Automatic Verification of Determinism for Structured Parallel Programs

Martin Vechev[1], Eran Yahav[1], Raghavan Raman[2], and Vivek Sarkar[2]

[1] IBM T.J. Watson Research Center.
{mtvechev,eyahav}@us.ibm.com
[2] Rice University.
{raghav,vsarkar}@rice.edu

Abstract. We present a static analysis for automatically verifying determinism of structured parallel programs. The main idea is to leverage the structure of the program to reduce determinism verification to an independence property that can be proved using a simple sequential analysis. Given a task-parallel program, we identify program fragments that may execute in parallel and check that these fragments perform independent memory accesses using a sequential analysis. Since the parts that can execute in parallel are typically only a small fraction of the program, we can employ powerful numerical abstractions to establish that tasks executing in parallel only perform independent memory accesses. We have implemented our analysis in a tool called DICE and successfully applied it to verify determinism on a suite of benchmarks derived from those used in the high-performance computing community.

1 Introduction

One of the main difficulties in parallel programming is the need to reason about possible interleavings of concurrent operations. The vast number of interleavings makes this task difficult even for small programs, and impossible for any sizeable software.

To simplify reasoning about parallel programs, it is desirable to reduce the number of interleavings that a programmer has to consider [19,4]. One way to achieve that is to require parallel programs to be *deterministic*. Informally, determinism means that for a given input state, the parallel program will always produce the same output state. Determinism is an attractive correctness property as it abstracts away the interleavings underlying a computation.

In this paper, we present a technique for automatic verification of determinism. A key feature of our approach is that it uses *sequential analysis* to establish independence of statements in the parallel program. The analysis works by applying simple assume-guarantee reasoning: the code of each task is analyzed *sequentially*, under the assumption that all memory locations the task accesses are independent from locations accessed by tasks that may execute in parallel. Then, based on the sequential proofs produced in the first phase, the analysis checks whether the independence assumption holds: for each pair of statements that may execute in parallel, it (conservatively) checks that their memory accesses are independent. Our analysis does not assume any a priori bounds on the number of heap allocated objects, the number of tasks, or sizes of arrays.

Our approach can be viewed as automatic application of the Owicki/Gries method, used to check independence assertions. The restricted structure of parallelism limits the code for which we have to perform interference checks and enables us to use powerful (and costly) numerical domains.

Because in our language arrays are heap allocated objects, our analysis combines information about the heap with information about array indices. We leverage advanced numerical domains such as Octagon [23] and Polyhedra [7] to establish independence of array accesses. We show that tracking the relationships between index variables in realistic parallel programs requires such rich domains.

There has been a large volume of work on establishing independence of statements in the context of automatic parallelization (e.g.,[17,2,24]). These techniques were intended to be part of a parallelizing compiler, and their emphasis is on efficiency. Hence, they usually try to detect common patterns via simple structural conditions. In contrast, our focus is on verification and we use precise (and often expensive) abstract domains.

Our work can be viewed as a case study in using numerical domains for establishing determinism in realistic parallel programs. We show that proving determinism requires abstractions that are quite precise and are able to capture linear inequalities between array indices, as well as establish that different array cells point to different objects.

We implemented our analysis in a tool called DICE based on the Soot [36] analysis framework. DICE uses the Apron [15] numerical library to provide advanced numerical domains (specifically, the octagon and polyhedra domains). Our tool takes as input a normal Java program annotated with structured parallel constructs and automatically checks whether the program is deterministic. In the case where the analysis fails to prove the program as deterministic, DICE provides a description of (abstract) shared locations that potentially lead to non-determinism.

Related Work Recently, there has been growing interest in dynamically checking determinism [5,33]. The main weakness of such dynamic techniques is that they may miss executions where determinism violations occur. Other approaches have also explored building deterministic programs by construction, both in the language [10,4] and via dynamic mechanisms such as modifying the scheduler [8,28]. A related property that has gained much attention over the years is race-freedom (e.g., [13,27,22,34,25,27,11]). However, race-freedom and determinism are not comparable properties: a parallel program can be race-free but not deterministic or deterministic but not race-free.

Main Contributions The main contributions of this paper are:

- We present a static analysis that can prove determinism of structured parallel programs. Our analysis works by analyzing each task *sequentially*, computing assertions using a numerical domain and checking whether the computed assertions indeed imply determinism.
- We implemented our analysis in a tool called DICE based on Soot [36] and the Apron [15] numerical library. The analysis handles Java programs augmented with structured parallel constructs.
- We evaluated DICE on a set of parallel programs that are variants of the well-known Java JGF benchmarks [9]. Our analysis managed to prove five of the eight benchmarks as deterministic.

2 Overview

In this section, we informally describe our approach with a simple Java program augmented with structured parallel constructs.

2.1 Motivating Example

Fig. 1 shows the method `update` which is a simplified and normalized code fragment from the SOR benchmark program. The SOR program uses parallel computation to apply the method of successive over-relaxation for solving a system of linear equations. For this program, we would like to establish that it is deterministic.

```
1  void update(final double[][] G, final int start, final int last,
2   final double c1, final double c2, final int nm, final int ps) {
3    finish foreach (tid: [start,last]) {
4     int i = 2 * tid - ps;
5     double[] Gi = G[i];          {R:({A_G}, {idx = 2*tid - ps}) }
6     double[] Gim1 = G[i - 1];    {R:({A_G}, {idx = 2*tid - ps - 1}) }
7     double[] Gip1 = G[i + 1];    {R:({A_G}, {idx = 2*tid - ps + 1}) }
8     for (int j=1; j<nm; j++)
9      double tmp1 = Gim1[j]        {R:({A_Gim1}, {1 ≤ idx < nm})}
10     double tmp2 = Gip1[j]        {R:({A_Gip1}, {1 ≤ idx < nm})}
11     double tmp3 = Gi[j-1]        {R:({A_Gi}, {0 ≤ idx < nm - 1})}
12     double tmp4 = Gi[j+1]        {R:({A_Gi}, {2 ≤ idx < nm + 1})}
13     double tmp5 = Gi[j];         {R:({A_Gi}, {1 ≤ idx < nm})}
14     Gi[j] =                      {W:({A_Gi}, {1 ≤ idx < nm})}
15       c1 * (tmp1 + tmp2 + tmp3 + tmp4) + c2 * tmp5
16   }
17 }
```

Fig. 1. Example (normalized) code extracted from the SOR benchmark.

This program is written in Java with structured parallel constructs. The **foreach (var : [l,h])** statement spawns child tasks in a loop, iterating on the value range between l and h. Each loop iteration spawns a separate task and passes a unique value in the range $[l, h]$ to that task via the task local variable `var`. A similar construct, called **invokeAll**, is available in the latest Java Fork-Join framework [18].

In addition to **foreach**, our language also supports constructs such as **fork**, **join**, **async** and **finish**. These are basic constructs with similar counterparts in languages such as X10 [6], Cilk [3] and the Java Fork-Join framework [18]. The semantics of **finish** $\{\ s\ \}$ statement is that the task executing the finish must block and wait at the end of this statement until all descendant tasks created by this task in s (including their recursively created children tasks), have terminated.

In Fig. 1, tasks are created by **foreach** in the range of $[start, last]$. Each task spawned by the foreach loop is given a unique value for `tid`. This value is then used to compute an index for accessing a two-dimensional array `G[][]`. Because **foreach** is preceded by the **finish** construct, the main task which invoked the foreach statement cannot proceed until all concurrently executing tasks created by foreach have terminated.

Limitations Our analysis currently does not handle usage of synchronization constructs such as monitors, locks or atomic sections.

2.2 Establishing Determinism by Independence

Our analysis is able to automatically verify determinism of this example by showing that statements that may execute in parallel either access disjoint locations or read from the same location. Our approach operates in two steps: (i) analyzing each task sequentially and computing an over-approximation of its memory accesses; (ii) checking independence of memory accesses that may execute in parallel.

Computing an Over-approximation of Memory Accesses The first step in our approach is to compute an over-approximation of the memory locations read/written by every task at a given program point. To simplify presentation, we focus on the treatment of array accesses. The treatment of object fields is similar and simpler and while we do not present the details here, our analysis also handles that case.

In our programming language, arrays are heap-allocated objects: to capture information about what array locations are accessed, our abstract representation combines information about the heap with information about array indices.

Fig. 2 shows the array `G[][]` of our running example where three tasks with task identifiers (`tid`) 1,2, and 3 access the array. In the figure, we subscript local variables with the task identifier of the task to which they belong. Note that the only columns being written to are Gi_1, Gi_2, Gi_3. Columns which are accessed by two tasks are always only read, not written. The 2D array is represented using one-dimensional arrays.

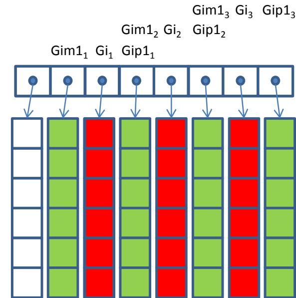

Fig. 2. Example of the array `G[][]` in SOR with three tasks with tids 1,2,3 accessing it.

A key aspect of our approach is that we use simple assume/guarantee reasoning to analyze each task separately, via sequential analysis. That is, we compute an over-approximation of accessed memory locations for a task assuming that all other tasks that may execute in parallel only perform independent accesses.

Fig. 1 shows the results of our sequential analysis computing symbolic ranges for array accesses. For every program label in which an array is being accessed, we compute 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 `idx` variable. In this example, we used the polyhedra abstract domain to abstract numerical values, and the array index range is generally represented as a set of linear inequalities on local variables of the task.

For example, in line 5 of the example, the array base `G` may point to a single abstract location A_G, and the statement only reads a single cell in the array at index $2 * tid - ps$. Note that the index expression uses the task identifier tid. It is often the case in our programs that accessed array indices depend on the task identifier. Furthermore, the

coefficient for tid in this constraint is 2 and thus, this information could not have been represented directly in the Octagon numerical domain. In Section 6, we will see a variety of programs, where some can be handled by Polyhedra and some by Octagon.

Checking Independence Next, we need to establish that array accesses of parallel tasks are independent. The only write access in this code is the write to Gi[j] in line 14. Our analysis therefore has to establish that for different values of tid (i.e., different tasks), the write in line 14 does not conflict with any of the read/write accesses made by other parallel tasks.

For example, we need to prove that when two different tasks identifiers $tid_1 \neq tid_2$ execute the write access in line 14, they will access disjoint locations. Our analysis can only do that if the pointer-analysis is precise enough to establish the fact that $G[2 * tid_1 - ps] \neq G[2 * tid_2 - ps]$ when $tid_1 \neq tid_2$. In this example program, we can indeed establish this fact automatically based on an analysis that tracks how the array G has been initialized. Generally, of course, the fact that cells of an array point to disjoint objects is hard to establish and may require expensive analyses such as shape analysis.

```
1  void update(final double[][] B, final double[][] C) {
2    finish {
3     asynch {
4      for (int i=1;i <=n; i++) {
5       double tmp1 = C[2*i];    {R:({A_C}, {2 ≤ idx ≤ 2*n})}
6       B[i] = tmp1;             {W:({A_B}, {1 ≤ idx ≤ n})}
7      }
8     }
9     asynch {
10     for (int j=n;j <=2*n; j++) {
11      double tmp2 = C[2*j+1]; {R:({A_C}, {2*n+1 ≤ idx ≤ 4*n+1})}
12      B[j] = tmp2;             {W:({A_B}, {n ≤ idx ≤ 2*n})}
13     }
14    }
15   }
16 }
```

Fig. 3. A simple example for parallel accesses to shared arrays.

2.3 Reporting Potential Sources of Non-Determinism

When independence of memory accesses cannot be established, our approach reports the shared memory locations that could not be established as independent.

Consider the simple example of Fig. 3. This example captures a common pattern in parallel applications where different parts of a shared array are updated in parallel. Applying our approach to this example, yields the ranges shown in the figure. Here, we used polyhedra analysis and a simple points-to analysis. Our simple points-to analysis is precise enough to establish two separate abstract locations for B and C. Checking the array ranges, however, shows that the write of line 6 overlaps with the write of line 12

on the array cell with index n. For this case, our analysis reports that the program is potentially non-deterministic due to conflicting access on the abstract locations described by `({`A_B`}, {idx == n})`.

In some cases, such failures may be due to imprecision of the analysis. In Section 6, we discuss the abstractions required to prove determinism of several realistic programs.

3 Concrete Semantics

We assume a standard concrete semantics which defines a program state and evaluation of an expression in a program state. The semantic domains are defined in a standard way in Table 1, where *TaskIds* is a set of unique task identifiers, *VarIds* is a set of local variable identifiers, and *FieldId* is a set of (instance) field identifiers.

$L^\natural \subseteq \mathit{objs}^\natural$	an unbounded set of dynamically allocated objects
$v^\natural \in \mathit{Val} = \mathit{objs}^\natural \cup \{\mathit{null}\} \cup \mathbb{N}$	values
$pc^\natural \in \mathit{PC} = \mathit{TaskIds} \rightharpoonup \mathit{Labs}$	program counters
$\rho^\natural \in Env^\natural = \mathit{TaskIds} \times \mathit{VarIds} \rightharpoonup \mathit{Val}$	environment
$h^\natural \in Heap^\natural = \mathit{objs}^\natural \times \mathit{FieldId} \rightharpoonup \mathit{Val}$	heap
$A^\natural \subseteq L^\natural$	array objects

Table 1. Semantic Domains

A *program state* is a tuple: $\sigma = \langle pc^\natural_\sigma, L^\natural_\sigma, \rho^\natural_\sigma, h^\natural_\sigma, A^\natural_\sigma \rangle \in ST^\natural$, where $ST^\natural = PC \times 2^{\mathit{objs}^\natural} \times Env^\natural \times Heap^\natural \times 2^{\mathit{objs}^\natural}$.

A state σ keeps track of the program counter for each task ($pc^\natural_\sigma$), the set of allocated objects ($L^\natural_\sigma$), an environment mapping local variables to values ($\rho^\natural_\sigma$), a mapping from fields of allocated objects to values ($h^\natural_\sigma$), and a set of allocated array objects ($A^\natural_\sigma$).

We assume that program statements are labeled with unique labels. For an assignment statement at label $l \in \mathit{Labs}$, we denote by $lhs(l)$ the left hand side of the assignment, and by $rhs(l)$ the right hand side of the assignment.

We denote $Tasks(\sigma) = dom(pc^\natural_\sigma)$ to be the set of task identifiers in state σ, such that for each task identifier, $pc^\natural_\sigma$ assigns a value. We use $enabled(\sigma) \subseteq dom(pc^\natural_\sigma)$ to denote the set of tasks that can make a transition from σ.

3.1 Determinism

Determinism is generally defined as producing observationally equivalent outputs on all executions starting from observationally equivalent inputs.

In this paper, we establish determinism of parallel programs by proving that shared memory accesses made by statements in different tasks are independent. This is a stronger condition which sidesteps the need to define "observational equivalence", a notion that is often very challenging to define for real programs.

In the rest of the paper, we focus on the treatment of array accesses. The treatment of shared field accesses is similar (and simpler).

Definition 1 (Accessed array locations in a state). *Given a state $\sigma \in ST^\natural$, we define $W^\natural_\sigma$:* TaskIds $\rightarrow 2^{(A^\natural \times \mathbb{N})}$ *which maps a task identifier to the memory location to be written by the statement at label $pc_\sigma(t)$. Similarly, we define $R^\natural_\sigma$:* TaskIds $\rightarrow 2^{(A^\natural \times \mathbb{N})}$ *mapping a task identifier to the memory location to be read by the statement at $pc_\sigma(t)$:*

$$R^\natural_\sigma(t) = \{(\rho^\natural_\sigma(t,a), \rho^\natural_\sigma(t,i)) \mid pc^\natural_\sigma(t) = l \wedge rhs(l) = a[i]\}$$
$$W^\natural_\sigma(t) = \{(\rho^\natural_\sigma(t,a), \rho^\natural_\sigma(t,i)) \mid pc^\natural_\sigma(t) = l \wedge lhs(l) = a[i]\}$$
$$RW^\natural_\sigma(t) = R^\natural_\sigma(t) \cup W^\natural_\sigma(t)$$

Note that $R^\natural_\sigma(t)$, $W^\natural_\sigma(t)$ and $RW^\natural_\sigma(t)$ are always singleton or empty sets.

Definition 2 (Conflicting Accesses). *Given two shared memory accesses in states $\sigma_1, \sigma_2 \in ST^\natural$, performed respectively by task identifiers t_1 and t_2, we say that the two shared accesses are* conflicting*, denoted by $(\sigma_1, t_1) \nparallel (\sigma_2, t_2)$ when: $t_1 \neq t_2$ and $W^\natural_{\sigma_1}(t_1) \cap RW^\natural_{\sigma_2}(t_2) \neq \emptyset$ or $W^\natural_{\sigma_2}(t_2) \cap RW^\natural_{\sigma_1}(t_1) \neq \emptyset$.*

Next, we define the notion of a conflicting program. A program that is not conflicting is said to be *conflict-free*.

Definition 3 (Conflicting Program). *Given the set of all reachable program states $RS \subseteq ST^\natural$, we say that the program is* conflicting *iff there exists a state $\sigma \in RS$ such that $t_1, t_2 \in Tasks(\sigma)$, $mhp(RS, \sigma, t_1, pc^\natural_\sigma(t_1), t_2, pc^\natural_\sigma(t_2)) =$* true *and $(\sigma, t_1) \nparallel (\sigma, t_2)$.*

Informally, the above definition says that a program is conflicting if and only if there exists a state from which two tasks can perform memory accesses that conflict. Similar definition is provided by Shacham et. al [35]. However, our definition is more strict as we do not allow even atomic operations to conflict (recall that we currently do not handle atomic operations).

In the above definition we used the predicate $mhp : 2^{ST^\natural} \times ST^\natural \times$ *TaskIds* $\times$ *Labs* $\times$ *TaskIds* $\times$ *Labs* $\rightharpoonup Bool$. The predicate $mhp(S, \sigma, t_1, l_1, t_2, l_2)$ evaluates to *true* if t_1 and t_2 *may* run in parallel from state σ.

Computing mhp The computation of the mhp is parametric to our analysis. That is, we can consume an mhp of arbitrary precision. For instance, we can define $mhp(S, \sigma, t_1, l_1, t_2, l_2)$ to be *true* iff $t_1, t_2 \in enabled(\sigma)$ and $t_1 \neq t_2$.

We can also define less precise (more abstract) variants of mhp. For example, $mhp(S, \sigma, t_1, l_1, t_2, l_2) =$ *true* iff $\exists \sigma' \in S, t_1, t_2 \in enabled(\sigma'), t_1 \neq t_2$ such that $pc^\natural_{\sigma'}(t_1) = l_1$ and $pc^\natural_{\sigma'}(t_2) = l_2$. As the mhp depends on S and not on σ, we can write the mhp as $mhp(S, t_1, l_1, t_2, l_2)$. This less precise definition only talks at the level of labels and may be preferable for efficiency purposes. When the set S is assumed to be all reachable programs states, we write $mhp(t_1, l_1, t_2, l_2)$.

In this paper, we use the structure of the parallel program to compute the mhp precisely, but in cases where we consider general Java programs with arbitrary concurrency, we can also use more expensive techniques [26].

3.2 Pairwise Semantics

Next, we abstract away the relationship between the different tasks and define semantics that only tracks each task separately, rather than all tasks simultaneously.

We define the projection $\sigma|_t$ of a state σ on a task identifier t as $\sigma|_t = \langle pc|_t, L, \rho|_t, h, A\rangle$, where:

- $pc|_t$ is the restriction of pc to t
- $\rho|_t$ is the restriction of ρ to t

Given a state $\sigma \in ST^\natural$, we can now define the program state for a single task t via $\sigma|_t = \langle pc, L, \rho, h, A\rangle \in \Sigma$, where $ST^\natural_{pw} = (PC \times 2^{objs^\natural} \times Env^\natural \times Heap^\natural \times 2^{objs^\natural})$. For $S \subseteq ST^\natural$:

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

Next, we adjust our definition of a conflicting program.

Definition 4 (Pairwise-Conflicting Program). *Given the set of all reachable program states $RS^{pw} \subseteq ST^\natural_{pw}$, we say that the program is* pairwise conflicting *when there exists $\sigma^{pw}_1, \sigma^{pw}_2 \in RS^{pw}$ such that for some $t_1 \in Tasks(\sigma^{pw}_1)$, $t_2 \in Tasks(\sigma^{pw}_2)$, $mhp(RS^{pw}, t_1, pc^\natural_{\sigma^{pw}_1}(t_1), t_2, pc^\natural_{\sigma^{pw}_2}(t_2)) =$ true and $(\sigma^{pw}_1, t_1) \nleftrightarrow (\sigma^{pw}_2, t_2)$.*

Note that in this definition of a conflicting program, we use Definition 2 with *two* states σ^{pw}_1 and σ^{pw}_2, while in Definition 3, we used it only with a single state.

Assuming the mhp predicate computes identical results in Definition 3 and Definition 4, we now have the following simple theorem:

Theorem 1. *Any conflicting program is pairwise-conflicting.*

Of course, due to losing precision with the pairwise semantics, it could be the case that a program is pairwise-conflicting but not conflicting.

4 Abstract Semantics

The pairwise semantics tracks a potentially unbounded set of memory locations accessed by each task. In this section, we use standard abstract domains to represent sets of locations in a bounded way. We represent sets of objects using standard points-to abstractions, and ranges of array cells using numerical abstractions on array indices. Next, we abstract the semantics of Section 3.2.

4.1 Abstract State

Our abstraction is parametric on both the heap abstraction α_h and the numerical abstraction α_n. In the following, we assume an abstract numerical domain $ND = \langle NC, \sqsubseteq_{ND}\rangle$ equipped with operations $\sqcap_{ND}$ and $\sqcup_{ND}$, where NC is a set of numerical constraints over the primitive variables in *VarIds*, and do not go into further details about the particular abstract domain.

Definition 5. *An* abstract program state σ *is a tuple* $\langle pc, L_a, \rho_a, h_a, A_a, nc\rangle \in ST_a$*, where* $ST_a = PC \times 2^{objs} \times \text{Env} \times \text{Heap} \times 2^{objs} \times (TaskIds \rightarrow 2^{NC})$ *such that:*

- $L_a \subseteq$ *objs is a bounded set of abstract objects, and* $A_a \subseteq L_a$ *is a set of abstract array objects.*
- $\rho_a\colon TaskIds \times VarIds \rightarrow 2^{AVal}$ *maps a task identifier and a variable to its abstract values.*
- $h_a\colon$ *objs* $\times$ FieldId $\rightarrow 2^{AVal}$ *map an abstract location and a field identifier to their abstract values.*
- $nc\colon TaskIds \rightarrow 2^{NC}$ *maps a task to a set of numerical constraints, capturing relationship between local numerical variables of that task.*

An abstract program state is a sound representation of a concrete pairwise program state $\sigma^{pw} = \langle pc^{\natural}, L^{\natural}, \rho^{\natural}, h^{\natural}, A^{\natural}\rangle$ when:

- $pc = pc^{\natural}$.
- for all $o \in L^{\natural}$, $\alpha_h(o) \in L_a$.
- for all $o_1, o_2 \in L^{\natural}$, $f \in$ *FieldId*, if $h^{\natural}(o_1, f) = o_2$ then $\alpha_h(o_2) \in h_a(\alpha_h(o_1), f)$.
- $dom(\rho) = dom(\rho^{\natural})$
- for all task references $(t, r) \in dom(\rho^{\natural})$, if $v = \rho^{\natural}(t, r)$ then $\alpha_h(v) \in \rho_a(t, r)$.
- Let $TL_t = \{(pr_0, v_0)...(pr_n, v_n)\}$ be the set of primitive variable-value pairs, such that for all pairs $(pr_i, v_i) \in TL_t$, $(t, pr_i) \in dom(\rho^{\natural})$. Then $\alpha_n(TL_t) \sqsubseteq_{ND} nc(t)$.

Next, we define the accessed array locations in an abstract state:

Definition 6 (Accessed array locations in an abstract state). *Given an abstract state* $\sigma \in ST_a$*, we define* $W_\sigma\colon \text{TaskIds} \rightarrow 2^{(AVal \times \text{VarIds})}$ *which maps a task identifier to the memory location to be written by the statement at label* $pc_\sigma(t)$*. Similarly, we define* $R_\sigma\colon \text{TaskIds} \rightarrow 2^{(AVal \times \text{VarIds})}$ *mapping a task identifier to the memory location to be read by its statement at* $pc_\sigma(t)$*.*

$$
\begin{aligned}
&R_\sigma(t) = \{(\rho_\sigma(t, a), i) \mid pc_\sigma(t) = l \wedge rhs(l) = a[i]\} \\
&W_\sigma(t) = \{(\rho_\sigma(t, a), i) \mid pc_\sigma(t) = l \wedge lhs(l) = a[i]\} \\
&RW_\sigma(t) = R_\sigma(t) \cup W_\sigma(t)
\end{aligned}
$$

Note that R_σ, W_σ and RW_σ are always singleton or empty sets. We use $D_\sigma(t).B$ and $D_\sigma(t).I$ to denote the first and second components of the entry in the singleton set D, where D can be one of R, W or RW. If $D_\sigma(t)$ is empty, then $D_\sigma(t).B$ and $D_\sigma(t).I$ also return the empty set. Next, we define the notion of conflicting accesses:

Definition 7 (Abstract Conflicting Accesses). *Given two shared memory accesses in states* $\sigma_1, \sigma_2 \in ST_a$*, performed respectively by task identifiers* t_1 *and* t_2*, we say that the two shared accesses are* conflicting*, denoted by* $(\sigma_1, t_1) \nleftrightarrow_{abs} (\sigma_2, t_2)$ *when:*

- $W_{\sigma_1}(t_1).B \cap RW_{\sigma_2}(t_2).B \neq \emptyset$ *and* $(W_{\sigma_1}(t_1).I = RW_{\sigma_2}(t_2).I) \sqcap_{ND} AS \neq \bot$ *or*
- $W_{\sigma_2}(t_2).B \cap RW_{\sigma_1}(t_1).B \neq \emptyset$ *and* $(W_{\sigma_2}(t_2).I = RW_{\sigma_1}(t_1).I) \sqcap_{ND} AS \neq \bot$

where $AS = nc_{\sigma_1}(t_1) \sqcap_{ND} nc_{\sigma_2}(t_2) \sqcap_{ND} (t_1 - t_2 \geq 1)$

The definition uses the meet operation $\sqcap_{ND}$ of the underlying numerical domain to check whether the combined constraints are satisfiable. If the result is not empty (e.g. not $\bot$), then this indicates a potential overlap between the array indices. The constraint of the kind $(W.I = RW.I)$ corresponds to the property we are trying to refute, namely that the indices are equal. In addition, we add the global constraint that requires that task identifiers are distinct. The reason why we write that constraint as $(t_1 - t_2 \geq 1)$ as opposed to $(t_1 - t_2 > 0)$ is that the first form is precisely expressible in both Octagon and Polyhedra, while the second form is only expressible in Polyhedra. We assume that primitive variables from two different tasks have distinct names.

The definition of abstract conflicting accesses leads to a natural definition of *abstract pairwise conflicting program* based on Definition 4. Due to the soundness of our abstraction it follows that if we establish the program as abstract (pairwise) conflict free, then it is (pairwise) conflict free under the concrete semantics.

In the next section, we describe our implementation which is based on a sequential analysis of each task, computing the reachable abstract states of a task in the absence of interference from other tasks. We then (conservatively) check that tasks perform independent memory accesses. When tasks may be performing conflicting memory accesses, the sequential information computed may be invalid, and our analysis will not be able to establish determinism of the program. When tasks are only performing non-conflicting memory accesses, the information we compute sequentially for each task is stable, and we can use it to establish the determinism of the program.

5 Implementation

We implemented our analysis as a tool based on the Soot framework [36]. This allows us to potentially use many of the existing analyses already implemented in Soot, such as points-to analyses. The input to our tool is a Java program with annotations that denote the code of each task. In fact, as our core analysis is based purely on the Jimple intermediate representation produced by the Soot front end, as long as it knows what code each task executes, the analysis is applicable to standard concurrent Java programs.

The complete analysis works as follows:

Step 1: Apply Heap Abstraction First, we apply the SPARK flow-insensitive pointer analysis on the whole program [20]. We note that flow-insensitive analysis is sound in the presence of concurrency, but as we will see later, the pointer analysis may be imprecise in most cases and hence we compute additional heap information (see the UniqueRef invariant later).

Step 2: Apply Numerical Abstraction Second, for each task, we apply the appropriate (sequential) numerical analysis. Our numerical analysis uses the Java binding of the Apron library [15]. We initialized the environment of the analysis only with variables of integer type. As real variables cannot be used as array indices, they are ignored by the analysis. Currently, we do not handle casting from real to integer variables. However in our benchmarks we have not encountered such cast operations. The numerical constraints contain only variables of integer type.

Step 3: Compute MHP Third, we compute the mhp predicate. In the annotated Java code that we consider, this is trivially computed as the annotations denote which tasks can execute in parallel and given that parallel tasks don't use any synchronization constructs internally, it implies that all statements in two parallel tasks can also execute in parallel. When analyzing standard Java programs which use synchronization primitives such as monitors, one can use an off-the-shelf MHP analysis (cf. [21,26]).

Step 4: Verify Conflict-Freedom Finally, we check whether the program is conflict-free: for each pair of abstract states from two different tasks, we check whether that pair is conflict-free according to Definition 7. In our examples, it is often the case that the same code is executed by multiple tasks. Therefore, in our implementation, we simply check whether the abstract states of a single task are conflict-free with themselves. To perform the check, we make sure that local variables are appropriately renamed (the underlying abstract domain provides methods for this operation). Parameter variables that are common to all tasks that execute the same code maintain their name under renaming and are distinguished by special names. Note that our analysis verifies conflict-freedom between tasks in a pairwise manner, and does not make any assumption on the number of tasks in the system (thus also handling programs with an unbounded number of tasks).

5.1 Reference Arrays

Many parallel programs use reference arrays, usually multi-dimensional primitive arrays (e.g. `int A[][]`) or reference arrays of standard objects such as `java.lang.String`. In Jimple (and Java bytecodes), multi-dimensional arrays are represented via a hierarchy of one-dimensional arrays. Accesses to a k-dimensional array is comprised of k accesses to one-dimensional arrays. In many of our benchmarks, parallel tasks operate on disjoint portions of a reference array. However, often, given an array `int A[][]`, each parallel task accesses a different outer dimension, but accesses the internal array `int A[]` in the same way. For example, task 1 can write to `A[1][5]`, while task 2 can write to `A[2][5]`): the outer dimension (e.g. 2) is different, but the inner dimension (e.g. 5) is the same. The standard pointer analysis fails to establish that `A[1][5]` and `A[2][5]` are disjoint, and hence our analysis fails to prove determinism.

UniqueRef Global Invariant However, in all of our benchmarks, the references inside reference arrays never alias. This is common among scientific computing benchmarks as they have a pre-initialization phase where they fill up the array, and thereafter, only the primitive values in the array are modified. To capture this, on startup, we perform a simple global analysis to establish that all writes of reference variables to cells in the reference array are only assigned to once with a fresh object, either a newly allocated object or a reference obtained as a result of a library call such as `java.lang.String.substring` that returns a fresh reference. While this simple treatment suffices for all of our benchmarks, general treatment of handling references inside objects may require more elaborate heap analysis.

Once we establish this invariant, we can either refine the pointer analysis information (to know that the inner dimensions of an array are distinct), or we can use the invariant directly in the analysis. In almost all of our benchmarks, we used this invariant directly.

6 Evaluation

To evaluate our analysis, we selected the JGF benchmarks used by the HJ suite [1]. These benchmarks are modified versions of the Java JGF benchmarks [9]. As currently our numerical analysis is intra-procedural, we have slightly modified these benchmarks by inlining some of the function calls. The code for all benchmarks is available in [1].

Our analysis works on the Jimple intermediate representation, which is a three-address code representation for Java. Working at the Jimple level enables us to use standard analyses implemented in Soot, such as the Spark points-to analysis [20]. However, Jimple creates a large number of temporary variables, resulting in many more variables than the original Java source. This may lead to a larger number of numerical constraints compared to the ones arising when analyzing the program at the source level as in the case of the Interproc analyzer [16].

Analysis of some of our benchmarks required the use of widening. We used the `LoopFinder` API provided by Soot to identify loops and applied a basic widening strategy which only widens at the head of the loop and does so every k'th iteration, where k is a parameter to the analysis.

All of our experiments were conducted using a Java 1.6 runtime running on a 4-core Intel(R) Xeon(TM) CPU 3.80GHz processor with 5GB.

6.1 Results

Benchmark	Description	LOC	Vars	Domain	Iter	Time (s)	Widen	PA	Result
CRYPT	IDEA encryption	110	180	Polyhedra	439	54.8	No	No	✓
SOR	Successive over-relaxation	35	21	Polyhedra	72	0.41	Yes	No	✓
LUFACT	LU Factorization	32	22	Octagon	57	1.94	Yes	No	✓
SERIES	Fourier coefficient analysis	67	14	Octagon	22047	55.8	No	No	✓
MOLDYN1	Molecular dynamics simulation	85	18	Octagon	85	24.6	No	No	✓
MOLDYN2		137	42	Polyhedra	340	2.5	Yes	Yes	✓
MOLDYN3		31	8	Octagon	78	0.32	Yes	No	✓
MOLDYN4		50	10	Polyhedra	50	1.01	No	No	✓
MOLDYN5		37	18	Polyhedra	37	0.34	No	No	✓
SPARSE	Sparse matrix multiplication	29	17	Polyhedra	45	0.2	Yes	Yes	✗
RAYTRACER	3D Ray Tracer	-	-	-	-	-	-	-	✗
MONTECARLO	Monte Carlo simulation	-	-	-	-	-	-	-	✗

Table 2. Experimental Results

Table 2 summarizes the results of our analysis. The columns indicate the benchmark name and description, lines of code for the analyzed program, the number of

integer-valued variables used in the analysis, the numerical domain used, the number of iterations it took for the analysis to reach a fixed point, the combined time of the numerical analysis and verification checking (pointer analysis time is not included even if used), whether the analysis needed widening to terminate, whether we used Spark pointer analysis (note that we almost always use the UniqueRef invariant as the programs make heavy use of multi-dimensional arrays), and the result of the analysis where ✓ denotes that it successfully proved determinism, and ✗ denotes that it failed to do so. As mentioned earlier, in our benchmarks, it is easy to pre-compute the mhp predicate and determine which tasks can run in parallel. That is, there is no need to perform numerical analysis on tasks that can never run in parallel with other tasks. Therefore, the lines of code in the table refer only to the relevant code that may run in parallel and is analyzed by the numerical analysis. The actual applications contain many more lines of code (in the range of thousands), as they need to pre-initialize the computation, but such initialization code never runs in parallel with other tasks.

Applying the analysis For every benchmark, we first attempted to verify determinism with the simplest available configuration: e.g. Octagon domain without widening or pointer analysis. If the analysis did not terminate within 10 minutes, or failed to prove the program deterministic, then we tried adding widening and/or changing the domain to Polyhedra and/or performing pointer analysis. Usually, we did not find the need for using Spark. Instead, we almost always rely on the UniqueRef invariant.

For five of the benchmarks, the analysis managed to prove determinism, while it failed to do so for three benchmarks. Next, we elaborate on the details.

CRYPT involves reading and updating multiple shared one-dimensional arrays. This is a computationally intensive benchmark and its intermediate representation contains many variables. When we used the Octagon domain without widening, the analysis did not terminate and the size of the constraints kept growing. Even after applying our widening strategy (widening at the head of the loop) with various frequencies (e.g. the parameter *k* mentioned earlier), we still could not get the analysis to terminate. Only after applying very aggressive widening: in addition to every loop head, to widen at some points in the loop body, did we get the analysis to terminate. But even when it terminated, the analysis was unable to prove determinism. The key reason is that the program computes array indices for each task based on the task identifier via statements such as $ix_i = 8{*}tid_i$, where ix_i is the index variable and tid_i is the task identifier variable. Such constraints cannot be directly expressed in the Octagon domain. However, by using the Polyhedra domain, the analysis managed to terminate without widening. It managed successfully to capture the simple loop exit constraint $ix_i \geq k$ (even with the loop body performing complicated updates). It also managed to successfully preserve constraints such as $ix_1 = 8 * tid_1$. As a result, the computed constraints were precise enough to prove the program deterministic, which is the result that we report in the table.

In SOR, without widening, both Octagon and Polyhedra failed to terminate. With widening, Octagon failed to prove determinism due to the use of array index expressions such as $ix_i = 2 * tid_i - v$, where tid_i is the task identifier variable and v is a parameter variable. Constraints, such as $i_i = k * tid_i$, where $k > 1$ cannot be expressed in the Octagon domain and hence the analysis fails. Using Polyhedra with widening quickly succeed in proving determinism.

Without widening and with Octagon, the analysis did not terminate in LUFACT. However, with widening and Octagon, the analysis quickly reached a fixed point. The SERIES benchmark succeeds only with Octagon and required no widening but it took the longest to terminate.

MOLDYN contains a sequence of five blocks where only the tasks inside each block can run in parallel and tasks from different blocks cannot run in parallel. Interestingly, each block of tasks can be proved deterministic by using different configurations of domain, widening and pointer analysis. In Table 2, we show the result for each block as a separate row in the table. In the first block, tasks execute straight line code and determinism can be proved only with Octagon and no widening. In the second block, tasks contain loops and require Polyhedra, widening and pointer analysis. Without widening, both Octagon and Polyhedra do not terminate. With widening, Octagon terminates, but fails. The problem is that the array index variable ix_i is computed with the statement $ix_i = k * tid_i$, where k is a constant and $k > 1$. The Octagon domain cannot accurately represent abstract elements with such constraints. We used the pointer analysis to establish that references loaded from two different arrays are distinct, but we could have also computed that with the UniqueRef invariant. Tasks in the third block succeed with Octagon but also required widening. Tasks in the fourth and fifth blocks do not need widening (there are no loops), but require Polyhedra as they are using constraints such as $ix_i = k * tid_i$, where $k > 1$.

In SPARSE, the Polyhedra fails as the tasks use array indices obtained from other arrays, e.g. $A[B[i]]$, where the values of $B[i]$ are initialized on startup. The analysis required widening to terminate, but is unable to establish anything about $B[i]$ and hence cannot prove independence of two different array accesses $A[j]$ and $A[k]$, where j and k come from some $B[i]$.

In RAYTRACER, analysis fails as the program involves non-linear constraints and also uses atomic sections, which our analysis currently does not handle.

As mentioned, our analysis is intra-procedural. However, unlike the other benchmarks, MONTECARLO makes many nested calls and it would have been very error-prone to try and inline all of these nested calls. To handle such cases, in the future, we plan to extend our analysis to handle procedures.

6.2 Summary

In summary, in all cases where the analysis was successful in proving determinacy, it finished in under a minute. Different benchmarks could be proved with different combination of domain (Octagon or Polyhedra) and widening (to widen or not). In fact, the suite exercised all four combinations. In general, we did not find that we needed expensive pointer analysis, and it was sufficient to have the simple invariant that all arrays contain unique references, which was easily verifiable for our benchmarks (but in general may be a very hard problem). In cases where we failed, the program was using features that we do not currently handle such as: non-linear constraints, atomic sections, procedure calls or required maintaining scalar invariants over arrays (e.g. that integers inside an array are distinct). In the future, we plan to address these issues. This would also allow us to handle the full Java JGF benchmarks [9], where many benchmarks make use of such features.

7 Related Work

Recent papers by Burnim and Sen [5] and Sadowski et. al [33] focus on checking determinism dynamically. The first work focuses on user-defined notion of observationally equivalent states while the second paper checks for absence of conflicts. While both of these works are only able to dynamically *test* for determinism, our work focus on statically proving determinism.

There is a vast literature on dependence analysis for automatic parallelization (see e.g., [24, Sec. 9.8]). The focus of these analyses is on *efficiently* identifying independent loop iterations that can be performed in parallel. In contrast, our work focuses on verifying determinism of parallel programs. This usually involves dependence checking between tasks that may execute in parallel (and not necessarily in a loop). As we focus on verification, we can employ precise (and often expensive) numerical domains.

There is a large volume of work on dependence analysis for heap-manipulating programs (e.g., [14]), which at the end boils down to having a sufficiently precise heap abstraction (e.g., [29]). The current task-parallel applications we are dealing with mostly involve numerical computations over arrays. For such programs, simple heap abstractions were sufficient for establishing determinism. In the future, we plan to integrate more advanced heap abstractions into our analysis framework.

In [31,32], Rugina and Rinard present an analysis framework for computing symbolic bounds on pointer, array indices, and accesses memory regions. In order to support challenging features such as pointer arithmetic, their analysis framework requires an expensive flow-sensitive and context-sensitive pointer analysis [30] as a preceding phase. In our (simpler) setting, this is not required.

In [12], Ferrera presents a static analysis for establishing determinism of concurrent programs. The idea is to record for each variable what thread wrote it. This instrumented concrete semantics is then abstracted to an abstract semantics that records separately the value written to a variable by each thread. Determinism is established by comparing (abstract) states and showing that for each variable, its value is only determined by a single thread. The paper does not handle arrays, dynamic memory allocation, and assumes a bounded number of threads. Further, Ferrera's analysis is based on concurrent executions and therefore has to consider all possible interleavings. In contrast, using basic assume-guarantee reasoning, our analysis reduces the problem to a sequential analysis.

8 Conclusion

We present a static analysis for automatically verifying determinism of structured parallel programs. Our approach uses *sequential analysis* to establish that tasks that may execute in parallel only perform non-conflicting memory accesses. Our sequential analysis combines information about the heap with information about array indices to show that memory accesses are non-conflicting. We show that in realistic programs, establishing that accesses are non-conflicting requires powerful numerical domains such as Octagon and Polyhedra. We implemented our approach in a tool called DICE and applied it to verify determinism of several non-trivial benchmark programs. In the future, we plan to extend our analysis to handle general Java programs.

Acknowledgements We thank the anonymous reviewers for their helpful comments that improved the paper, and Antoine Mine for helping us with using Apron.

References

1. Dojo: Ensuring determinism of concurrent systems. https://researcher.ibm.com/researcher/view_project.php?id=1337.
2. BANERJEE, U. K. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.
3. 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* (Oct. 1995), pp. 207–216.
4. BOCCHINO, R., ADVE, V., ADVE, S., AND SNIR, M. Parallel programming must be deterministic by default. In *First USENIX Workship on Hot Topics in Parallelism (HOTPAR 2009)* (2009).
5. BURNIM, J., AND SEN, K. Asserting and checking determinism for multithreaded programs. In *ESEC/FSE '09* (2009), ACM, pp. 3–12.
6. CHARLES, P., GROTHOFF, C., SARASWAT, V. A., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA* (Oct. 2005), pp. 519–538.
7. COUSOT, P., AND HALBWACHS, N. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Tucson, Arizona, 1978), ACM Press, New York, NY, pp. 84–97.
8. DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. Dmp: deterministic shared memory multiprocessing. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems* (2009), ACM, pp. 85–96.
9. EDINBURGH PARALLEL COMPUTING CENTRE. Java grande forum benchmark suite. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.
10. EDWARDS, S. A., AND TARDIEU, O. Shim: a deterministic model for heterogeneous embedded systems. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software* (2005), ACM, pp. 264–272.
11. FENG, M., AND LEISERSON, C. E. Efficient detection of determinacy races in cilk programs. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures* (1997), ACM, pp. 1–11.
12. FERRARA, P. Static analysis of the determinism of multithreaded programs. In *Proceedings of the Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008)* (November 2008), I. C. Society, Ed.
13. FLANAGAN, C., AND FREUND, S. N. Fasttrack: efficient and precise dynamic race detection. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (2009), ACM, pp. 121–133.
14. HORWITZ, S., PFEIFFER, P., AND REPS, T. Dependence analysis for pointer variables. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation* (1989), ACM, pp. 28–40.
15. JEANNET, B., AND MINE, A. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification* (2009), vol. 5643 of *LNCS*, Springer Berlin / Heidelberg, pp. 661–667.

16. LALIRE, G., ARGOUD, M., AND JEANNET, B. The interproc analyzer. http://pop-art.inrialpes.fr/interproc/interprocweb.cgi.
17. LAMPORT, L. The parallel execution of do loops. *Commun. ACM 17*, 2 (1974), 83–93.
18. LEA, D. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande* (2000), ACM, pp. 36–43.
19. LEE, E. A. The problem with threads. *Computer 39*, 5 (2006), 33–42.
20. LHOTK, O., AND HENDREN, L. Scaling java points-to analysis using spark. In *Compiler Construction* (2003), vol. 2622 of *LNCS*, Springer, pp. 153–169.
21. LI, L., AND VERBRUGGE, C. A practical MHP information analysis for concurrent java programs. In *Languages and Compilers for High Performance Computing* (2005), vol. 3602 of *LNCS*, Springer, pp. 194–208.
22. MARINO, D., MUSUVATHI, M., AND NARAYANASAMY, S. Literace: effective sampling for lightweight data-race detection. In *PLDI '09* (2009), ACM, pp. 134–143.
23. MINÉ, A. The octagon abstract domain. *Higher Order Symbol. Comput. 19*, 1 (2006), 31–100.
24. MUCHNICK, S. S. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
25. NAIK, M., AIKEN, A., AND WHALEY, J. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (2006), ACM, pp. 308–319.
26. NAUMOVICH, G., AVRUNIN, G. S., AND CLARKE, L. A. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the joint 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Sept. 1999), pp. 338–354.
27. O'CALLAHAN, R., AND CHOI, J.-D. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming* (2003), ACM, pp. 167–178.
28. OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09* (2009), ACM, pp. 97–108.
29. RAZA, M., CALCAGNO, C., AND GARDNER, P. Automatic parallelization with separation logic. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 348–362.
30. RUGINA, R., AND RINARD, M. Automatic parallelization of divide and conquer algorithms. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (1999), ACM, pp. 72–83.
31. RUGINA, R., AND RINARD, M. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (2000), ACM, pp. 182–195.
32. RUGINA, R., AND RINARD, M. C. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program. Lang. Syst. 27*, 2 (2005), 185–235.
33. SADOWSKI, C., FREUND, S. N., AND FLANAGAN, C. SingleTrack: A dynamic determinism checker for multithreaded programs. In *Programming Languages and Systems* (2009), vol. 5502 of *LNCS*, Springer Berlin / Heidelberg, pp. 394–409.
34. SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (1997), 391–411.
35. SHACHAM, O., SAGIV, M., AND SCHUSTER, A. Scaling model checking of dataraces using dynamic information. In *PPoPP '05* (2005), ACM, pp. 107–118.
36. VALLEE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND CO, P. Soot - a java optimization framework. In *Proceedings of CASCON 1999* (1999), pp. 125–135.