *infinite* future of another thread, we only inspect a finite future for collisions, and guarantee safety by exploring both possible extensions at any collision point. In Sec. 4.1, we show that this approach yields significant improvements even over optimal persistent sets. This result is also supported by our preliminary empirical study in Sec. 7.

Our technique is presented as new operational (or execution) semantics that can be applied to both finite and infinite systems. In particular, it can be combined with abstract interpretation in order to conservatively handle infinite traces and infinite state systems.

```
N=12;
boolean A[N,N];
Robot(int x,int y)
 int dirX = 1, dirY = 1;
 while(true)
  A[x,y]=false;
  x += dirX; y += dirY;
  if(x=N-1 or x=0) dirX*=(-1);
  if(y=N-1 or y=0) dirY*=(-1);
  assert(A[x,y]⇒(x=9 or x=2));
  A[x,y]=true;
Main()
 newthread Robot(0,0);
 newthread Robot(4,0);
```

Fig. 1. Two threads implementing robots

A motivating example. The concurrent program of Fig. 1 simulates an arena with two robots which move in different paths. Each robot is represented by a thread that calculates and updates its position in an infinite loop. The program verifies that the robots can meet only at the 9th and the 2nd rows by using an assert instruction (identical to the Java assert). Although this program is quite simple, its state space is relatively large. An attempt to reduce the state space by existing partial order reduction methods is problematic because:

1. Most partial order reduction methods (e.g., persistent sets) are based on a static dependence analysis. Such analyses will fail to establish the independence of the transitions in this program, and therefore yield a poor reduction of the state space.
2. Dynamic partial order reduction [6] requires a stateless search, and so cannot handle examples such as this one, where there are cycles in the state space.
3. The approach of [5] provides limited benefit on this benchmark because it does not contain much thread-local or lock-protected data.

In Sec. 7, we show that our approach saves close to 73% of the transitions that need to be explored for this program.

We present cartesian partial order reduction as an operational (or execution) semantics, which we believe makes it simpler to understand and to establish correctness (see [9]). For example, in contrast to the dynamic analysis of [6], it does not rely on happens-before relations [13]. Also, since it saves intermediate states, it supports cycles and behaves well in transition systems with multiple paths into a single state. Finally, it can be combined with (counter-example driven) abstract interpretation to handle concurrent programs with infinite state spaces (e.g., [19]).

The contributions of this paper can be summarized as follows:

– We present a novel *cartesian semantics* that reduces the nondeterminism in concurrent programs.
– Based on this semantics, we derive a corresponding *cartesian partial order reduction* algorithm that can be used to improve both finite-state model checkers and infinite-state abstract interpreters. Our algorithm identifies dependencies dynamically, avoiding the inherent imprecision of static dependence analyses. It also overcomes the cycle-free restriction of [6], and so is applicable to more programs.
– We present preliminary experimental results showing that our approach can lead to significant savings in the number of explored states and transitions. We also show that our approach is beneficial in cases where traditional partial order reduction methods are unable to reduce the space.

The rest of this paper is organized as follows. Sec. 2 provides an informal overview of our method. Sec. 3 includes basic definitions and notations. Sec. 4 defines our cartesian semantics and shows that it is observationally equivalent to the standard semantics. Sec. 5 and Sec. 6 realize this semantics as a model checking algorithm. Sec. 7 reports initial empirical results on the behavior of this model checking algorithm. Sec. 8 describes related work and Sec. 9 concludes. Appendix A describes the benchmarks from Sec. 7.

2 Overview

This section provides an overview of our approach for the simple concurrent program shown in Fig. 2. The two threads in this program share two variables, `x` and `y`, and all variables are initially zero.

Thread 1:	Thread 2:
`0: z := 8`	`0: q := 8`
`1: x := 1`	`1: priv := y`
`2: z := 42`	`2: q := 42`
`3: y := 7`	`3: priv := x`
`4: w := z`	`4: nop`

Fig. 2. Two threads using shared variables `x` and `y`

Whereas traditional model checking would explore all possible interleavings of these two threads, our approach explores only a representative subset of these interleavings, based on the notion of *dependent transitions*. For this program, there are two pairs of dependent transitions: the statement `x` := `1` (of thread 1) is dependent with `priv` := `x` (of thread 2); similarly, `y` := `7` is dependent with `priv` := `y`. (In this simple example, a static notion of dependence is sufficient. Our approach detects dependencies dynamically, however, thus overcoming the inherent imprecision of statically identified dependencies.)

The key idea of our approach is to find, for each explored state, a sequence of transitions for each thread such that only the *last* transitions in these two sequences are allowed to be dependent (i.e., every pair of transitions other than the last two transitions must be independent). We refer to the two sequences of transitions found for a state as a *cartesian vector* for that state.

For the program's initial state, a suitable cartesian vector is:

T_1: `z:=8; x:=1` T_2: `q:=8; priv:=y; q:=42; priv:=x`

since `z:=8` is independent of all transitions in T_2's sequence, and `x:=1` is independent of all transitions in T_2's sequence except the last. The last transitions `x:=1` and `priv:=x` may be (and indeed are) dependent.

After finding the two sequences, we nondeterministically pick one of them, execute that sequence in its entirety (without a context switch), and then continue exploration from that resulting state. For example, suppose we first execute the sequence T_1 : `z:=8; x:=1`. At the resultant state, a suitable cartesian vector is:

T_1: `z:=42; y:=7` T_2: `q:=8; priv:=y`

since only the last pair of transitions are dependent. Again, we nondeterministically pick one of these sequences and execute it entirely, without context switches.

By proceeding in this manner, we eventually explore all possible orderings of the dependent transitions in this program. Fig. 3 shows how our approach explores a representative subset of all possible traces of this program.

As an aside, it is worth noting that the statement `z:=8` in T_1 is a persistent transition, as it has no future collisions with T_2. In principle, this could have allowed exploring only representative traces that begin with `z:=8` as their first step. Establishing that `z:=8` is indeed a persistent transition, however, requires inspection of the future execution of T_2 (which in general, may be infinite). In some cases, the persistence of a transition can be established by a preceding static dependence analysis phase. Like methods based on persistent sets, our approach can also benefit from such static dependence information when it exists. Unlike `z:=8`, the statement `x:=1` is not persistent, as it has a future collision with `priv:=x` in T_2 (as long as `priv:=x` is not executed).

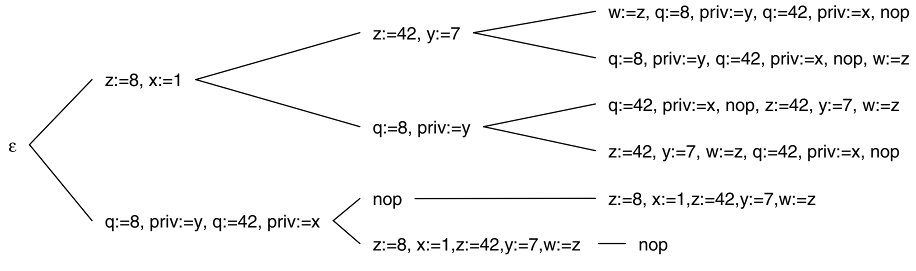

ε
z:=8, x:=1
q:=8, priv:=y, q:=42, priv:=x
z:=42, y:=7
q:=8, priv:=y
nop
z:=8, x:=1,z:=42,y:=7,w:=z
w:=z, q:=8, priv:=y, q:=42, priv:=x, nop
q:=8, priv:=y, q:=42, priv:=x, nop, w:=z
q:=42, priv:=x, nop, z:=42, y:=7, w:=z
z:=42, y:=7, w:=z, q:=42, priv:=x, nop
z:=8, x:=1,z:=42,y:=7,w:=z
nop

Fig. 3. Exploration of representative traces of the example program of Fig. 2

3 Basic Definitions

We consider a concurrent system composed of a finite set *Threads* of threads. The threads communicate by performing atomic operations on communication objects (e.g. shared variables).

A *state* of the concurrent system consists of the *LocalState* of each thread (the values for all the thread's private variables), and of the *SharedState* (values for all the communication objects). That is, $\mathit{State} = \mathit{SharedState} \times \mathit{LocalStates}$ where $\mathit{LocalStates} = \mathit{Threads} \rightarrow \mathit{LocalState}$. For $ls \in \mathit{LocalStates}$, we write $ls[T \mapsto l]$ to denote the map that is identical to ls except that it maps T to the local state l.

A *transition* moves the system from one state to a subsequent state, by performing an atomic operation of a chosen thread. The transition $t_{T,l}$ of thread T for local state l is defined via a total function: $t_{T,l}\colon \mathit{SharedState} \rightarrow \mathit{LocalState} \times \mathit{SharedState}$. A transition $t_{T,l} \in \tau$ is *enabled* in a state $s = \langle g, ls\rangle$ (where $g \in \mathit{SharedState}$ and $ls \in \mathit{LocalStates}$) if $l = ls(T)$. If $t = t_{T,l}$ is enabled in $s = \langle g, ls\rangle$ and $t(g) = \langle g', l'\rangle$, then we say the execution of t from s produces a unique successor state $s' = \langle g', ls[T \mapsto l']\rangle$, written $\mathit{exec}(s,t) = s'$ or $s \Rightarrow s'$. We say that *q is reachable from s in the standard semantics* if $s \overset{*}{\Rightarrow} q$.

Notice that in a given state every thread has exactly one enabled transition, therefore no thread can be blocked. This is not restrictive, as blocking or termination of a thread can be modeled by a self loop. Let τ denote the set of all transitions of the system $\tau = \{t_{T,l} | T \in \mathit{Threads}, l \in \mathit{LocalState}\}$.

A *trace* is an infinite sequence $\sigma = s_1, t_1, s_2, t_2, \ldots$ such that for every $i \in \mathbb{N}^+$, $\mathit{exec}(s_i, t_i) = s_{i+1}$. A *trace prefix* is a nonempty (possibly infinite) prefix of a trace, that does not end with a transition. We denote the set of all trace prefixes (of the considered concurrent system) by *Prefix*. A *legal prefix of thread T* is a trace prefix that has at least one transition and all its transitions are executed by thread T.

For $A \in \mathit{Prefix}$, we say that $t \in A$ if t is a transition in A. We denote the last transition of A by $\mathit{last\_tran}(A)$. If there is no transition in A or A is infinite then $\mathit{last\_tran}(A)=\perp$. We denote the first and last states of A by $\mathit{first}(A)$ and $\mathit{last}(A)$ respectively. If A is infinite then $\mathit{last}(A)=\perp$. We denote the set of states in A by $\mathit{states}(A)$.

Our cartesian partial order reduction technique is based on the notion of transitions being *independent*, which essentially means that the order in which these transitions are executed does not matter.

Definition 1 (Independence). *We say that transitions t and t' of different threads are* independent *if* [1] *for every $s \in \mathit{State}$: $t, t' \in \mathit{enabled}(s) \implies \mathit{exec}(\mathit{exec}(s,t),t') = \mathit{exec}(\mathit{exec}(s,t'),t)$. If two transitions of different threads t and t'* are independent, *then we write $t \parallel t'$, otherwise we write $t \not\parallel t'$.*

4 Cartesian Partial Order Reduction

The standard semantics of multithreaded programs nondeterministically chooses a thread for scheduling right after every transition, but this degree of nondeterminism results in state space explosion. In this section, we present a non-standard *cartesian* semantics that avoids many context switches, while preserving both soundness and completeness.

[1] Sometimes similar definitions require that independent transitions are not disable each other, this is not necessary because two transitions from different threads can never disable each other in the presented concurrent system.

As outlined in Section 2, our cartesian semantics is defined in terms of *cartesian vectors*. Essentially, a cartesian vector (CV) for a state describes a sequence of transitions that each thread can perform without context switches from that state.

Definition 2 (Cartesian Vector). *In a concurrent system with n threads of control, a vector $(p_1, \ldots, p_n) \in$ Prefix^n is a* cartesian vector from a state s *if for every $T_i, T_j \in$ Threads the following holds:*

1. *first$(p_i) = s$;*
2. *p_i is a legal prefix of thread T_i;*
3. *$\forall t \in p_i, t' \in p_j : t \not\parallel t' \implies t = last\_tran(p_i) \land t' = last\_tran(p_j)$.*

Intuitively, this definition implies that if two prefixes are in the same cartesian vector, then only their last transitions may depend on each other. Note that each state may have multiple CVs. In particular, every state has at least the *minimal CV*, which contains exactly one transition for each thread, but many states will also admit larger CVs.

Example 1. For the program of Fig. 2, consider the two trace prefixes from the initial state: p_1 is the sequence `z:=8; x:=1; z:=42` (of thread 1) and p_2 is the sequence `q:=8; priv:=y` (of thread 2). Each prefix accesses different variables, therefore the vector (p_1, p_2) is a cartesian vector for the initial state.

Now consider the longer prefix p'_1: `z:=8; x:=1; z:=42; y:=7`. In this case (p'_1, p_2) is still a cartesian vector because only the last transitions are dependent.

To generate a cartesian vector for any explored state, we assume the existence of an *cartesian function* ϕ: *State* $\rightarrow$ *Prefix*^{n} such that, for every $s \in$ *State*, $\phi(s)$ is a cartesian vector from s. Every state space has at least the *minimal cartesian function*, which simply returns the minimal CV for each state (see Section 5). Section 5 describes an algorithm for computing better CVs.

Given a cartesian function ϕ, we can build a *a cartesian semantics* that uses ϕ as a guide for execution. The intuition behind the cartesian semantics is as follows: when the cartesian semantics starts the execution from a state s it selects a prefix σ from the vector $\phi(s)$ and executes the transitions of σ. When the semantics reaches *last*(σ) (the last state of σ) it starts the procedure again from *last*(σ). If σ is infinite it continues to go over the states of σ forever.

The cartesian semantics generated by ϕ is formalized as two binary relations $\longrightarrow_\phi$ and $\Longrightarrow_\phi$ on states, where $\longrightarrow_\phi$ relates states at the end of prefixes, and is transitively closed, and $\Longrightarrow_\phi$ extends $\longrightarrow_\phi$ to also include intermediate states.

Definition 3. *We define the binary relations $\longrightarrow_\phi$ and $\Longrightarrow_\phi$ on State with respect to a cartesian function ϕ inductively in Fig. 4. Here $\longrightarrow_\phi$ is the relation on final states in which scheduling occurs and $\Longrightarrow_\phi$ is the relation on both final and intermediate states.*

An important property of cartesian semantics is described by the following theorem, which states that the set of local states is identical for the standard semantics and the cartesian semantics. Consequently, if a thread sees a violation of a local safety property (e.g., by using an assert instruction as in Java), then the same thread will see the same violation under the cartesian semantics.

$$s \longrightarrow_{\phi} s \qquad \text{reflexivity}$$

$$s \longrightarrow_{\phi} s' \qquad \exists \pi \in \phi(s) \colon s' = \mathit{last}(\pi) \qquad \text{basis}$$

$$\frac{s \longrightarrow_{\phi} s' \quad s' \longrightarrow_{\phi} s''}{s \longrightarrow_{\phi} s''} \qquad \text{transitivity}$$

$$s \Longrightarrow_{\phi} s \qquad \text{reflexivity}$$

$$s \Longrightarrow_{\phi} s' \qquad \exists \pi \in \phi(s) \colon s' \in \mathit{states}(\pi) \qquad \text{basis}$$

$$\frac{s \longrightarrow_{\phi} s' \quad s' \Longrightarrow_{\phi} s''}{s \Longrightarrow_{\phi} s''} \qquad \text{pseudo-transitivity}$$

Fig. 4. Inference rules for a cartesian semantics

Theorem 1. *For every cartesian function ϕ, if $s \overset{*}{\Rightarrow} \langle g, ls[T \mapsto l]\rangle$ then there exist $g' \in \mathit{SharedState}$ and $ls' \in \mathit{LocalStates}$ such that $s \Longrightarrow_{\phi} \langle g', ls'[T \mapsto l]\rangle$.*

The proof appears in [9].

The situation with global properties is somewhat more complex. To illustrate this situation, consider again the program of Fig. 2, for which we can build a cartesian semantics with the following cartesian vector from the initial state: T_1: `z:=8; x:=1; z:=42`, T_2: `q:=8; priv:=y; q:=42`. This cartesian semantics will never reach a state with `z` $= 8$ and `q` $= 8$. Therefore, the global property *"there is a state in which* `z`*=8 and* `q`*=8*" cannot be directly proven by using the cartesian semantics. Instead, we can convert this global property into a local property by introducing a dummy thread that merely observes the variables involved in the property (i.e., a thread that reads `z` and `q` in an infinite loop), and then use the cartesian semantics to verify this localized version of the original global property.

4.1 Cartesian Semantics Versus an Optimal Persistent Sets Algorithm

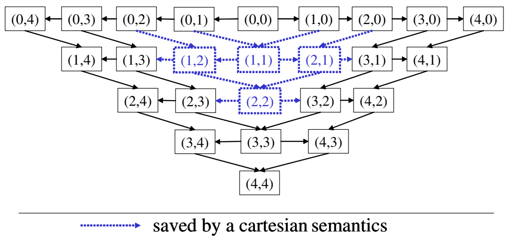
To illustrate the relation between the cartesian semantics and persistent sets, consider the example program shown in Fig. 5 (a). For this example, the program counters of the two threads uniquely define the current value of `x` and `y`, and so we can represent each state simply as a pair of program counters (pc_1, pc_2).

For this program, an optimal persistent sets algorithm will save only one transition, that from the state (3,3), because in any other state, in which the two threads have not terminated, there is a collision between the next step of T1 and a future step of T2 (and, symmetrically, a collision between the next step of T2 and a future step of T1).

In contrast, a suitable cartesian vector for this program's initial state is: $T1$: `x++;x++;x++;` $T2$: `y++;y++;y++`. Hence, the cartesian semantics saves 12 transitions and entirely avoids the states $(1,2)$, $(1,1)$, $(2,1)$, $(2,2)$, as illustrated in

```
 Thread 1:           Thread 2:
0: x++              0: y++
1: x++              1: y++
2: x++              2: y++
3: assert(y≤c)      3: assert(x≤c)
4:end               4:end
```

(a)



(b)

Fig. 5. (a) A simple concurrent program, and (b) reduced state space with a cartesian semantics

Fig. 5 (b). The algorithm we propose in Sec. 6 utilizes this fact and does not explore these states and transitions.

Note that a combination of persistent sets and sleep sets will not reduce these states because sleep sets is not able to reduce states.

5 Computing Cartesian Vectors

In order to build an algorithm based on the cartesian semantics, we need the ability to calculate a cartesian vector for every observed state of the concurrent system. The algorithm `CalcCV` in Fig. 6 computes such CVs. `CalcCV` assumes that the state space is finite or acyclic.

The algorithm starts with a *minimal* CV, where each prefix contains a single transition. Such a vector necessarily satisfies Def. 2. However, for such minimal CVs, the cartesian semantics provides no benefits since it coincides with the standard semantics.

To yield longer prefixes that reduce the explored state space, the algorithm then repeatedly extends this CV with additional transitions, while still satisfying Def. 2. The array `extendable` identifies threads whose prefix can still be extended. Initially, all threads are extendable, and threads are removed from this set as conflicts are detected.

Each iteration of the `while` loop picks some extendable prefix, and tries to extend it with the next transition of that thread. Two complications arise here. First, if the added transition conflicts with the *last* transition of a different prefix, then such conflicts are allowed by Def. 2, but the algorithm records that neither prefix can be further extended.

Second, if a thread is in an infinite loop whose transitions do not conflict with concurrent threads, then that thread has an infinite prefix. To avoid diverging in such situations, the `CalcCV` algorithm avoids extending a prefix once a cycle has been detected. Instead, it marks such prefixes as being *infinite*; these marks are used by the model checking algorithm of the following section.

This cycle check guarantees that, on any finite state system, the `CalcCV` algorithm will eventually terminate, once all threads are exhausted. Indeed, this procedure actually returns a *maximal cycle-free CV*. That is, adding additional transitions to the

```
 CalcCV(s) {
  for each i ∈ 1..n do {
    CV[i] = s.NextTrans(s,T_i).nextState(s,T_i);
  }
  extendable = { 1..n }
  for each i,j ∈ 1..n such that i≠j and
      last_tran(CV[i]) is dependent with last_tran(CV[j]) {
    extendable = extendable - {i,j}
  }
  while (extendable≠∅) {       // repeatedly extend CV
    pick any i ∈ extendable
    s = last(CV[i]);
    if( $\exists j \neq i$. NextTrans(s,T_i) is dependent
        with some transition in CV[j] (other than the last)) {
      extendable = extendable - {i}
    } else {
      for each j≠i such that NextTrans(s,T_i)
          is dependent with last_tran(CV[j]) {
        extendable = extendable - {i,j}
      }
      if( NextState(s,T_i) in CV[i] and i ∈ extendable ) {
        mark CV[i] as infinite
        extendable = extendable - {i}
      }
      // add this transition to CV
      add NextTrans(s,T_i) and NextState(s,T_i) to CV[i]
    }
  }
  return CV
}
  Helper functions:
  NextTrans(s, T): return $t_{T,l}$ for $s = \langle g, ls[T \mapsto l] \rangle$
  NextState(s, T): return exec(s,NextTrans(s,T))
```

Fig. 6. Algorithm for calculating cartesian vectors.

result of CalcCV(s) yields an CV that is either invalid or contains cycles that re-visit previously-explored states.

Note that the order in which our algorithm tries to extend prefixes is arbitrary, and different exploration orders can lead to different resulting CVs. Our implementation of the algorithm uses a round-robin exploration (we did not test the effect of other exploration orders).

The correctness of the algorithm is established in the following lemma, which holds for any finite state system:

Lemma 1. *For every state s, CalcCV(s) terminates and returns a valid CV.*

The proof appears in [9].

Example 2. The following steps describe an execution of CalcCV from the initial state of the program shown in Fig. 2.

1. At the beginning, both threads are extendable, and each prefix contains only the program's initial state, where both threads are about to execute line 0.
2. T_1 executes `z:=8`, T_2 executes `q:=8`, and no conflicts are detected.
3. T_1 executes `x:=1`, T_2 executes `priv:=y`, and no conflicts are detected.
4. T_1 executes `z:=42`, T_2 executes `q:=42`, and still no conflicts are detected.

5a. The next transition of T_1 is `y:=7`, which conflicts with the previously-executed transition `priv:=y` of T_2, so this thread is no longer extendable.

5b. The next transition of T_2 is `priv:=x`, which conflicts with the previously-executed transition `x:=1` of T_1, so this thread is also no longer extendable.

At this point, the extendable set is empty, so `CalcCV` returns the cartesian vector: T_1: `z:=8; x:=1; z:=42;` T_2: `q:=8; priv:=y; q:=42;`.

Since CalcCV is called for each visited state, a key concern is the running time of this procedure. For our intended application of software model checking, we assume that each transition accesses at most one memory location, and two transitions of different threads are dependent only if they access the same memory location and that at least one of these accesses is a write. Under these assumptions, it is fairly straightforward to implement CalcCV such that its running time is proportional to the size of the resulting CV (that is, to the sum of the lengths of the prefixes in this CV). In particular, each step of the implementation either extends CV or reduces the extendable set.

6 Model Checking Algorithm

Fig. 7 presents a state exploration or model checking algorithm that explores all reachable states of the cartesian semantics, using the subroutine CalcCV to compute cartesian vectors for each reached state. Notice that only the last states of finite prefixes are added to WorkSet (according to the cartesian semantics the exploration does not have to continue from infinite prefixes).

Notice that CalcCV stops only before or after transitions that participate in a memory contention (only such transitions can be detected as dependent), therefore the reduced state space does not contain a state in which two threads (or more) are at the middle of sections without memory contentions. Therefore we can simply identify a class of states that are not present in the reduced state space. It is worth mentioning that in many large programs most of the code does not involve memory contention, therefore many states are saved by our method.

A simple variant of this algorithm executes a few instances of CalcCV in parallel (on different processors). This variant utilizes the fact that CalcCV runs independently on one processor without being affected by what happening on the other processors. Such variant can efficiently utilize a few processors and reduces the running time of the model checking, especially when the calculated CVs are long. We present the pseudo-code of this simple variant in Fig. 8, and evaluate its performance in our experiments.

```
modelCheck($s_0$) {
  WorkSet = {$s_0$}
  CoveredSet = $\emptyset$
  while WorkSet is not empty {
    select and remove s from WorkSet
    if not member(s,CoveredSet) {
      CoveredSet = CoveredSet ∪ { s }
      CV = CalcCV(s)
      for each prefix ∈ CV {
        verify local properties in states(prefix)
        if prefix is not marked as infinite
          WorkSet = WorkSet ∪ { last(prefix) }
    }}}}
```

Fig. 7. A cartesian model checking algorithm based on CalcCV

```
InitThread($s_0$)
  WorkSet = {$s_0$}
  CoveredSet = $\emptyset$
  ActiveThreads = 0
  start a worker thread for each processor
  wait until ( (WorkSet is empty) and (ActiveThreads=0) )
  terminate all worker threads

WorkerThread()
  begin:
  lock {
    if( WorkSet is empty ) goto begin
    ActiveThreads++
    select and remove s from WorkSet
    if member(s,CoveredSet)
      ActiveThreads--
      goto begin
    CoveredSet = CoveredSet ∪ { s }
  }
  CV = CalcCV(s)
  for each prefix ∈ CV
    verify local properties in states(prefix)
    if prefix is not marked as infinite
      lock { WorkSet = WorkSet ∪ { last(prefix) } }
  lock { ActiveThreads-- }
  goto begin
```

Fig. 8. A concurrent variant of the cartesian model checking algorithm

7 Experimental Evaluation

In this section, we describe preliminary experimental results comparing the cartesian algorithm to other exploration algorithms.

Table 1. Number of stored states, transitions, and running time (milliseconds.) of the cartesian and standard exploration algorithms for our benchmarks. In this table, *Conc Time* indicates the running time of the concurrent variant of the cartesian algorithm.

	Standard algorithm			Cartesian algorithm				Percentage of Saving			
Benchmark	**States**	**Transitions**	**Time (ms)**	**States**	**Transitions**	**Time (ms)**	**Conc Time (ms)**	**States**	**Transitions**	**Time**	**Conc Time**
SharedPtr	32131	64262	266	418	12785	47	32	98.7	80.1	82.3	31.9
SharedArray	2276	4552	16	132	1648	0	0	94.2	63.8	99	0
2 Robots	4877	9754	109	56	2635	15	15	98.9	73	86.2	0
3 Robots	326759	980277	1206422	56	6387	62	31	100	99.3	99	50
File System (1 Threads)	9	8	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0
File System (2 Threads)	81	144	0	1	16	0	0	98.8	88.9		0
File System (3 Threads)	729	1944	16	1	24	0	0	99.9	98.8	99	0
File System (4 Threads)	6561	23328	437	1	32	0	0	100	99.9	99	0
File System (5 Threads)	59049	262440	24047	1	40	0	0	100	100	99	0
File System (6 Threads)	531441	2834352	2567703	1	48	0	0	100	100	99	0
File System (7 Threads)				1	56	0	0				0
File System (8 Threads)				1	64	0	0				0
File System (9 Threads)				1	72	0	0				0
File System (10 Threads)				1	80	0	0				0
File System (11 Threads)				1	88	0	0				0
File System (12 Threads)				1	96	0	0				0
File System (13 Threads)				1	104	0	0				0
File System (14 Threads)				10	1026	62	32				48.4
File System (15 Threads)				100	10120	563	203				63.9
File System (16 Threads)				1000	99800	5968	2078				65.2
File System (17 Threads)				10000	984000	64204	23000				64.2
Indexer (1 Threads)	5	4	0	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Indexer (2 Threads)	25	40	0	1	8	0	0	96	80		0
Indexer (3 Threads)	125	300	0	1	12	0	0	99.2	96		0
Indexer (4 Threads)	625	2000	0	1	16	0	0	99.8	99.2		0
Indexer (5 Threads)	3125	12500	47	1	20	0	0	100	99.8	99	0
Indexer (6 Threads)	15625	75000	641	1	24	0	0	100	100	99	0
Indexer (7 Threads)	78125	437500	15297	1	28	0	0	100	100	99	0
Indexer (8 Threads)	390625	2500000	494687	1	32	0	0	100	100	99	0
Indexer (9 Threads)				1	36	0	0				0
Indexer (10 Threads)				1	40	0	0				0
Indexer (11 Threads)				1	44	0	0				0
Indexer (12 Threads)				9	394	16	16				0
Indexer (13 Threads)				81	3528	187	79				57.8
Indexer (14 Threads)				729	31590	1813	625				65.5
Indexer (15 Threads)				6561	282852	17172	6250				63.6
Indexer (16 Threads)				59049	2532546	191421	82859				56.7
2 Philosophers	11	22	0	9	28	0	0	18.2	-27.3		0
3 Philosophers	36	108	0	27	174	0	0	25	-61.1		0
4 Philosophers	119	476	0	94	750	0	0	21	-57.6		0
5 Philosophers	393	1965	16	295	2984	31	31	24.9	-51.9	-93.8	0
6 Philosophers	1298	7788	172	942	11233	187	156	27.4	-44.2	-8.7	16.6
7 Philosophers	4287	30009	1766	2955	41091	1187	969	31.1	-36.9	32.8	18.4
8 Philosophers	14159	113272	29594	9212	145717	11609	11141	34.9	-28.6	60.8	4
9 Philosophers	46764	420876	383219	28675	509218	132078	138703	38.7	-21	65.5	-5
CMIS C=2 N=8	16430	115010	813	51	1627	32	15	99.7	98.6	96.1	53.1
CMIS C=4 N=16	1014131	7098917	10294344	51	3091	47	31	100	100	99	34
CMIS C=8 N=32				51	8035	156	62				60.3
CMIS C=16 N=64				51	25987	735	281				61.8
CMIS C=32 N=128				51	94147	4875	1719				64.7
CMIS C=64 N=256				51	359491	36531	17672				51.6
CMIS C=128 N=256				6	100336	12141	12250				-0.9
CMIS C=127 N=255				11	221954	27860	27328				1.9

We compared the number of states, transitions, and CPU time measured by a standard model checking algorithm (exhaustive exploration without partial order reduction) and by the cartesian algorithm of Fig. 7. The comparison was done for a few benchmark programs, and the results are reported in Table 1. The number of states mentioned in the results is the number of states that the algorithm stores during its execution (i.e. the size of CoveredSet when the algorithm terminates). An empty cell in the table indicates that the algorithm ran out of memory. Additional results and details about the benchmarks can be found in the appendix.

In order to check dependency between transitions, the implementation of the cartesian algorithm conservatively assumes that two transitions are dependent if they have conflicting memory accesses (i.e., one writes and the other reads or writes from the same location). During the execution of CalcCV, the algorithm remembers the memory locations accessed by each thread (in the current CalcCV execution) and uses this information for determining dependency between transitions.

The benchmarks were also tested on SPIN [11], but its partial order reduction algorithm was unable to reduce the state space of any of the benchmarks (i.e. SPIN's partial order reduction did not affect the numbers of states and transitions).

Some of the acyclic benchmarks were tested on the dynamic partial order reduction algorithm from [6] (hereafter, referred to as FG). Because FG is stateless we only compared the number of transitions. For some acyclic benchmarks, the cartesian algorithm executed much fewer transitions than FG, even when FG was combined with sleep sets [8] (e.g. for the SharedArray benchmark, the cartesian algorithm executed only 1648 transitions whereas FG executed more than 10^7 transitions). For some other acyclic benchmarks such as FileSystem, FG executed less transitions than the cartesian algorithm, but in these cases the differences were less significant.

We also implemented the concurrent variant of the cartesian algorithm mentioned in Sec. 6 and run the benchmarks on it using a machine with 4 processors. In some cases (Indexer, FileSystem, CMIS) it saved around 60% of the running time (comparing to the sequential variant).

8 Related Work

A key limitation in model checking concurrent software systems [2] is the notorious state explosion problem. One approach to this problem is to reduce the size of the state space via *abstraction* [4] and abstraction refinement [1,10,3] techniques. A complementary approach is to only explore a (sufficiently large) fraction of the system's state space, via *partial order reduction* techniques.

One standard partial order reduction technique is based on *persistent (or stubborn) sets* [18,8]. This technique computes a subset of the enabled transitions in each visited state, and only explores those transitions. This computed subset is called a *persistent set*, and contains sufficiently many transitions to guarantee certain completeness properties. Our approach can yield improvements even over the most precise persistent sets.

A traditional limitation of persistent sets is that they are typically obtained from a static analysis of the code, via algorithms such as those described in [8]. Hence, the approximations inherent in any static analysis can result in coarse persistent sets, particularly for pointer-rich code. Our algorithm overcomes this limitation by detecting conflicts between transitions dynamically, instead of statically.

The approach of dynamic partial order reduction [6] computes persistent sets on-the-fly by detecting conflicts dynamically, but only performs a stateless search, and extending it to a stateful search has proven quite difficult. In contrast, the algorithm of this paper performs a stateful search, which provides two key improvements over [6]: (1) it can handle systems with cycles; and (2) even on cycle-free systems, storing states avoids repeated explorations of the same parts of the state space.

A number of recent techniques have considered various kinds of *exclusive access predicates* for shared variables that specify synchronization disciplines such as "this variable is only accessed when holding its protecting lock" or "this variable is local to this thread" [15,16,5,7]. These exclusive access predicates can be leveraged to dynamically infer persistent transitions, and so reduce the search space. At the same time, exclusive access predicates can be verified or inferred during reduced state-space exploration. These techniques of [5,16] in particular have demonstrated significant performance improvements for the common cases of thread-local and lock-protected data. However, these techniques are less effective when the synchronization discipline changes during program execution, such as when an object is protected by different variables at different stages during the program's execution.

9 Conclusions

We have presented a new approach *Cartesian* approach to partial order reduction that can be used by model checkers and abstract interpreters. We are encouraged by the empirical results that show improvement over prior approaches for some benchmarks.

References

1. Ball, T., Rajamani, S.: The SLAM Toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
2. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
3. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Robby, C.S.P., Zheng, H.: Bandera: Extracting Finite-State Models from Java Source Code. In: Proceedings of the 22nd International Conference on Software Engineering (2000)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Symp. on Principles of Prog. Languages, pp. 269–282. ACM Press, New York, NY (1979)
5. Dwyer, M.B., Hatcliff, J., Prasad, V.R., Robby,: Exploiting Object Escape and Locking Information in Partial Order Reduction for Concurrent Object-Oriented Programs. Formal Methods in System Design 25(2–3) (2004)
6. Flanagan, C., Godefroid, P.: Dynamic Partial-Order Reduction for Model Checking Software. In: Proceedings of POPL'2005, 32nd ACM Symposium on Principles of Programming Languages, Long beach (January 2005)
7. Flanagan, C., Qadeer, S.: Transactions for Software Model Checking. In: Proceedings of the Workshop on Software Model Checking, pp. 338–349 (June 2003)
8. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
9. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian partial-order reduction. Technical Report TA-CS-2007-052, School of Computer Science, Tel Aviv University (2007) Available at `http://www.cs.tau.ac.il/~guygueta/Cartesian.pdf`

10. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proc. of the 29th ACM Symposium on Principles of Programming Languages, Portland, pp. 58–70. ACM Press, New York (2002)
11. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual
12. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Proceedings of the 7th IFIP WG6 International Conference on Formal Description Techniques VII, pp. 197–211. Chapman & Hall Ltd, London, UK (1995)
13. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
14. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
15. Stoller, S.D.: Model-Checking Multi-Threaded Distributed Java Programs. International Journal on Software Tools for Technology Transfer 4(1), 71–91 (2002)
16. Stoller, S.D., Cohen, E.: Optimistic Synchronization-Based State-Space Reduction. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, pp. 489–504. Springer, Heidelberg (2003)
17. Valmari, A.: Stubborn sets for reduced state space generation. In: 10th Conference on Applications and Theory of Petri Nets, pp. 491–515 (1991)
18. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) Advances in Petri Nets 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)
19. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. In: Proc. Symp. on Principles of Prog. Languages, pp. 27–40. ACM Press, New York (2001)

A Benchmarks Description

In this appendix we describe the benchmarks.

A.1 Robots

The Robots example shown in Fig. 1. This program simulates an arena with a number of robots that move in different paths, where each robot is represented by a separate thread. Approaches based on static dependence will not be able to determine when a collision is possible, and would yield a poor reduction of the state space. The dynamic partial order reduction of [6] is not applicable for this benchmark, as its statespace contains cycles.

For this benchmark, we consider two configurations: one that uses 2 robots, as shown in Fig. 1, and one with 3 robots in which a new robot is added and set to start from position $(7, 0)$.

Table 1 shows that for both configurations (2 robots, and 3 robots), the cartesian algorithm provides a significant improvement over the standard semantics.

A.2 CMIS

CMIS is a concurrent sorting algorithm which is composed from Merge-Sort and Insert-Sort, its pseudo code appears in Fig. 9. In Table 1, C indicates an array length from which CMIS uses a sequential Insert-Sort (see pseudo code), N indicates the length of the array. In all the cases the input was an array sorted in a descending order (CMIS

```
ConcurrentMergeInsertSort(A, p, r) {
  if( r-p+1 $\leq$ C )
    InsertSort(A, p, r);
  else {
    q = $\lfloor \frac{p+r}{2} \rfloor$ ;
    run ConcurrentMergeInsertSort(A, p, q) on a child thread ;
    ConcurrentMergeInsertSort(A, q+1, r);
    wait for child thread termination ;
    Merge(A, p, q, r);
  }
  Assert(A is sorted) ;
}

InsertSort(A, p, r) {
  for j = p+1 to r {
    key = A[j];
    i = j - 1 ;
    while ((i > p-1) and (A[i] > key)) {
      A[i+1] = A[i];
      i--;
    }
    A[i+1] = key ;
  }
}

Merge(A, p, q, r) {
  for i = p to r
    draft[i] = A[i] ;
  i = p; j = q+1; k = p;
  while ((i $\leq$ q) and (j $\leq$ r)) {
    if( draft[i] $\leq$ draft[j] )
      A[k++] = draft[i++];
    else
      A[k++] = draft[j++];
  }
  while (i $\leq$ q)
    A[k++] = draft[i++];
}
```

Fig. 9. The CMIS (Concurrent-Merge-Insert-Sort) benchmark

sorted the array in an ascending order). Our approach does not deal with dynamic thread creation therefore we simulated the dynamic threads creation by using threads that wait on a loop until they receive an appropriate request.

A.3 SharedArray

The code of the SharedArray benchmark is shown in Fig. 10. In this program, there are two threads writing to a shared array in a loop. Each of the threads accesses different

```
N = 64;
int A[N];
int idx$_0$ = 0, idx$_1$ = 1,counter = 1;
Thread $i$ ($i = 0, 1$)
  While( idx$_i$ < N) atomic {
      A[idx$_i$]=counter + idx$_i$;
      idx$_i$ += 2 ;
  }
  atomic {
    counter = counter + 1 + idx$_{1-i}$ ;
    assert(counter $\leq$ 2*N + 4) ;
  }
```

Fig. 10. SharedArray Example

portions of the array. In every iteration of the loop each thread reads the value of a shared variable $counter$ and updates the array using its value. After finishing the loop each thread updates the value of the shared variable $counter$. The instructions within the atomic blocks (marked by the keyword atomic) are executed together atomically.

Partial order reduction algorithms based on persistent sets will not be able to reduce the state space of this program. This is due to the fact that in every state in which the two threads are still running, every persistent set contains all enabled transitions.

A.4 SharedPtr

The code for the SharedPtr benchmark is shown in Fig. 11. In this benchmark, two threads are performing updates to memory locations identified using a shared pointer p.

The behavior of this example is similar to that of the SharedArray example, in the sense that the threads sometimes access disjoint parts of memory, but in a way that a static partial order reduction approach will not be able to detect.

```
N = 100;
int x=3, y=4, c1=0, c2=0
int* p
Thread 1
  p = &y;
  for(int i=0; i < N; i++) c1 += x;
  *p += 3;
  assert($3 \leq x, y \leq 9$);
Thread 2
  p = &x;
  for(int i=0; i < N; i++) c2 += y;
  *p += 2;
  assert($3 \leq x, y \leq 9$);
```

Fig. 11. SharedPtr Example

```
const int size = 128;
const int max = 4;
int[size] table;
int m = 0, w, h;
Thread tid
  while (true) {
    w := getmsg();
    h := hash(w);
    while (cas(table[h],0,w) == false) {
      h := (h+1) % size;
    }
  }
  int getmsg() {
    if (m < max ) {
      return (++m) * 11 + tid;
    } else {
      exit(); // terminate
    }
  }
  int hash(int w) {
    return (w * 7) % size;
  }
```

Fig. 12. Indexer Example (from [6])

A.5 Indexer

This example is taken from [6]. This example has no cycles and behaves well with a persistent sets algorithm. In this benchmark, there are no collisions between the threads when the number of threads is less than 12. As a result, the cartesian algorithm is able to considerably reduce the number of transitions when using up to 11 threads. In contrast, the standard exploration suffers from exponential increase in the number of transitions. Notice that in some cases the number of stored states is 1, this is reasonable because in these cases the threads have no conflicts between them.

A.6 File System

This example is also taken from [6]. It uses up to 17 threads that communicate via a shared memory. The properties of this example are similar to those of the Indexer example.

A.7 Dining Philosophers

This example is the classical dining philosophers program.