Automatic Synthesis of Deterministic Concurrency

Veselin Raychev[1], Martin Vechev[1], and Eran Yahav[2]

[1] ETH Zurich
{veselin.raychev,martin.vechev}@inf.ethz.ch
[2] Technion
yahave@cs.technion.ac.il

Abstract. Many parallel programs are meant to be deterministic: for the same input, the program must produce the same output, regardless of scheduling choices. Unfortunately, due to complex parallel interaction, programmers make subtle mistakes that lead to violations of determinism.
In this paper, we present a framework for static synthesis of deterministic concurrency control: given a non-deterministic parallel program, our synthesis algorithm introduces synchronization that transforms the program into a deterministic one. The main idea is to statically compute inter-thread ordering constraints that guarantee determinism and preserve program termination. Then, given the constraints and a set of synchronization primitives, the synthesizer produces a program that enforces the constraints using the provided synchronization primitives.
To handle realistic programs, our synthesis algorithm uses two abstractions: a thread-modular abstraction, and an abstraction for memory locations that can track array accesses. We have implemented our algorithm and successfully applied it to synthesize deterministic control for a number of programs inspired by those used in the high-performance computing community. For most programs, the synthesizer produced synchronization that is as good or better than the handcrafted synchronization inserted by the programmer.

1 Introduction

Many parallel programs are meant to be deterministic: for the same input, the program must produce the same output. Unfortunately, concurrent programming mistakes often result in parallel programs that are non-deterministic: for the same input, different executions of the program produce different outputs. Manually enforcing determinism is a time consuming, error-prone and inefficient task: introducing too much synchronization can lead to sequentializing the parallel program, while introducing too little synchronization can produce a non-deterministic program.

In this paper we propose to automatically synthesize deterministic concurrency control: given a non-deterministic (potentially infinite-state) parallel program, our algorithm will statically introduce synchronization that transforms the input program into a deterministic parallel program.

Determinism Verification under Abstraction Direct verification of determinism requires comparing the output states of different executions starting from the same input state. Equality between states can be easily determined when states are concrete. However, to handle infinite-state programs one must employ abstraction. Under abstraction,

the equality relationship between concrete states is lost. Equality between abstract states does not entail equality between the concrete states they represent, and therefore establishing equality between abstract states is insufficient.

Establishing Determinism by Conflict-Freedom Rather than verifying (and enforcing) determinism directly, we focus on verifying (and enforcing) a stronger property called *conflict-freedom*: if the program is conflict-free then it is deterministic. Informally, a program is conflict-free if in any concrete program state, parallel threads do not access (where at least one access is a write) the same memory location. Conflict-freedom allows us to reason about determinism in a local way – by using a property that can be locally enforced, we ensure that the resulting program is deterministic.

Our approach uses abstract interpretation [9] to compute an over-approximation of the possible concrete program behaviors. Then, the algorithm checks whether the over-approximation is conflict-free and if so, verification of conflict-freedom (and thus determinism) succeeds. Otherwise, the algorithm synthesizes a repair that enforces conflict freedom. It does so by synthesizing an inter-thread ordering constraint on the accesses performed by conflicting threads. That is, the synthesis algorithm *statically determinizes* the order of operations performed by conflicting threads.

Motivation A comprehensive study [18] shows that nearly a third of all concurrency errors in a variety of open source projects are inter-thread "ordering" violations. As noted in [18], such violations cannot be easily fixed with atomicity and locking constructs. Vasuvedan et al. [28] express desire for a "determinizing compiler", but do not provide any analysis.

Over the years, there has been significant interest in automatically enforcing mutual exclusion properties in parallel programs, usually by inferring locks and atomicity constructs [20, 30, 8]. Comparatively, there has been little focus on static techniques for enforcing "ordering" relationships or determinism, exceptions being the works of [23, 6, 14]. Relationship to existing work is discussed in detail in Section 8.

We present a synthesis framework for statically enforcing determinism. The framework consists of a novel thread-modular synthesis algorithm that enables the use of flow-sensitive techniques for analyzing each thread. We instantiate the framework with powerful abstract domains such as Octagon [22] and Polyhedra [10], enabling tracking and avoidance of conflicting memory accesses at a fine granularity.

Main Contributions Our main contributions are as follows:

- A thread-modular synthesis algorithm which takes as input a potentially non-deterministic parallel program, and "determinizes" the program by synthesizing inter-thread ordering constraints between conflicting statements in a way that preserves program termination.
- An algorithm that takes as input a set of inter-thread ordering constraints produced by the synthesizer and a particular synchronization primitive and realizes the constraints using the synchronization constructs. To illustrate the concept, we show a translation to two kinds of synchronization primitives: the classic *signal/wait* synchronization and the *spawn/sync* constructs used in structured parallel languages such as Cilk [4].

– An implementation of the algorithm in a tool based on Soot [27] and Apron [13], using powerful numerical abstract domains such as Octagon and Polyhedra.
– An evaluation of the tool on a set of Java programs derived from those used in the high performance community. Our results indicate that the tool can be practically useful: for most programs, it produced synchronization that is as good or better than the initial hand-crafted synchronization inserted by the programmer.

The tool's source code, the benchmarks and instructions how to build and run the tool are available open source at: `http://www.srl.inf.ethz.ch/dps.php`.

Limitations

– We note that for general programs, non-determinism may occur due to other reasons like random number generators, network or user interaction. In this work, we focus on programs for which the non-determinism is only due to conflicts.
– We focus on programs with a constant number of threads. However, this limitation is imposed only by the used synchronization primitives. For example, programs using *signal*/*wait* synchronization require careful attention to the number of *signal*/*wait* calls based on the number of threads.

2 Overview

Given a parallel program P, our goal is to synthesize a deterministic parallel program P' by adding synchronization to P. To handle infinite-state programs, our synthesis algorithm is based on abstract interpretation [9], and takes an abstraction α as one of its parameters. In this setting, the problem can be phrased as:

> *Given a parallel program P, and an abstraction α, our goal is to synthesize a deterministic parallel program P' by adding synchronization to P such that P' can be automatically verified as deterministic under the abstraction α.*

Challenges Any synthesis algorithm targeting the above problem must address at least the following three challenges:

– Scalability: The synthesizer should soundly handle realistic infinite-state concurrent programs.
– Termination: The inferred synchronization should preserve program termination.
– Number of solutions: The synthesizer should provide a mechanism which allows to control the number of solutions.

Next, we illustrate our approach on an example. The formalization and evaluation are provided in later sections.

2.1 Example Program

Consider the simple program shown in Fig. 1(a). Here, a main thread creates two threads using the `spawn` construct which in turn execute in parallel and access shared variables `x` and `y` (both initialized to 0). The conditional at line 7 executes atomically. In

this program, different schedules can lead to different final values for `x` and `y`. For example, the schedule `x = 0;y = 0;x = 1;if (x == 0) y = 1` results in values $x = 1, y = 0$, and the schedule `x = 1;x = 0;if (x == 0) y = 1;y = 0` results in values $x = 0, y = 0$. Our goal is to add efficient synchronization to the program such that its result is deterministic, i.e., all executions of the new program yield the same output state when starting from the same initial values for x and y.

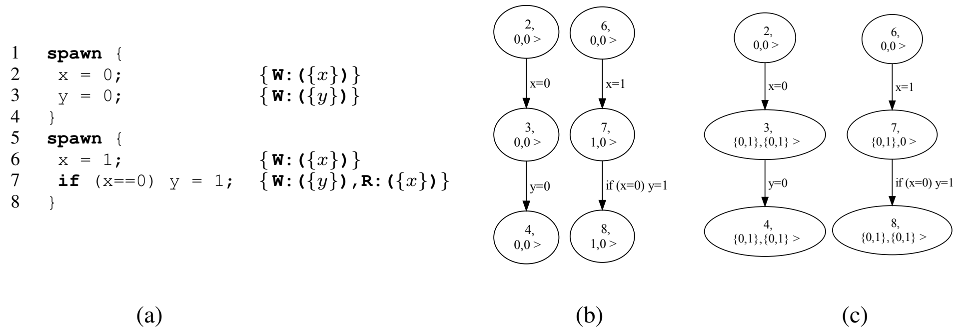

Fig. 1. Example with two spawned threads (a) and its thread-modular transition systems for the threads, (b) before, and (c) after stabilization.

2.2 Thread-Modular Synthesis

To add the necessary synchronization, we present a novel synthesis algorithm that generates *inter-thread ordering constraints*, describing an ordering between statements of different threads. A set of inter-thread ordering constraints can then be implemented in various ways, for instance by adding synchronization to the program.

One approach would be to build a global transition system of the program in the style of [16] and use an iterative algorithm to eliminate the "bad" states (states that cause non-determinism) from the transition system. Unfortunately, building a global transitions system does not scale to realistic concurrent programs as one needs to reason about all program interleavings.

Instead, to avoid global reasoning, we introduce a thread-modular synthesis algorithm. To enable thread-modular reasoning the algorithm uses a thread-modular abstraction. The synthesis algorithm consists of the following three phases:

Phase 1: Compute Stable Invariants First, each thread is analyzed sequentially, initially assuming that all memory locations accessed by the thread are independent from locations accessed by threads that may execute in parallel. Fig. 1 (b) shows the thread-modular transition systems for the two threads in the program of Fig. 1 (a) after each thread is analyzed sequentially. The values in a state are denoted as a tuple $\langle pc, x, y\rangle$. Note that the combination of states (i.e., concretization) in the transition systems of

Fig. 1 (b) does not cover all possible states of the original program: it does not represent a final state $x = 0, y = 1$, clearly possible in the program.

To guarantee soundness, after the initial analysis of each thread, the analyzer checks whether the independence assumption holds. If not, it iteratively weakens the computed invariants until they stop changing as a result of interference with other threads. This stabilization is usually achieved by having each thread include the values of interfering locations produced by other threads (e.g., [21]).

For the program in Fig. 1 (a), the accessed memory locations for each statement are shown in curly braces. The statements `x = 0` and `x = 1` are *conflicting* as they can execute in parallel and both write to location `x`. Similarly for `x = 0` and `if (x==0) y = 1` as well as for `y = 0` and `if (x==0) y = 1`. As conflicts arise, our algorithm needs to weaken the invariants computed in each thread's transition system. In our example, the weakening results in the transition systems of Fig. 1 (c). Note that in these (abstract) transition systems, `x` and `y` can have more than one possible value in each state. After the weakening, the result is sound, and indeed, the transition systems do capture the state where $x = 0, y = 1$.

Phase 2: Identify and Resolve Conflicts After the invariants for each thread are computed, the next step is to synthesize a repair that determinizes each conflict. Determinization of a conflict is achieved via an *inter-thread ordering constraint*. An inter-thread ordering constraint restricts the order in which statements from different threads may be executed. Using the transition systems of Fig. 1 (c), the algorithm produces the following formula: $\psi = (l_2 \prec l_6 \vee l_6 \prec l_2) \wedge (l_2 \prec l_7 \vee l_7 \prec l_2) \wedge (l_3 \prec l_7 \vee l_7 \prec l_3)$.

Each term in the formula is a constraint determinizing two conflicting statements. The meaning of a term $l_2 \prec l_6$ is defined in terms of traces. Informally, the traces which satisfy $l_2 \prec l_6$ are those where if statements at labels l_2 and l_6 are performed in the trace, then l_2 must occur before l_6. The formal semantics of such terms are defined in Section 4.

Id	(a) Satisfied terms	(b) Non-redundant terms
1	$l_2 \prec l_6, l_2 \prec l_7, l_3 \prec l_7$	$l_2 \prec l_6, l_3 \prec l_7$
2	$l_6 \prec l_2, l_2 \prec l_7, l_3 \prec l_7$	$l_6 \prec l_2, l_3 \prec l_7$
3	$l_2 \prec l_6, l_7 \prec l_2, l_3 \prec l_7$	—
4	$l_6 \prec l_2, l_7 \prec l_2, l_3 \prec l_7$	—
5	$l_2 \prec l_6, l_2 \prec l_7, l_7 \prec l_3$	$l_2 \prec l_6, l_7 \prec l_3$
6	$l_6 \prec l_2, l_2 \prec l_7, l_7 \prec l_3$	$l_6 \prec l_2, l_2 \prec l_7, l_7 \prec l_3$
7	$l_2 \prec l_6, l_7 \prec l_2, l_7 \prec l_3$	—
8	$l_6 \prec l_2, l_7 \prec l_2, l_7 \prec l_3$	$l_7 \prec l_2$

Table 1. Solutions to ψ.

The models which satisfy the synthesized formula represent *potential solutions* for making the program deterministic. For instance, for the formula above, we have eight minimal solutions as shown in Table 1 (column a). Each row in column (a) lists a solution of the formula (we list the terms which are $true$).

Termination and Redundancy Unfortunately, some of these solutions are undesirable and others can be minimized. In particular, solutions 3, 4 and 7 cause non-termination via deadlock. For instance, solution 3 requires label 3 to execute before label 2, clearly not possible. Further, solutions 1, 2, 5 and 8 contain redundant terms that can lead to unnecessary synchronization when implemented directly. For instance, in solution 1, the term $l_2 \prec l_7$ is redundant because it is subsumed by the term $l_3 \prec l_7$. Intuitively, this is because if the statement at label 3 is executed before the statement at label 7, then

the statement at label 2 is also executed before the statement at label 7. Interestingly, solution 6 contains no redundant terms and does not introduce non-termination.

Our algorithm addresses both of these issues: it adds terms to the formula ψ that prevents cycles, and only outputs solutions that do not contain redundant terms. With the new formula (details are in Section 5), our algorithm produces the five solutions shown in Table 1(b). Here, '—' means that the corresponding solution in that row in column (a) does not terminate and hence it is not selected in column (b). Indeed, these solutions do not introduce non-termination and they do not contain redundant terms (up to the thread-modular abstraction as discussed later).

Reducing the number of solutions Even after the filtering above, it is possible to produce too many solutions. There are three principal approaches to deal with this problem: (i) provide additional specifications that solutions must satisfy. For example, in the case of a read-write conflict, require that the write always takes place before the read as the write initializes the data accessed by the read. This particular specification filters solution 6 from the list. (ii) define criteria that compare solutions. A simple criteria could be to filter solutions that sequentialize the program when there are other solutions which do not. This criteria filters solution 8. (iii) using coarser synchronization constructs to realize the constraints, this arises naturally in phase 3, and is discussed below.

```
1  spawn {
2   o1.wait(); x = 0;
3   y = 0; o2.signal();
4  }
5  spawn {
6   x = 1; o1.signal();
7   o2.wait(); if (x==0) y = 1;
8  }
```

(a)

```
spawn {
 x = 0;
 y = 0;
}
sync; spawn {
 x = 1;
 if (x==0) y = 1;
}
```

(b)

```
spawn {
 x = 1;
 if (x==0) y = 1;
}
sync; spawn {
 x = 0;
 y = 0;
}
```

(c)

Fig. 2. (a) Enforcing solution $l_6 \prec l_2$, $l_3 \prec l_7$ with signal/wait. (b) Enforcing solution $l_2 \prec l_6$ and $l_3 \prec l_7$ with *sync*. (c) Enforcing solution $l_7 \prec l_2$ with *spawn* and *sync*.

Phase 3: Realization of Solutions A solution can be enforced with a variety of synchronization mechanisms. To illustrate the issues that arise when realizing solutions into the program, we selected two different synchronization primitives:

- *spawn/signal/wait*: a thread is created with a *spawn*, a thread can notify another thread by invoking *o.signal()* on a signaling object *o* and a thread can wait to be notified with *o.wait()*. Once a thread is notified, it continues execution.
- *spawn/sync*: this synchronization mechanism is used by structured parallel programming languages such as Cilk [4]. A thread is created with a *spawn*. When a thread calls *sync*, the thread blocks and waits until all of its children threads (threads that it has spawned) as well as their descendants complete.

With the first mechanism all five solutions in Table 1(b) can be implemented directly. For instance, the implementation of solution $l_6 \prec l_2$, $l_3 \prec l_7$ is shown in Fig. 2 (a).

Two interesting points need to be noted when using the *spawn*/*sync* constructs. First, not all of the five solutions are directly implementable with *spawn*/*sync*. For example, the solution $l_6 \prec l_2$, $l_2 \prec l_7$, $l_7 \prec l_3$ cannot be implemented by placing a *sync* construct anywhere in the program. In fact, from the set of five solutions, only the ordering $l_2 \prec l_6$, $l_3 \prec l_7$ can be implemented by placing a *sync* in the program. The resulting program is shown in Fig. 2 (b). Second, the implemented solution enforces sequentialization of the two threads, that is, the implemented solution is *more coarse* than what the actual constraints require.

Indeed, with certain synchronization primitives, one may obtain fewer and coarser solutions than what the solutions yielded by phase 2 require. Hence, one side-effect of using coarser synchronization constructs is obtaining fewer solutions. Therefore, this is the third mechanism that can lead to fewer solutions produced by the synthesizer. In Section 7 we show an evaluation of the two synchronization mechanisms and their final number of solutions.

Inferring* spawn*s In addition to *sync* statements, our approach can also infer a placement for *spawn*'s. In our example it is impossible to find a placement of *sync* in the program that realizes the solution $l_7 \prec l_2$. However, if the user had omitted the *spawn* statements, then our algorithm can infer a placement of *spawn*'s (and *sync*) that realizes $l_7 \prec l_2$. The result is shown in Fig. 2 (c).

Precision Because of abstraction, it is possible to produce unnecessary constraints. This is expected as the abstraction loses information in order to make static analysis tractable. Consider solution $l_2 \prec l_6$, $l_3 \prec l_7$ from Table 1(b). Here, the term $l_3 \prec l_7$ is unnecessary because if $l_2 \prec l_6$ is enforced, `y = 1` would not execute and hence there would be no conflict with the statement at label 3. One can attempt to refine the abstraction to avoid unnecessary solutions, but in general it is impossible to completely avoid this effect.

Preserving Termination The solutions produced in phase 2 should not introduce non-termination. However, when implementing these solutions into a program, deadlock may be introduced.

To illustrate the point, we slightly modify the example of Fig. 1: assume that the statement `y = 1` at label 7 is now executed separately from its guard. Suppose that we would like to realize the solution where `y = 1` is always performed before `y = 0`. If we implement this with *signal*/*wait*, we can introduce non-termination. The reason is that if we place a *signal* right after `y = 1` and a *wait* right before `y = 0`, then it is possible that the execution of statements (in this sequence): `x = 0`, `x = 1`, *wait* leads to a deadlock. The reason is that `y = 1` will never be reached (and the *signal* will never be invoked). This issue can be addressed at any of the three phases. We address the problem in phase 1 and make sure that the inter-thread constraints in the formula only use labels that are always performed by the program (defined later).

Our approach soundly handles programs with loops and conditionals, the main point here is that care must be taken when conflicting labels participate inside conditionals and loops (the details of our solution can be found in Section 5).

2.3 Abstracting Memory Accesses

So far, we illustrated the steps of our algorithm on a simple example. However, realistic programs introduce additional challenges in the form of unbounded number of dynamically allocated objects, and accesses to arrays of unknown sizes. To address this issue, we use an abstraction of memory locations that combines information from a (simple) heap abstraction with information from a numerical abstraction of array indices. Here, we briefly illustrate the abstractions on the example in Fig. 3.

```
1   void update(double[] B, double[] C) {
2    spawn {
3     for (int i=1;i <= n; i++) {
4      int ci = 2*i;
5      double t1 = C[ci];    {R:({A_C},{2 ≤ ci ≤ 2*n})}
6      B[i] = t1;            {W:({A_B},{1 ≤ i ≤ n})}
7     }
8    }
9    spawn {
10    for (int j=n;j <=2*n; j++) {
11     int cj = 2*j+1;
12     double t2 = C[cj]; {R:({A_C},{2*n+1 ≤ cj ≤ 4*n+1})}
13     B[j] = t2;         {W:({A_B},{n ≤ j ≤ 2*n})}
14    }
15   }
16  }
```

Fig. 3. Simple example for parallel accesses to shared arrays.

The threads in the program of Fig. 3 access two arrays `B` and `C` passed as parameters. Our abstraction for memory locations over-approximates the memory locations accessed by each statement. We represent the set of (abstract) memory locations accessed by each statement as a pair of heap information and array index range. The heap information records what abstract locations may be pointed to by the array base reference. The array index-range records what indices of the array may be accessed by the statement via constraints on the index variable.

For this program, our pointer analysis is able to establish that `B` and `C` correspond to disjoint (abstract) memory locations A_B and A_C, respectively (by analyzing the rest of the program, not shown here). In this example, we used the Polyhedra abstract domain [10] to abstract numerical values, and the array index range is generally represented as a set of linear inequalities on local variables of the thread. For example, in Line 5 of the example, the array base `C` may point to a single abstract location A_C, and the statement reads from the range $2 \leq ci \leq 2 * n$.

To identify a conflict, our algorithm reasons about potential overlaps between abstract memory locations. In the example of Fig. 3, the ranges of array indices represented by linear inequalities overlap. That is, the writes at Line 6 and Line 13 overlap as the abstract memory locations $(\{A_B\}, 1 \leq i \leq n)$ and $(\{A_B\}, n \leq j \leq 2 * n)$ intersect, leading to potentially conflicting writes by the two threads when `i=j=n`.

3 Background

Here, we provide basic notations and definitions which we use in the rest of the paper.

Programming Language Our synthesis algorithms are applicable to standard off-the-shelf concurrent/parallel programming languages. To simplify presentation, we assume a simple sequential imperative language augmented with the *spawn* statement for creating parallel threads. We use *TIds* to denote the set of thread identifiers, *VarIds* to denote the set of local variable identifiers, and *Labs* to denote the set of program labels. We assume the code in each thread is augmented with an initial label (the label of the first statement in the thread) and a final label (the label after the last statement in the thread). We assume that labels are unique to each thread. We denote the thread of a label l by $tid(l)$. For an assignment statement at label l, $lhs(l)$ denotes the left hand side, and $rhs(l)$ the right hand side. To simplify exposition, we assume the language only contains array accesses. The treatment of shared field accesses is similar (and simpler).

Transition System A transition system is a tuple $\langle \Sigma_0, F, \Sigma, T\rangle$ where Σ is a set of states, $T \subseteq \Sigma \times \Sigma$ is a set of transitions between states, $\Sigma_0 \subseteq \Sigma$ are the initial states and $F \subseteq \Sigma$ are the final states. For a transition $\tau \in T$, we use $src(\tau)$ and $dst(\tau)$ to denote its source and destination states and $tid(\tau)$ to denote the thread which performed τ. A state is final if all threads are at their final label in that state. There are no outgoing transitions from a final state.

Concrete Semantics We assume standard semantics which define a program state and evaluation of expressions and statements in that program state. The semantic domains are defined in the standard way in Table 2. As we focus our exposition on arrays, each l-value is a pair $(a, n) \in (A^\natural \times \mathbb{N})$.

$A^\natural \subseteq \mathit{aobjs}^\natural$	allocated arrays
$v^\natural \in \mathit{Val} = \mathit{aobjs}^\natural \cup \{\mathit{null}\} \cup \mathbb{N}$	values
$lv^\natural \in LVal = \mathit{aobjs}^\natural \times \mathbb{N}$	l-values
$pc^\natural \in PC = \mathit{TIds} \rightharpoonup (\mathit{Labs} \cup \bot)$	program counters
$\rho^\natural \in Env^\natural = \mathit{TIds} \times \mathit{VarIds} \rightharpoonup \mathit{Val}$	environment
$h^\natural \in Heap^\natural = LVal \rightharpoonup \mathit{Val}$	heap

Table 2. Semantic Domains

A *program state* is a tuple: $\sigma = \langle pc^\natural_\sigma, \rho^\natural_\sigma, h^\natural_\sigma, A^\natural_\sigma\rangle \in ST^\natural$, where $ST^\natural = PC \times Env^\natural \times Heap^\natural \times 2^{\mathit{aobjs}^\natural}$. A state σ keeps track of the program counter for each thread ($pc^\natural_\sigma$) (undefined if the thread has not yet been activated), an environment mapping local variables to values ($\rho^\natural_\sigma$), a mapping from allocated array objects and indices to values ($h^\natural_\sigma$), and a set of allocated array objects ($A^\natural_\sigma$).

We denote $threads(\sigma)$ the set of thread identifiers in $dom(pc^\natural_\sigma)$ which are not mapped to $\bot$. We use $succ(\sigma)$ to denote the set of states that are direct successors of σ in the transition system. The set of threads which can perform a transition out of state σ is denoted by $succtid(\sigma)$. For a transition τ, we denote by $wr(\tau) \subseteq LVal$ the set of memory locations written by τ, by $rd(\tau) \subseteq LVal$ the set of memory location read by τ, and by $rw(\tau) = wr(\tau) \cup rd(\tau)$ the set of locations accessed by τ.

The transition system of a program P is denoted by $\langle \Sigma_0, F_P, \Sigma_P, T_P\rangle$. Every transition $\tau \in T_P$ is associated with a statement that performed the transition and its label is denoted by $lbl(\tau)$.

A trace $\pi = \tau_0 \cdot \tau_1 \ldots$ of a program P is a sequence of transitions, such that for $i \geq 0$, $\tau_i \in T_P$, $src(\tau_{i+1}) = dst(\tau_i)$ and $src(\tau_0) \in \Sigma_0$. We denote the set of traces of P by $[\![P]\!]$. We denote the first state of a trace π by $first(\pi) = src(\tau_0)$ and the last state of a finite trace π by $last(\pi) = dst(\tau_{n-1})$, $n = |\pi|$.

Determinism Informally, a program is deterministic if it produces (observationally) equivalent outputs for all (observationally) equivalent inputs. For programming languages where each statement is deterministic, ensuring end-to-end determinism can be achieved if concurrent shared memory accesses are ordered such that the program is *conflict-free*. Conflict-freedom is a strong property which allows us to prove and establish determinism without devising abstractions for automatically reasoning about state equality, a task that can be very challenging when analyzing real programs.

Definition 1 (Conflicting Transitions). *We say that two transitions τ and τ' are* conflicting*, denoted by $\tau \not\parallel \tau'$, when: i) $tid(\tau) \neq tid(\tau')$, ii) $src(\tau) = src(\tau')$ and iii) $wr(\tau) \cap rw(\tau') \neq \emptyset$ or $wr(\tau') \cap rw(\tau) \neq \emptyset$.*

Definition 2 (Conflict State). *A state $\sigma \in \Sigma$ is a* conflict state *if there are two transitions τ, τ' such that $src(\tau) = \sigma$ and $\tau \not\parallel \tau'$.*

A program P is *conflicting* if it has a reachable conflict state $\sigma \in \Sigma_P$. Otherwise, the program is *conflict-free*.

4 Constraints and Termination Guarantees

This section states a theorem which outlines the conditions under which enforcing ordering constraints will preserve termination. To state the theorem, we define necessary concepts such as termination, thread blocking, and (combination of) ordering constraints. Indeed, any synthesis algorithm which operates in the setting outlined in this section can provide the guarantees stated by the theorem. One such synthesis algorithm is provided in Section 5.

4.1 Program Termination

To define that a program P *halts*, it is enough to show that every trace $\pi \in [\![P]\!]$ is finite. This property is sufficient when all states with no outgoing transitions are final states. However programs that deadlock do not reach final state and yet they *halt*. We refine the definition of termination to exclude halting in non-final states.

Definition 3 (Terminating Set of Program Traces). *A set of traces $S \subseteq [\![P]\!]$ is terminating if:*

1. *every trace $\pi \in S$ is finite.*
2. *for any trace $\pi' \in S$, there exists a trace $\pi \in S$ such that π' is a prefix of π and $last(\pi) \in F_P$.*

We say that a program is terminating if the set $[\![P]\!]$ is a terminating set. Note that it is possible for the program to terminate, yet during its execution some threads can be temporarily disabled from making progress. This can happen for instance when a thread is waiting for an external action to occur before it can make a transition. Below we define what it means for a thread to be blocked (or not to be enabled at any point).

Definition 4. *A thread t blocks in a program P if there exists a reachable state $\sigma \in \Sigma_P$, such that $t \notin succtid(\sigma)$ and $pc^{\natural}_{\sigma}(t)$ is not a final label of t.*

For example, if a thread calls *wait* then it (temporarily or permanently) blocks.

4.2 Ordering Constraints

In this work we focus on determinization by enforcing ordering between labels that execute *exactly once* — the motivation for this approach is to ensure termination.

Definition 5 (Single-transition label). *A label l in a program P is a single-transition label if for every finite trace $\pi \in [\![P]\!]$ where $last(\pi) \in F_P$, there is exactly one transition $\tau \in \pi$, such that $lbl(\tau) = l$.*

Next, we define the meaning of a constraint $l_m \prec l_n$ in terms of traces that satisfy it.

Definition 6 (Meaning of $l_m \prec l_n$). *Given a program P, we say that a trace $\pi \in [\![P]\!]$ violates an ordering constraint $l_m \prec l_n$ if:*

- *l_m or l_n are not single-transition labels, or*
- *they are single transition labels where $\exists i, j.\ 0 \leq i \leq j < |\pi|$ such that $lbl(\pi_i) = l_n$ and $lbl(\pi_j) = l_m$.*

Any trace which does not violate $l_m \prec l_n$ satisfies the predicate.

The definition above is naturally extended to a set of ordering constraints $C = \{l_1 \prec l_2, \ldots, l_m \prec l_n\}$. That is, a trace satisfies C only if it satisfies each constraint in C. We use $[\![P]\!]_C$ to denote all program traces which satisfy the set of ordering constraints C. Where convenient we treat the set C as a binary relation on labels. We use $labels(C)$ to denote all labels appearing in the constraints of set C and $labels(C)|_t = \{l \mid l \in labels(C), tid(l) = t\}$ to denote the set labels of thread t appearing in $labels(C)$.

4.3 Constraining Traces

Next, we define what it means for a set of constraints to be consistent. Intuitively, this will correspond to what a synthesis algorithm must produce as the output right before this output is implemented with particular synchronization constructs.

Definition 7. *Given a program P, we say that a set of ordering constraints C is consistent w.r.t P, if:*

1. *$labels(C)$ contains only single transition labels.*
2. *C does not contain cycles: $I_{labels(C)} \cap C^* = \emptyset$.*

3. *for every thread t, there exists a unique set $T \subseteq C$ such that:*
   (a) *T is a total order on $labels(C)|_t$.*
   (b) *$[\![P]\!]_T = [\![P]\!]$.*

The first consistency property is self explanatory. The second property requires that the set of constraints does not conflict with itself. Here $I_{labels(C)}$ is the identity function defined over the set $labels(C)$. Property 3a) requires that if two labels of the same thread appear in C (could be in different constraints), then these two labels must be ordered. Property 3b) states that all traces of the program satisfy the total order. The last two conditions prevents a situation where two labels of the same thread always appear in all program traces (i.e., they are single-transition labels), yet in some traces they appear in one order, and in other traces they appear in the opposite order.

Next, we state a key theorem: removing traces induced by a consistent set of ordering constraints will not introduce non-termination.

Theorem 1. *Given a terminating program P where no thread blocks, and a set of constraints C, if C is consistent, then $[\![P]\!]_C$ is a terminating set of program traces.*

This theorem means that if we produce a program P_C where C is enforced in P such that $[\![P_C]\!] = [\![P]\!]_C$, the resulting program will still be terminating. Next, we will see a thread-modular synthesis algorithm that takes as input a potentially conflicting program and infers a consistent set of constraints C. Then, we will see how to implement C with particular synchronization primitives so to obtain a final conflict-free program.

5 Thread-Modular Synthesis

In this section, we present our thread-modular synthesis algorithm. The algorithm is based on a thread-modular abstraction which over-approximates the concrete behaviors from Section 3, allowing us to reason in a thread-modular way. The algorithm takes as input a potentially conflicting program and outputs a conflict-free program.

First, we define a thread modular abstraction. This abstracts away the relationship between different threads and leads to semantics that tracks each thread separately, rather than tracking all threads simultaneously. Then, once stabilization is obtained, we check for conflicts by combining pairwise thread states and checking whether the combined state is conflict-free.

5.1 Abstraction

We define the projection $\sigma|_t$ of a state σ on a thread identifier t as $\sigma|_t = \langle pc|_t, \rho|_t, h, A\rangle$, where $pc|_t$ is the restriction of pc to t and $\rho|_t$ is the restriction of ρ to t. Given a concrete state $\sigma \in ST^\natural$, the program state for a single thread t is $\sigma|_t \in \widehat{ST}$, where $\widehat{ST} = PC \times Env^\natural \times Heap^\natural \times 2^{aobjs^\natural}$. Given a set of states $S \subseteq ST^\natural$, its abstraction is defined as:

$$\alpha_{tm}(S) = \bigcup_{\sigma \in S} \{\sigma|_t \mid t \in threads(\sigma)\}$$

The program counter pc of a state $\hat{\sigma} \in \widehat{ST}$ contains only a single thread in its domain. Similarly, $threads(\hat{\sigma})$ returns a singleton set that contains the single thread represented in $\hat{\sigma}$ (or the empty set if the thread is mapped to $\perp$ in σ).

Algorithm 1: Thread-Modular Synthesis

```
    Input: Program P with n threads
    Output: Program P' that is conflict-free
1   compute stabilized Σ_P^1, ..., Σ_P^n
2   ψ ← true
3   foreach i in 1, ..., n do
4       foreach j in i+1, ..., n do
5           foreach σ̂_i^tm ∈ Σ_P^i, σ̂_j^tm ∈ Σ_P^j do
6               σ ← σ̂_i^tm ⊕ σ̂_j^tm
7               if σ ≠ ⊥ then
8                   ψ ← ψ ∧ resolve(σ)
9   φ ← ψ ∧ nocycles(ψ)
10  S ← SAT(φ)
11  return implement(P, S)
```

Abstraction Computation We have defined what the abstraction does and not how it is computed. There are various techniques which can automatically compute a thread-modular abstraction of a program [21]. Typically, these analysis algorithms begin by computing inductive invariants for each thread. Then, based on the interference between threads, they weaken the proof of a given thread until the interference checking succeeds. In a later section, we will discuss how this stabilization is accomplished in our setting. After the thread-modular abstraction is obtained, one can apply standard abstractions such as heap or numerical in order to abstract unbounded state (we will see an example later).

5.2 Synthesis

Our thread modular synthesizer is shown in Algorithm 1. After computing invariant stabilization, the algorithm checks for conflicts between states and computes ordering constraints to avoid any conflicts. The constraints are accumulated in a global inter-thread constraint formula ψ. Next, we discuss the ingredients of the algorithm.

Step 1: Identifying Conflicts As defined in Section 3, a conflict is a property of two transitions executed by different threads. Since our abstraction is thread modular, identifying a conflict requires pairwise composition of states of individual threads.

First, we define a pairwise state as a composition of individual thread states. The idea is to define when individual states can be combined into a pairwise state (corresponding to partial concretization, e.g., in [19]). For example, we can define that two individual states can be combined only when they agree on shared data, or when the program locations of the individual threads may indeed occur in parallel.

The combination $pc_1 \oplus pc_2$ of program counter functions pc_1, pc_2 is defined as

$$pc_1 \oplus pc_2 = \begin{cases} \perp, & mhp(dom(pc_1), dom(pc_2)) = false \\ \lambda t.\{pc_1(t) \mid t \in dom(pc_1)\} \cup \{pc_2(t) \mid t \in dom(pc_2)\} & \text{otherwise.} \end{cases}$$

Here, we use the predicate mhp to decide whether two labels may happen in parallel. Our approach is parametric on this predicate's implementation: we can use any existing may-happen analysis to compute the predicate (e.g. [1]). The combination $\rho_1 \oplus \rho_2$ of environments is defined similarly.

Definition 8. *Given two states $\hat{\sigma}_1 = \langle pc_1, \rho_1, h_1, A_1\rangle$, $\hat{\sigma}_2 = \langle pc_2, \rho_2, h_2, A_2\rangle \in \widehat{ST}$, we say that the states are* matching *when $pc_1 \oplus pc_2 \neq \bot$, $\rho_1 \oplus \rho_2 \neq \bot$, $h_1 = h_2$ and $A_1 = A_2$, and define the* composed pairwise state $\sigma^{pw} = \hat{\sigma}_1 \oplus \hat{\sigma}_2$ *of matching states as $\sigma^{pw} = \langle pc_1 \oplus pc_2, \rho_1 \oplus \rho_2, h_1, A_1\rangle$. If the states are not matching, we define $\hat{\sigma}_1 \oplus \hat{\sigma}_2$ to be $\bot$.*

Definition 9 (Conflicting Program). *Given a program P with n threads (1..n), let the reachable states for each thread be $\Sigma_P^1, \ldots, \Sigma_P^n$ respectively, where $\Sigma_P^i \subseteq \widehat{ST}, 1 \leq i \leq n$. We say that the program is conflicting when there exist matching states $\hat{\sigma_i}^{tm} \in \Sigma_P^i, \hat{\sigma_j}^{tm} \in \Sigma_P^j$ such that $\hat{\sigma_i}^{tm} \oplus \hat{\sigma_j}^{tm}$ is a conflict state.*

A program that is not conflicting is a conflict-free program.

Step 2: Compute Single-Transition Labels and Total Orders Next, we show how to build a constraint formula whose satisfying assignments form a consistent set as in Definition 7. First, for each thread t we find a set of single-transition labels $S_t = \{l_i^t\}_{i=1}^n$ such that there exists a total order $TO_t = \{\cup_{i=1}^{n-1}\{l_i^t \prec l_{i+1}^t\}\}$ on the labels in S_t, in a way that each trace in $[\![P]\!]$ satisfies this total order. The set containing the total order of each thread is denoted by $thords = \cup_{t \in TIds} TO_t$.

Next, given a transition τ, we discuss how to compute the functions $l_{pred}(\tau)$ and $l_{succ}(\tau)$ (both of these return a label). Intuitively, the reason we need these functions is to lift labels which participate in a conflict and are not single-transition labels. We assume that l_1^t is the label of the first statement in the thread and l_n^t is the final label in the thread where both are single-transition labels. This guarantees that if a thread t performs a transition τ such that $lbl(\tau) \notin S_t$, then we can always find a transition τ' in a trace π performed by t so that τ' precedes τ in π and $lbl(\tau') \in S_t$. We use the function $l_{pred}(\tau)$ to denote such a label. The function returns the same label regardless in which π the transition τ appears. Similarly, we ensure the existence of a single-transition label of a transition performed after τ in some trace π. We use $l_{succ}(\tau)$ to denote such a label. A trivial solution is to use $l_{pred}(\tau) = l_1^t$ and $l_{succ}(\tau) = l_n^t$, however, we can also choose labels that are "closer" to τ (in all traces where τ appears). In case $lbl(\tau) \in S_t$, we define $l_{pred}(\tau) = l_{succ}(\tau) = lbl(\tau)$.

Step 3: Resolve conflicts The formula ψ accumulates constraints for each conflict state. Let the conflict transitions of state σ be defined as:

$$conflicts(\sigma) = \{(\tau, \tau') \mid \tau \not\parallel \tau', src(\tau) = src(\tau') = \sigma\}$$

Resolving a conflict state with a pair of conflicting transition τ, τ' can be done in two ways: performing τ first or τ' first. Since we would like our formula to contain only single-transition labels, the formula for resolving conflicts in a state becomes:

$$resolve(\sigma) = \bigwedge\{l_{succ}(\tau') \prec l_{pred}(\tau) \vee l_{succ}(\tau) \prec l_{pred}(\tau') \mid (\tau, \tau') \in conflicts(\sigma)\}$$

Up to here, we have ensured that conditions 1 and 3 in Definition 7 are enforced. Next, we make sure that condition 2 (i.e., no solutions with cycles) is also met. Let $terms(\psi)$ be the set of all terms in the boolean formula ψ. Every term has the form $l_a \prec l_b$. Then, the following formula describes all possible ways in which cycles can be eliminated.

$$nocycles(\psi) = \bigwedge\{\vee\{\neg a \mid a \in A, a \in terms(\psi)\} \mid A \subseteq terms(\psi) \cup thords, A \text{ is a cycle}\}$$

After all conflicts are resolved and ψ is computed, the formula $nocycles(\psi)$ is added to ψ obtaining the final formula φ (line 9 of Algorithm 1). Note that all labels of a given thread that appear in $terms(\psi)$ are contained in $thords$, that is, the labels of a given thread are totally ordered. As an optimization, we only need to consider cycles that do not visit the same node multiple times because such cycles can be decomposed into several smaller ones.

Step 4: Compute satisfying assignments to φ Finally, line 10 of Algorithm 1 computes a satisfying assignment to φ. From this satisfying assignment, we select the constraints with positive truth values, which results in a consistent set of constraints that makes the program conflict-free. Note that this set may contain constraints which are implied by other constraints. This is addressed by performing a transitive reduction on the set. Such a reduction is unique and can be computed with an iterative greedy algorithm that at each step removes a constraint implied by others.

6 From Constraints to a Program

In the previous section, we showed how to obtain a consistent set of constraints S. In this section, we discuss how to enforce S in the program by adding synchronization. This process is realized by the $implement(P, S)$ procedure of Algorithm 1.

Realization with* signal/wait *synchronization A synchronization method, which directly corresponds to an ordering constraint between a pair of labels, is a *signal/wait* object. Every *signal/wait* object starts non-signaled and can be signaled by a call to its *signal* method. The *wait* method of a non-signaled object blocks the current thread until the object gets signaled. If l_m and l_n are single-transition labels, then the ordering constraint $l_m \prec l_n$ can be implemented calling *o.signal()* after the statement at label l_m and calling *o.wait()* before the statement at label l_n.

Realization with structured synchronization We also considered a set of constructs used in the structured parallel programming language Cilk [4]. Here, *spawn* creates a new child thread while *sync* blocks until all existing child threads as well as their recursively created children complete.

Consider the program in Fig. 4(a). Here, a main thread spawns two children threads and then updates several variables. Suppose we would like to enforce that x `= 0` $\prec$ x `= 1`, x `= 1` $\prec$ x `= 3`, x `= 3` $\prec$ x `= 2`. Here we abuse notation and use statements instead of labels for readability. Fig. 4(b) shows one possible determinization. To enforce x `= 0` $\prec$ x `= 1`, the second thread is spawned only after the x `= 0` statement, while the thread with x `= 1` is joined before spawning the next thread in order to enforce its order to take place before x `= 3`. Finally, the last *sync* enforces x `= 3` $\prec$ x `= 2`.

In general, as mentioned in Section 2, not every solution can be directly implementable with *spawn*/*sync*: either some coarsening may take place or the solution may not be directly enforceable. In turn, this leads to fewer overall solutions. In the cases when *spawn*/*sync* is possible, we would like a solution that allows for maximum parallelism. The same order as Fig. 4(b) may be enforced by using *sync* immediately after the end of the *spawn* statements. However, larger portions of the program will be sequentialized and leading to less parallelism. We can solve this by allowing *sync* statements to

be inserted only at single-transition labels right before a conflict or right before *spawn* statements. This leads to Algorithm 2.

In this algorithm, we use a rooted tree of program threads. Each thread is a node and its parent node is the thread who spawned it. The main thread is the root node. We refer to this tree as the thread hierarchy. We define *lca* to return the lowest common ancestor in the thread hierarchy and $spawnlabel(a, b)$ to return the label at which b executes *spawn* of thread a (or a parent thread of a if a is not a direct child of b).

x = 0;
spawn {
 x = 1;
}
spawn {
 x = 3;
}
y = 7;
x = 2;
z = 8;
	x = 0;
	spawn {
	 x = 1;
	}
	sync;
	spawn {
	 x = 3;
	}
	y = 7;
	sync;
	x = 2;
	z = 8;
		x = 0;
		y = 7;
		x = 2;
		spawn {
		 x = 1;
		}
		z = 8;
		sync;
		spawn {
		 x = 3;
		}
			x = 0;
			y = 7;
			x = 2;
			spawn {
			 x = 1;
			}
			sync;
			spawn {
			 x = 3;
			}
			z = 8;

(a)	(b)	(c)	(d)

Fig. 4. Example showing different determinizations. (a) the original program, (b) a determinization by adding *sync* statements. (c) a determinization inferring *sync* and *spawn* statements. (d) another determinization inferring *sync* and *spawn* statements.

Inferring* spawn *statements As mentioned earlier, we can realize ordering constraints by inferring not only *sync*, but also *spawn* statements. This is useful in cases where the programmer provides a set of threads without the corresponding *spawn* statement (or they can be only partially specified). Ability to infer both *sync* and *spawn* allows for finer-grained solutions.

Inference of *spawn* statements can produce several solutions for the same set of ordering constraints. For example, programs (c) and (d) in Fig. 4 enforce the same ordering, but they differ in the way they order statement z = 8 (z is a non-conflicting variable).

7 Experimental Evaluation

We implemented the thread-modular synthesis algorithm in a tool called DPS and evaluated its effectiveness on a set of parallel programs. The implementation handles programs written in the (sequential) Java language augmented with parallel constructs. The experiments were conducted using Oracle's Java 1.6 VM on a 4-core 3.5GHz Core i7 machine. The input to DPS is a standard Java program optionally augmented with constructs for thread creation (e.g. *spawn*). The output of DPS is a determinization of the program expressed with the desired synchronization primitives: *signal/wait* or *spawn*/*sync*.

Components of the Synthesizer Our analysis is based on the Soot analysis engine [27]. First, our analysis computes an abstraction of the heap using a flow-insensitive global pointer analysis [17]. Since the pointer analysis is flow-insensitive, its results are sound even in the presence of concurrency. We use the may-alias information mainly to determine abstract array objects. We perform a thread-modular analysis using numerical abstract domains (based on Apron [13]). For our experiments, we used the Octagon

and Polyhedra abstract domains with a simple widening strategy (we identify loops and widen after some constant number of iterations around the loop). The thread modular analysis computes the set of abstract states as required by Algorithm 1. To solve the constraint formulas we used the SAT4J solver [26].

Algorithm 2: Inference of sync

```
   Input: Program P, a set constraints
   Output: Program P' with added
           sync statements
 1 P' = P
 2 foreach l_a ≺ l_b ∈ constraints do
 3   t_a, t_b ← tid(l_a), tid(l_b)
 4   t^p ← lca(t_a, t_b)
 5   if t_a = t^p then l_a^p ← l_a
 6   else l_a^p ← spawnlabel(t_a, t^p)
 7   if t_b = t^p then l_b^p ← l_b
 8   else l_b^p ← spawnlabel(t_b, t^p)
 9   if l_a^p ≺ l_b^p then
10     add sync at l_b^p to P'
11   else
12     return "not realizable"
13 return P'
```

Stabilized Proofs The particular heap abstract domain we use ensures the sequential analysis of each thread produces a stabilized proof and there is no need for refinement. The reason is that the domain abstracts away the *contents* of an abstract object, meaning all possible interferences on that object are considered. Generally this need not be the case, and refinement may be necessary to compute a stabilized proof.

Experimental Data To evaluate DPS, we used benchmarks from the Habanero project [25]. We slightly modified the benchmarks to ensure the number of spawned threads is a constant (all modifications preserve the input-output behavior of the program). Also, our numerical analysis and synthesis focus on a program fragment where threads can execute in parallel and interference is possible. All resulting programs listed in Table 3 perform parallel numerical computations and are meant to be deterministic. To evaluate our tool, we first removed all initial synchronization from the program and then ran the synthesizer. The questions we wanted to answer were:

– can the tool discover the initial synchronization and if so, with which abstract domains?
– which methods are useful to reduce the number of solutions?
– can viable determinizations be obtained in reasonable time?

The results for the first question are summarized in the third and fourth columns of Table 3. Except for SPARSE, running with Polyhedra produced at least as good synchronization as the initial one. In fact, for MOLDYN and SERIES, the tool discovered synchronization that allows for more statements to execute in parallel than in the program before removing synchronization.

We found that in some programs, the Octagon domain was too imprecise and led to coarser than necessary synchronization. Still, the tool produced a deterministic program, but forced threads to sequentialize. For SPARSE, we were unable to discover the initial synchronization because the program contains complicated array aliasing manipulations (an array is indexed with the contents of another array) and the Polyhedra numerical domain is too imprecise to establish that parallel array accesses are disjoint. In all cases, the running time of DPS was less than two minutes.

Program	**Description**	**Abstract Domain**		**Number of Determinizations**			
		Octagon	Polyhedra	fine grained	$W \prec R$	sync + spawn	sync
CRYPT	IDEA encryption	✗	✓	6	1	1	1
MOLDYN	Molecular dynamics simulation	✗	✓	992	72	72	1
SOR	Succesive over-relaxation	✗	✓	2	1	1	1
LUFACT	LU Factorization	✓	✓	7	4	2	1
SERIES	Fourier coefficient analysis	✓	✓	3	2	2	1
SPARSE	Sparse matrix multiplication	✗	✗	2	1	1	1

Table 3. Reconstruction of the initial synchronization with different abstract domains and the number of determinizations with Polyhedra.

Next, we evaluated different methods for reducing the number of solutions. We experimented with the following:

– Adding a specification that orders writes before reads: in case of a read-write conflict, it is often that the write should be ordered before the read except if this would create a cycle in the constraints. The intuition is that the read should access the most recently updated value.
– Choosing orderings that are implementable only with a coarser set of synchronization primitives (e.g., only *spawn* and *sync*).

The fifth column of Table 3 presents the number of solutions with the most fine grained constraints the algorithm could generate. For some programs, this setting produced a high number of determinizations. The sixth column adds a specification to order the writes before the reads. The last two columns include only solutions, where *both*, *spawn* and *sync* are inferred. The last column contains only one determinization. This can happen if the *spawn* statements are fixed in the program and only *sync* statements are inferred. It is worth noting that even in this setting, the synthesized synchronization for MOLDYN and SERIES allowed for more parallelism than the initial synchronization.

8 Related Work

Next, we survey some of the more closely related work concerning determinism.

The work of Navabi et al. [23] focuses on migrating sequential programs into parallel ones. Our work has a different focus, but shares a similar high level problem: given a potentially non-deterministic parallel program, construct an output program that is deterministic. However, there are a number of key technical differences: (i) we use numerical domains to gain precision while [23] only relies on pointer analysis. Without precise numerical domains such as Octagon or Polyhedra, we will end up sequentializing all threads of our benchmarks. Generally, applications in High Performance Computing require rather precise domains to establish correctness. In contrast, in [23], it is often sufficient to enforce coarse-grained synchronization, as any parallel solution is considered an improvement over the sequential program; (ii) our solutions do not

require ta total logical order between threads, resulting in more solutions. This is particularly important when we have a pair of threads but a solution is possible where the thread which is spawned first (but still can run in parallel with the second thread) needs to wait for a transition in the second thread; (iii) in [23], a synchronization point is generated automatically for every shared memory access, even if that access does not conflict with any logically preceding thread. This means that some synchronization may be inserted even if the program is conflict-free. In our approach such synchronization points are unnecessary; (iv) we produce a set of constraints as intermediate form, enabling us to experiment with different synchronization constructs for realizing it.

Next, we examine the technical differences with the work of Botincan et.al. [6]. Here, they start with two sequential programs (e.g. two iterations of a loop) and a proof of some property for each program in separation logic. Then, by examining each assertion in the proof, one can check whether the resources in the proof can be released. Conversely, one can check which resources are needed. Then, a thread can grant these resources to another thread, or block execution until it receives the resources it needs from another thread. Once the releasing and granting of resources is ensured, the programs (i.e., two iterations of a loop) can run in parallel. Our work differs in the following ways: (i) their approach is centered around resources, a concept in separation logic, while our approach is based on abstract interpretation; (ii) the inference algorithms are different: theirs uses logical resources and directly maps (required or unnecessary) resources to specific synchronization primitives, while we use abstract conflict states and produce constraints that can then be mapped to various synchronization primitives (as we show); (iii) their work lacks evaluation, while we present a detailed study of how different specifications and synchronization primitives affect the solution space.

The work of Jin et al. [14] presents a method for enforcing two types of constraints, called *allA-B* and *firstA-B*. While the two works share similar high level goals, the technical details are very different: for instance, their inference algorithm can introduce non-termination via deadlocks. Finally, we present a sound thread-modular synthesizer, while in their work it is assumed that conflicts are provided by an external analysis.

There has been significant interest in various aspects of determinism: automatic verification [29, 15], programming models and systems [7, 24, 11, 2, 3]. Some of these dynamically ensure that the program is deterministic (e.g., aborts in case of conflicts or performs deterministic merge of conflicts, or uses deterministic schedulers). A concern with some of these approaches is that the program may suffer unnecessary slowdowns. To reduce these overheads, some techniques put stringent requirements on the input program (e.g., [24] requires that the input program is data-race free). Further, there is an issue that a small change to the input may cause a vastly different scheduling strategy, causing unpredictable slowdowns. In contrast, our approach is static and guarantees that the output program is deterministic for *all* input states. We believe that the two approaches are complementary.

Other approaches such as DPJ [5] extend a programming language with deterministic constructs and rely on a type system to verify conflict freedom. However, DPJ's type system handling of numerical computations is not as powerful as classic abstract domains and as such cannot prove conflict-freedom for programs such as SOR. More

importantly, it requires explicit annotations of disjointness and suggests no repairs when statements conflict.

Program Synthesis Program synthesis techniques have been successfully used to help programmers discover tricky details, see [12] for a survey. For instance, inference techniques have been used to automatically synthesize missing synchronization such as atomic sections [30] and locks [20]. All of these approaches effectively synthesize a constraint over the statements of the same thread. In contrast, we consider *inter-thread* constraints where comparatively speaking, there has been significantly less work.

9 Conclusion and Future Work

We introduced a synthesis framework for statically enforcing determinism of infinite-state programs. We presented a thread-modular synthesis algorithm, which given a potentially non-deterministic parallel program, discovers ordering constraints that make the program deterministic, without introducing non-termination.

The algorithm identifies abstract conflict states and then synthesizes an inter-thread constraint formula that describes ways to resolve these conflicts. Then, the synthesizer realizes a satisfying assignment to such a constraint in the program via synchronization constructs. We showed how this is accomplished for *signal*/*wait* and *spawn*/*sync* constructs.

We implemented our synthesizer and evaluated it on a set of programs adapted from those used in the high performance community. Our results indicate that the tool is effective: for most programs it managed to quickly synthesize the initial synchronization placement, and in some cases improve it.

There are several interesting directions for future work: (i) defining more expressive inter-thread constraints, (ii) extending the notion of single-transition labels, (iii) refining the thread-modular synthesis algorithm so that stabilization interacts with repairs, and (iv) designing translation algorithms that convert constraints to more expressive synchronization, also enabled by (i).

References

1. AGARWAL, S., BARIK, R., SARKAR, V., AND SHYAMASUNDAR, R. K. May-happen-in-parallel analysis of x10 programs. In *PPoPP '07: Proceedings of the 12th symposium on Principles and practice of parallel programming* (2007), ACM, pp. 183–193.
2. AVIRAM, A., WENG, S.-C., HU, S., AND FORD, B. Efficient system-enforced deterministic parallelism. In *OSDI'10*.
3. BERGER, E. D., YANG, T., LIU, T., AND NOVARK, G. Grace: safe multithreaded programming for c/c++. In *OOPSLA '09*.
4. BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., AND ZHOU, Y. Cilk: an efficient multithreaded runtime system. In *PPoPP'95*.
5. BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., AND VAKILIAN, M. A type and effect system for deterministic parallel java. In *OOPSLA'09*.
6. BOTINCAN, M., DODDS, M., AND JAGANNATHAN, S. Resource-sensitive synchronization inference by abduction. In *POPL '12*.

7. Burckhardt, S., Baldassin, A., and Leijen, D. Concurrent programming with revisions and isolation types. In *OOPSLA '10*.
8. Cherem, S., Chilimbi, T., and Gulwani, S. Inferring locks for atomic sections. In *PLDI '08*.
9. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL'77*.
10. Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *POPL'78*.
11. Devietti, J., Lucia, B., Ceze, L., and Oskin, M. Dmp: deterministic shared memory multiprocessing. In *ASPLOS '09*.
12. Gulwani, S. Dimensions in program synthesis. In *PPDP '10* (2010).
13. Jeannet, B., and Mine, A. Apron: A library of numerical abstract domains for static analysis. In *CAV'09*.
14. Jin, G., Zhang, W., Deng, D., Liblit, B., and Lu, S. Automated concurrency-bug fixing. In *OSDI'12*.
15. Kawaguchi, M., Rondon, P., Bakst, A., and Jhala, R. Deterministic parallelism via liquid effects. In *PLDI '12*.
16. Kuperstein, M., Vechev, M., and Yahav, E. Automatic inference of memory fences. In *FMCAD'10*.
17. Lhoták, O., and Hendren, L. Scaling Java points-to analysis using Spark. In *CC'03*.
18. Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGOPS Oper. Syst. Rev.'2008*.
19. Manevich, R., Lev-Ami, T., Sagiv, M., Ramalingam, G., and Berdine, J. Heap decomposition for concurrent shape analysis. In *SAS '08*.
20. McCloskey, B., Zhou, F., Gay, D., and Brewer, E. Autolocker: synchronization inference for atomic sections. In *POPL '06*.
21. Miné, A. Static analysis of run-time errors in embedded critical parallel c programs. In *ESOP'11*.
22. Miné, A. The octagon abstract domain. *Higher Order Symbol. Comput. 19* (March 2006), 31–100.
23. Navabi, A., Zhang, X., and Jagannathan, S. Quasi-static scheduling for safe futures. In *PPoPP '08*.
24. Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09*.
25. Habanero Multicore Software Research project. http://habanero.rice.edu/hj.
26. The SAT4J SAT solver. available at `http://www.sat4j.org/`.
27. Vallée-Rai, R., et al. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999* (1999), pp. 125–135.
28. Vasudevan, N., and Edwards, S. A. A determinizing compiler. In *PLDI, FIT Session* (2009).
29. Vechev, M., Yahav, E., Raman, R., and Sarkar, V. Automatic verification of determinism for structured parallel programs. In *SAS'10*.
30. Vechev, M., Yahav, E., and Yorsh, G. Abstraction-guided synthesis of synchronization. In *POPL '10*.