Static Partial-Order Reduction of Concurrent Systems in Polynomial Time

Robert Mittermayr and Johann Blieberger

Institute of Computer-Aided Automation
Vienna University of Technology
Austria

Abstract. We present an algorithm for attacking the state explosion problem in analyzing multithreaded programs. Our approach employs partial-order reduction and static virtual coarsening. It uses information on shared variables to generate and interleave blocks of statements.
Our algorithm performs polynomially as long as the number of shared variables is constant.

1 Introduction

With the advent of multi-core processors scientific and industrial interest focuses on multithreaded applications. Examples like [3] show that writing even small multithreaded programs can be a tedious task.

For safety-critical systems or robust embedded systems, software has to be provably correct. So, in order to incorporate concurrently executing threads in safety-critical systems, these concurrent programs have to be proved to be correct. Verifying concurrent programs is challenging because the number of thread interleavings grows exponentially in the number of statements of the program. All state-of-the-art methods, such as model-checking, suffer from this so-called *state explosion problem*.

The main contributions of this paper are a theoretical analysis of interleavings, an algorithm for the static reduction of interleavings needed to be taken into account in order to generate the state space, and a worst-case estimation of this algorithm.

The remainder of the paper is organized as follows. In Section 2 the state explosion problem is discussed theoretically. An algorithm for reducing the amount of interleavings without losing any computational results is presented in Section 3. The worst-case behavior of the presented algorithm is presented in Section 4. In Section 5 an example shows how the algorithm works. Related work is discussed in Section 6. Finally, we conclude the paper and outline possible future work in Section 7.

2 Interleavings and the State Explosion Problem

For the analysis of multithreaded software in general it is very important to analyze all possible execution sequences. This ensures that each possible state is reached. In this section we will start with an example which introduces the problem in practice. The simple example will be followed by a theoretical analysis of the state explosion problem.

As a motivating example consider

$$P\colon (\underbrace{x{:=}4}_{a};\underbrace{x{:=}x+3}_{b}) \parallel (\underbrace{x{:=}2}_{c};\underbrace{x{:=}(x*x)+1}_{d}) \; .$$

Program P may result in six states. All possible final states of program P are depicted in Table 1. Please note that we define all statements in this paper to be atomic.

Order	x
a b c d	5
a c b d	26
a c d b	8
c a b d	50
c a d b	20
c d a b	7

Table 1. Computation results

For enumerating the number of interleavings for n threads $(t_1, t_2, \ldots, t_n)$ where each t_i has k_i statements (where $1 \leq i \leq n$) the multinomial theorem can be applied. This results in

$$\binom{k_1+k_2+k_3+\cdots+k_n}{k_n}\cdots\binom{k_1+k_2+k_3}{k_3}\cdot\binom{k_1+k_2}{k_2} = \frac{(k_1+k_2+k_3+\cdots+k_n)!}{k_1!k_2!k_3!\ldots k_n!}$$

interleavings.

Lemma 1 (Number of Interleavings). *Given n threads $(t_1, t_2, \ldots, t_n)$ where each t_i has k_i statements, the number of interleavings is given by*

$$\frac{\left(\sum_{i=1}^{n} k_i\right)!}{\prod_{i=1}^{n} k_i!} \, . \tag{1}$$

□

Lemma 1 shows how simple it is to get astronomically high numbers of interleavings. If we have 20 statements in each of three threads we get $\frac{(60!}{(20!)^3} \approx 5\times 10^{26}$ interleavings.

In order to find the maximum of Eq. (1) we have to maximize

$$\frac{k!}{\prod_{i=1}^{n} k_i!} \tag{2}$$

where $k = \sum_{i=1}^{n} k_i$. In the following we use the gamma function $\Gamma(x)$ (cf. [1]) to replace the integer factorial by a real-valued function. Note that $\Gamma(m+1) = m!$.

In order to find the extreme value of Eq. (2) we employ the logarithmic derivative of Eq. (2) which simplifies the calculations significantly.

The derivative of

$$log(k!) - \sum_{i=1}^{n} log(\Gamma(k_i + 1)) + \lambda(\sum_{i=1}^{n} k_i - k)$$

with respect to k_i is

$$-\frac{\Gamma'(k_i + 1)}{\Gamma(k_i + 1)} + \lambda = 0,$$

where $1 \le i \le n$. This is valid for all k_i, in particular for $i = s$ and $i = t$, i.e.,

$$\frac{\Gamma'(k_s + 1)}{\Gamma(k_s + 1)} = \frac{\Gamma'(k_t + 1)}{\Gamma(k_t + 1)}.$$

Using the digamma function $\psi(x) = \frac{\Gamma'(x)}{\Gamma(x)}$ (cf. [1]) we can write

$$\psi(k_s + 1) = \psi(k_t + 1).$$

Because $\psi(x)$ is monotonically increasing for $x \ge 0$ (cf. e.g. [1]) we get

$$k_s = k_t, \text{ for all } 1 \le s, t \le n$$

which implies $k_i = k/n$, provided that n divides k $(n|k)$.

Thus we have proved the following lemma.

Lemma 2. *For a given number of statements k, the worst-case number of interleavings appears if all n threads have the same number of statements. In this case the number of interleavings is given by*

$$\frac{k!}{\left((\frac{k}{n})!\right)^n} \tag{3}$$

where $n|k$. □

In the following we write $k = \beta \cdot n$. If the number of statements per thread $\beta \ge 1$ is fixed, Formula (3) can be estimated by Stirling's approximation $m! = (\frac{m}{e})^m \sqrt{2\pi m}(1 + O(\frac{1}{m}))$ as $m \to \infty$ (cf. e.g. [1]) giving

$$\frac{(\beta\, n)!}{(\beta!)^n} \sim n^{\beta\, n + \frac{1}{2}}\, (2\pi\beta)^{-\frac{n}{2}+\frac{1}{2}},\ (n \to \infty). \tag{4}$$

For $\beta \in \{5, 10, 15, 20, 25, 30\}$ the characteristics of this formula are depicted in Fig. 1. This case describes the practical case when there is e.g. an Ada task type defining tasks with β statements.

In Fig. 2 the behavior of a variable number of statements per thread $2 \le \beta \le 200$ and a variable number of threads $1 \le n \le 40$ is depicted in logarithmic scale. Please note that the functions depicted in Fig. 1 and 2 are actually defined for natural numbers only. The same applies for the figures in Section 4.

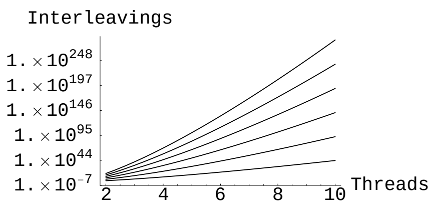


Fig. 1. Interleavings for fixed number of statements per thread

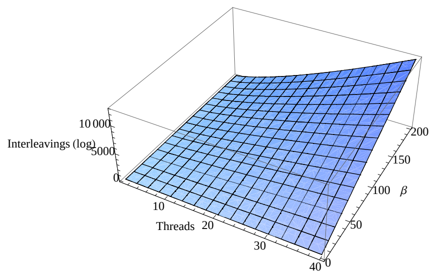


Fig. 2. Interleavings for a variable number of threads and statements per thread

3 Algorithm

In this section we present an algorithm for reducing the amount of interleavings without losing possible resulting states. We achieve this by building blocks of statements. We combine a *virtual coarsening* approach similar to [2, 16] with a partial order relation of blocks of statements. The correctness of virtual coarsening has been proved in [2].

We say that a statement *accesses* a variable v if the statement reads or writes the variable v. A variable is said to be shared if more than one thread accesses it. The set of shared variables is denoted by V. We refer to the set of threads with T.

Definition 1. *A* block *is a list of consecutive statements and contains at most one statement accessing shared variables. If the number of blocks is minimized, we call the resulting list of consecutive blocks* minimum block list.

We define a strict partial order relation "<" between blocks that is irreflexive, asymmetric, and transitive. Let bl_i and bl_j be blocks. Then $bl_i < bl_j$ holds iff there exists an execution sequence of the underlying program such that bl_i preceeds bl_j.

We distinguish between two kinds of strict partial order relations, intra-thread and inter-thread orders. Intra-thread orders define orders of blocks within a thread and inter-thread orders define an ordering between blocks of different threads.

Both relations can be represented by graphs. We call such graphs *partial order graphs* or *PO graphs* for short.

Let $t(bl_i) = t_j$ when bl_i is part of thread t_j, where $1 \leq j \leq n$ and $1 \leq i \leq b$. Let $a(v_i) = \{bl_j \mid \exists \textit{ statement } s \textit{ in } bl_j \textit{ which accesses } v_i\}$. Let $a(v_i, t_j) = \{bl_k \mid \exists \textit{ statement } s \textit{ in } bl_k \textit{ which accesses } v_i \textit{ and } t(bl_k) \neq t_j\}$. In addition, we define $SV(bl_i) = \{v_j \mid \text{where block } bl_i \text{ contains a statement which accesses the shared variable } v_j\}$. Let the number of shared variables be r and further let $b^{(v_j)}$ denote the number of blocks accessing shared variable v_j (over all threads).

The algorithm consists of the following six steps:

Find shared variables. This step fills the set V.

Build minimum block list.

Generate partial orders for the blocks accessing the same shared variables.

Construct PO graphs

Apply topological sorting to each PO graph. This step results in exactly one order of blocks for any PO graph constructed in the previous step.

Compute the state space of the program by executing the blocks in the orders calculated in the previous step.

The algorithm is presented in more detail below.

```
ALGORITHM ()
List[ ] blocks := new List[1..n]
List[ ][ ] blocks_var := new List[1..r][1..n]
List[ ] finalInterThreadOrderedBlocks
ListOfListOfPartialOrders combinedInterThreadOrderedBlocks
FINDSHAREDVARIABLES()
BUILDBLOCKS()
GENERATEORDERS()
CONSTRUCTPOGRAPHS()
TOPOLOGICALSORTING()
COMPUTESTATESPACE()
```

```
GENERATEORDERS ()
GENERATEINTRATHREADORDERS
GENERATEINTERTHREADORDERS
COMBINEINTERTHREADORDERS
```

```
GENERATEINTRATHREADORDERS ()
for each thread $t_i \in T$ do
   for each pair $bl_j, bl_{j+1} \in blocks[i]$ do
      DEFINE $bl_j < bl_{j+1}$
   endfor
endfor
```

```
BUILDBLOCKS ()
for each thread $t_i \in T$ do
   boolean firstBlock := true
   Block actualBlock := new Block()
   blocks[i].add(actualBlock)
   for each statement $s \in t_i$ do
      if $s$ accesses $v \in V$ then
         if not firstBlock then
            actualBlock := new Block()
            blocks[i].add(actualBlock)
         else
            firstBlock := false
         endif
      endif
      append $s$ to actualBlock
   endfor
endfor
```

```
GenerateInterThreadOrders ()
int b_var_k
for each thread $t_i \in T$ do
   for each $bl_j \in blocks[i]$ do
      for each $v_k \in SV(bl_j)$ do
         $blocks\_var[k][i].add(bl_j)$
      endfor
   endfor
endfor
for each $v_k \in V$ do
   $b\_var\_k := b^{(v_k)}$
   Block[ ] interThreadOrderedBlocks := new Block[1..b_var_k]
   Interleave_rec(blocks_var[k], interThreadOrderedBlocks, 1, k)
endfor
```

```
Interleave_rec (List[ ] blks, Block[ ] interThrdOrdBlks, int block_nr, int var_nr)
for each $i \in \{1 \dots n\}$ do
   if blks[i].count() > 0 then
      interThrdOrdBlks[block_nr]:=blks[i].getHead()
      if block_nr = b_var_k then
         finalInterThreadOrderedBlocks[var_nr].add(new List(interThrdOrdBlks))
      else
         List[ ] local_blks:=blks.clone()
         local_blks.removeHead()
         interleave_rec(local_blks, interThrdOrdBlks, block_nr + 1, var_nr)
      endif
   endif
endfor
```

```
CombineInterThreadOrders ()
GenerateCombinations_rec(0, new ListOfPartialOrders())
```

```
GenerateCombinations_rec (int cur_var, ListOfPartialOrders ordersTillNow)
for each interThreadOrderedBlocks ∈
      finalInterThreadOrderedBlocks[cur_var] do
   ListOfPartialOrders ordersToUse := ordersTillNow.clone()
   for each pair $bl_i, bl_{i+1} \in$
         finalInterThreadOrderedBlocks[cur_var][interThreadOrderedBlocks] do
      Define $bl_i < bl_{i+1}$ and add it to ordersToUse
   endfor
   if $cur\_var < r$ then
      GenerateCombinations_rec(cur_var+1, ordersToUse)
   else
      combinedInterThreadOrderedBlocks.add(ordersToUse)
   endif
endfor
```

```
ConstructPOGraphs ()
1 for each $concreteOrders \in combinedInterThreadOrderedBlocks$ do
2     use PO graph generated from intra-thread orders for a new PO graph
3     for each $PO(b_i, b_j) \in concreteOrders$ do
4         add directed edge from $node_{b_i}$ to $node_{b_j}$
5     endfor
6     add PO graph to the set of resulting PO graphs
7 endfor
```

Each of the steps (except the first one) uses information generated in the previous steps. Note that we generate the reduced state space without generating the original state space as an intermediate result. This is of course important because if the original state space would be generated it would abandon the achieved reductions. In addition, note that our approach, in its current version, needs no human modeling or specification input.

The initial exponential growth of interleavings in terms of the number of statements can be reduced to an exponential growth in terms of the number of shared variables. In a lot of cases this approach enables static analysis of multithreaded programs.

If a statement accesses two or more different shared variables, it may happen that conflicting partial order relations appear. If e.g. a statement s_1 in block bl_1 which is part of thread t_1 reads variable x and writes variable y, whereas statement s_2 in block bl_2 which is part of thread t_2 writes the shared variable x and reads the variable y then the algorithm generates the following inter-thread orders

$$\begin{pmatrix} bl_1 < bl_2 \\ bl_2 < bl_1 \end{pmatrix} \times \begin{pmatrix} bl_1 < bl_2 \\ bl_2 < bl_1 \end{pmatrix}.$$

This results in a cyclic directed graph[1]. Topological sorting can detect this and the algorithm can be aborted for such contradictory partial orders. In the above example the algorithm aborts two times and only the two orders $bl_1; bl_2$ and $bl_2; bl_1$ are being generated.

The presented algorithm acts on the assumption that all threads are running at the same time. In addition, it is assumed that every statement s_1 of a thread t_1 may happen in parallel to every statement s_2 of another thread t_2 (cf. e.g. [4]). This assumptions assure the completeness of the approach. On the other hand this conservative approach may lead to false positives.

Please note that currently we are not handling conditionals, loops, and procedure calls. This will be future work.

4 Worst-Case Analysis

In this section we assume that each statement only accesses one single shared variable. Note however that this is no real constraint because statements accessing several shared variables can be replaced by simpler statements accessing only one single shared variable by introducing artificial (local) variables.

[1] Note that duplicate partial orders can be ignored during PO graph construction.

In addition to the definitions in previous sections let $b_i^{(v_j)}$ denote the number of blocks in thread i accessing shared variable v_j.

To derive the worst-case complexity of the approach from Section 3, a multivariate extreme value calculation can be employed. The following expression, which denotes the number of graphs generated, has to be maximized

$$\prod_{j=1}^{r} \sum_{i=1}^{n} b_i^{(v_j)} \left(b^{(v_j)} - b_i^{(v_j)} \right). \tag{5}$$

In addition, we have the following constraint

$$\sum_{i=1}^{n} b_i^{(v_j)} = b^{(v_j)}.$$

Differentiating

$$\prod_{j=1}^{r} \sum_{i=1}^{n} b_i^{(v_j)} \left(b^{(v_j)} - b_i^{(v_j)} \right) + \lambda_1 \left(\sum_{i=1}^{n} b_i^{(v_j)} - b^{(v_j)} \right)$$

with respect to $b_i^{(v_j)}$, we get for all $1 \le i \le n$ and $1 \le j \le r$

$$b^{(v_j)} - 2\, {b_i}^{(v_j)} + \lambda_1 = 0. \tag{6}$$

Summing up (6) for $i = 1, 2, \ldots, n$, we have $n\, b^{(v_j)} - 2\, b^{(v_j)} + n\, \lambda_1 = 0$, which implies $\lambda_1 = \frac{2-n}{n}\, b^{(v_j)}$. Inserting this into (6), we obtain

$$b_i^{(v_j)} = \frac{b^{(v_j)}}{n}. \tag{7}$$

Inserting (7) into (5), we get

$$\prod_{j=1}^{r} b^{(v_j)} \left(b^{(v_j)} - \frac{b^{(v_j)}}{n} \right) = \prod_{j=1}^{r} \left[\left(b^{(v_j)} \right)^2 \left(1 - \frac{1}{n} \right) \right] =$$

$$\left(1 - \frac{1}{n} \right)^r \prod_{j=1}^{r} \left(b^{(v_j)} \right)^2 \tag{8}$$

which has to be maximized under the constraint

$$\sum_{j=1}^{r} b^{(v_j)} = b. \tag{9}$$

Differentiating

$$\left(1 - \frac{1}{n} \right)^r \prod_{j=1}^{r} \left(b^{(v_j)} \right)^2 + \lambda_2 \left(\sum_{j=1}^{r} b^{(v_j)} - b \right)$$

with respect to $b^{(v_j)}$ we obtain

$$\left(1-\frac{1}{n}\right)^r 2\,b^{(v_j)} \prod_{\substack{1\le l\le r\\ l\neq j}} \left(b^{(v_l)}\right)^2 + \lambda_2 = 0$$

which implies

$$\lambda_2 = -\frac{2}{b^{(v_j)}}\left(1-\frac{1}{n}\right)^r \prod_{l=1}^{r} \left(b^{(v_l)}\right)^2 . \tag{10}$$

Now, (10) is valid for all v_j, in particular for $j = s$ and $j = t$, i.e.,

$$\frac{2}{b^{(v_s)}}\left(1-\frac{1}{n}\right)^r \prod_{l=1}^{r} \left(b^{(v_l)}\right)^2 = \frac{2}{b^{(v_t)}}\left(1-\frac{1}{n}\right)^r \prod_{l=1}^{r} \left(b^{(v_l)}\right)^2$$

which implies $b^{(v_s)} = b^{(v_t)}$ for all s, t. By using Eq. (9) we get

$$b^{(v_j)} = \frac{b}{r}. \tag{11}$$

By inserting Eq. (11) into Eq. (8) we have proved the following theorem.

Theorem 1 (Number of PO Graphs). *The number of PO graphs with b nodes for n threads and r shared variables is bounded above by*

$$\prod_{j=1}^{r}\left(\left(\frac{b}{r}\right)^2\left(1-\frac{1}{n}\right)\right) = \left(\frac{b}{r}\right)^{2r}\left(1-\frac{1}{n}\right)^r \le \left(\frac{b}{r}\right)^{2r} . \tag{12}$$

□

From Theorem 1 we know an upper bound of the number of graphs with b nodes. Because topological sorting hast to be applied for every graph our algorithm has the following worst-case behavior

$$O\left(b\left(\frac{b}{r}\right)^{2r}\left(1-\frac{1}{n}\right)^r\right). \tag{13}$$

This shows that if $r = O(1)$ the algorithm behaves polynomially.

A simple computation shows that the extreme value of (13) appears if

$$r = \frac{b}{e}\sqrt{1-\frac{1}{n}},$$

where e denotes the base of the natural logarithm.

Corollary 1 (Worst-Case Value of r). *For a given pair (n, b), Eq. (12) has its maximum at $r = (b/e)\cdot\sqrt{1-1/n}$. For $n \to \infty$ this results in $r \to b/e$.* □

By using Corollary 1 the extreme value of (13) is bounded above by

$$O\left(b\left(\frac{e}{\sqrt{1-\frac{1}{n}}}\right)^{2\frac{b}{e}\sqrt{1-\frac{1}{n}}}\left(1-\frac{1}{n}\right)^{\frac{b}{e}\sqrt{1-\frac{1}{n}}}\right)=$$

$$O\left(b\left(e^{\frac{2}{e}}\right)^{b\sqrt{1-\frac{1}{n}}}\right)=O\left(b\left(e^{\frac{2}{e}}\right)^{b}\right).$$

Hence we summarize our results in the following theorem.

Theorem 2. *The worst-case timing behavior of the algorithm presented in Section 3 is*

$$O\left(b\left(e^{\frac{2}{e}}\right)^{b}\right)$$

where b denotes the number of blocks of the underlying program and $e^{\frac{2}{e}} = 2.087\ldots$. □

This concludes that our approach from Section 3 behaves exponentially in the worst-case.

For the remaining part of this section it is assumed that each block consists of 5 statements. In Fig. 3 the curves of Fig. 2 and our worst-case are compared. The achieved reduction of the complexity is for $n = 20$ and $\beta = 200$ at least 10^{4923}. Please keep in mind that this is still the worst-case behavior of our algorithm where $r = (b/e) \cdot \sqrt{1 - 1/n}$, i.e., for practical settings we expect an even higher reduction.

Large values of n and/or b lead to more reduction. Figure 4 depicts the reduction in the worst-case for $\beta = 50$ in dependence of n ranging from 1 to 20. Figure 5 shows the achieved reduction in worst-case settings for $n = 20$ in dependence of β ranging from 1 to 200.

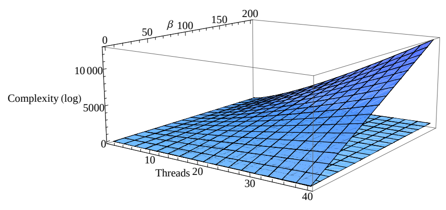


Fig. 3. Comparison

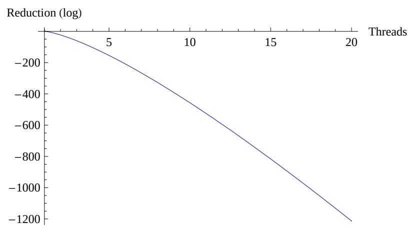


Fig. 4. Reduction for 50 blocks per thread

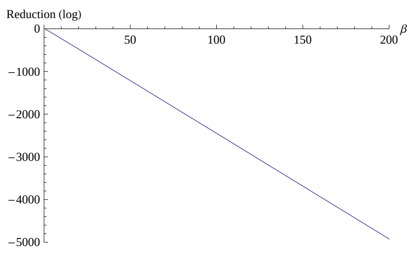


Fig. 5. Reduction for 20 threads with each β blocks

5 Example

If each of the three threads in Figure 6 has 20 statements then we have $\beta = 20$ and $n = 3$. The possible interleavings on a statement level (cf. Eq. (4)) are $\frac{(\beta\, n)!}{(\beta!)^n} = \frac{60!}{(20!)^3} \approx 5 \times 10^{26}$.

If only the shown statements access the shared variables `e` and `f` we get with our algorithm one block for each of the first two threads, namely bl_1 and bl_2, respectively. Because the third thread accesses both of the two variables two blocks are being generated, namely bl_3 and bl_4.

By interleaving these blocks (cf. Lemma 1) we obtain $\frac{4!}{1!1!2!} = 12$ different interleavings. Although this is already an enormous reduction, it can still be improved. This is due to the fact that only orders of blocks concerning the same shared variables are relevant. We express this by using the notation of inter-thread orders.

Only the orders of bl_1 and bl_3 and, similarly, the orders of bl_2 and bl_4 are relevant. To express all possible combinations GenerateInterThreadOrders of our algorithm generates

$$\begin{pmatrix} bl_1 < bl_3 \\ bl_3 < bl_1 \end{pmatrix} \times \begin{pmatrix} bl_2 < bl_4 \\ bl_4 < bl_2 \end{pmatrix}.$$

```
P: int e := 0;
   int f := 0;                                  thread {
   thread {               thread {                     .
              .                  .                     .
              .                  .                     .
              .                  .                  e := 2;
         e := 1;              f := 1;               f := 2;
              .                  .                     .
              .                  .                     .
              .                  .                     .
   }                      }                     }
```

Fig. 6. Example with 3 threads

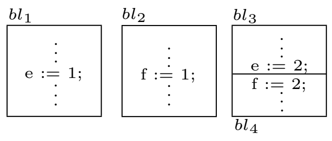
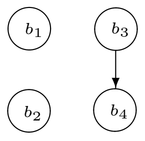


Fig. 7. Blocks for Example 6

The third thread contains an intra-thread order $bl_3 < bl_4$. Because of this the initial PO graph for every concrete interleaving (in this example) looks like the one depicted in Fig. 8(a). In the following this graph is now used as a basis for



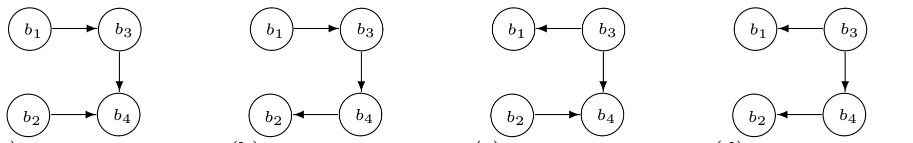
(a) Intra-thread orders

ON	inter-tread order
1	$bl_1 < bl_3, bl_2 < bl_4$
2	$bl_1 < bl_3, bl_4 < bl_2$
3	$bl_3 < bl_1, bl_2 < bl_4$
4	$bl_3 < bl_1, bl_4 < bl_2$

(b) Inter-thread orders

Fig. 8. Partial orders

every PO graph constructed by our algorithm. For every pair of the above inter-thread orders, a PO graph is being constructed. Table 8(b) shows the resulting partial orders with an assigned order number (ON) for ease of reference. By adding the partial orders of one order number to the PO graph in Fig. 8(a) results in a new graph. The four different PO graphs shown in Fig. 9 are being generated if this is done for every line in Table 8(b).



(a) Order Number 1 (b) Order Number 2 (c) Order Number 3 (d) Order Number 4

Fig. 9. PO Graphs

Sorting the nodes in each PO graph using *topological sorting* leads to four computations. Each computation gives a unique result[2]. This means we get four interleavings which compute all possible results for the shared variables `e` and `f`. For the PO graphs from Fig. 9(a), 9(b), 9(c), and 9(d) we get $bl_1; bl_3; bl_2; bl_4$, $bl_1; bl_3; bl_4; bl_2$, $bl_3; bl_1; bl_2; bl_4$, and $bl_3; bl_1; bl_4; bl_2$, respectively. The results are

[2] Unless two different computations result in an identical state by chance.

shown in Table 2. Please note that some orders of blocks depend on how topological sorting is implemented, in particular, when there are two or more possible block arrangements[3]. In this case two or more interleavings build an equivalence class (cf. [2]). For verification purposes only one exemplar in this equivalence class needs to be computed. For example topological sorting of the PO graph in Fig. 9(a) can also result in $bl_1; bl_2; bl_3; bl_4$. This is due to the fact that the order of bl_2 and bl_3 is irrelevant concerning the resulting state of the shared variables `e` and `f`. With our approach exactly one of the possible interleavings is being computed. This helps to achieve an enormous reduction in the number of interleavings.

ON	Order	e	f
1	$bl_1; bl_3; bl_2; bl_4$	2	2
2	$bl_1; bl_3; bl_4; bl_2$	2	1
3	$bl_3; bl_1; bl_2; bl_4$	1	2
4	$bl_3; bl_1; bl_4; bl_2$	1	1

Table 2. Computation Results

6 Related Work

Early reduction algorithms can be found in [14]. In [18] (Chapters 6 and 7) Valmari gives a good survey of models and approaches used so far. Due to space limitations we cite only some fundamental approaches and techniques in the following.

Virtual Coarsening. The idea is that in a concurrent program only the ordering of actions visible to other threads is important. This reduction can be made without loss of information [2, 16].

Stubborn Sets. In [17, 18] the theory of stubborn sets, which is based on commutativity, is presented. Two versions, weak and strong, are distinguished. The weak theory is more complicated and more difficult to implement, but it leads to better reduction results. This method tries to "save effort by postponing the investigation of structural transitions to future states..." [18].

Sleeping and Persistent Sets. In [6, 7] sleeping sets and persistent sets are presented. Sleeping sets capture information of the past of the search. This information is being used to avoid unnecessary transitions. "...sleep sets avoid the investigation of transitions that have been investigated in the past states." [18]. Persistent sets can be seen as an enhancement of stubborn sets. The semantic model was inspired by Mazurkiewicz's traces [11].

[3] Nevertheless, this has no effect on the computed results.

Ample Sets. Ample sets are persistent sets satisfying additional conditions sufficient for LTL model checking [15]. Minea [12] uses also ample sets, but with a less restrictive independence relation.

Symmetric Reduction. A system may contain several identical components that are coupled to each other. Symmetric reduction tries to find such symmetries. Its complexity is proved to be the same as that of the graph isomorphism problem [9].

Dynamic Partial-Order Reduction. An approach somehow similar to ours, but dynamic in its nature, can be found in [8].

There is a lot of work building on the papers mentioned above. Some combine several approaches to achieve better results. A short overview of other techniques e.g. binary decision diagrams (BDDs), unfolding method, data independence, and Holzmann's supertrace can be found in [18]. In order to perform a more precise commutativity analysis a static and dynamic object escape analysis is being incorporated in [5]. Information about locks is being collected. This approach improves the performance of partial-order techniques on shared-memory programs.

CHESS [13], a concurrency unit testing tool, exhaustively explores the thread schedules of a concurrent program within a budget of c preemptions. Model checking techniques are being used in order to systematically generate all interleavings for a given scenario.

In [10] it is shown that for unidirectional bitvector problems in analyzing parallel programs with shared memory it is sufficient to perform a linear scan of each thread rather than to analyze all possible interleaving sequences.

7 Conclusion

We have presented an algorithm for attacking the state explosion problem in analyzing multithreaded programs. Our approach employs partial-order reduction and static virtual coarsening. It uses information on shared variables to generate and interleave blocks of statements. The number of interleavings compared to the original setting is reduced significantly.

Our algorithm performs polynomially as long as the number of shared variables is constant. However, its worst-case behavior is exponential.

We have already implemented the algorithm and tested it on hand-crafted examples. An interface to an existing compiler or parser will be a future step.

We are currently working on an operational semantics which should enable the justification of our work. Furthermore, we are planning to support conditionals and loops.

References

1. M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover, New York, 1964.

2. E. A. Ashcroft and Z. Manna. Formalization of Properties of Parallel Programs. In B. Meltzer and D. Michie, editors, *Proc. of the Sixth Annual Machine Intelligence Workshop, Edinburgh, 1970*, pages 17–41. University of Edinburgh Press, 1971.
3. M. Ben-Ari and A. Burns. Extreme Interleavings. *IEEE Concurrency*, 6(3):90–91, 1998.
4. B. Burgstaller, J. Blieberger, and R. Mittermayr. Static Detection of Access Anomalies in Ada95. In *Proceedings of the 11th International Conference on Reliable Software Technologies – Ada-Europe'2006, Porto, Portugal*, volume 4006 of *Lecture Notes in Computer Science*, pages 40–55. Springer-Verlag, June 2006.
5. M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting Object Escape and Locking Information in Partial-Order Reductions for Concurrent Object-Oriented Programs. *Formal Methods in System Design*, 25(2-3):199–240, 2004.
6. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, USA, 1996.
7. P. Godefroid. On the Costs and Benefits of Using Partial-Order Methods for the Verification of Concurrent Systems. In D. A. Peled, V. R. Pratt, and G. J. Holzmann, editors, *POMIV'96: Proc. of the DIMACS workshop on Partial Order Methods in Verification*, pages 289–303, New York, USA, 1997. AMS Press, Inc.
8. G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian Partial-Order Reduction. In D. Bosnacki and S. Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 95–112. Springer-Verlag, 2007.
9. T. Junttila. *On The Symmetry Reduction Method For Petri Nets and Similar Formalisms*. PhD thesis, Helsinki University of Technology, 2003.
10. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.
11. A. Mazurkiewicz. Introduction to Trace Theory. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific Pub. Co., Inc., 1995.
12. M. Minea. *Partial Order Reduction for Verification of Timed Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1999.
13. M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. *SIGPLAN Not.*, 42(6):446–455, 2007.
14. W. T. Overman. *Verification of Concurrent Systems: Function and Timing*. PhD thesis, University of California, Los Angeles, 1981.
15. D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In D. Dill, editor, *CAV '94: Proc. of the 6th Int. Conf. on Computer Aided Verification*, volume 818 of *LNCS*, pages 377–390, London, UK, 1994. Springer-Verlag.
16. A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1986.
17. A. Valmari. Eliminating Redundant Interleavings During Concurrent Program Verification. In Odijk, E. and others, editor, *PARLE'89: Proc. of the Conf. on Parallel Architectures and Languages Europe, Eindhoven, Netherlands; Vol. 2*, volume 366 of *LNCS*, pages 89–103, Berlin, Germany, 1989. Springer-Verlag.
18. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer-Verlag, September 1996.