Array Dataflow Analysis for Polyhedral X10 Programs *

Tomofumi Yuki
Colorado State University
yuki@cs.colostate.edu

Paul Feautrier
LIP (ENS Lyon, INRIA, CNRS, UCBL)
paul.feautrier@ens-lyon.fr

Sanjay Rajopadhye
Colorado State University
Sanjay.Rajopadhye@colostate.edu

Vijay Saraswat
IBM Research
vijay@saraswat.org

Abstract

This paper addresses the static analysis of an important class of X10 programs, namely those with finish/async parallelism, and affine loops and array reference structure as in the polyhedral model. For such programs our analysis can certify whenever a program is deterministic or flags races.

Our key contributions are (i) adaptation of array dataflow analysis from the polyhedral model to programs with finish/async parallelism, and (ii) use of the array dataflow analysis result to certify determinacy. We distinguish our work from previous approaches by combining the precise statement instance-wise and array element-wise analysis capability of the polyhedral model with finish/async programs that are more expressive than doall parallelism commonly considered in the polyhedral literature. We show that our approach is exact (no false negative/positives) and more precise than previous approaches, but is limited to programs that fit the polyhedral model.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors — Compilers, Debuggers; D.3.3 [*Programming Languages*]: Language Constructs and Features — Concurrent programming structures; D.2.4 [*Software Engineering*]: Software/Program Verification — Validation

Keywords X10; Parallelism; Non-determinism; Array Data-flow Analysis; Polyhedral Model; Happens-Before; Race Detection; Execution Partial Order

1. Introduction

Because parallelism has gone mainstream, the problem of improving parallel programmer productivity is now increasingly important. It was the goal of the DARPA HPCS program, initially for the supercomputing niche, but is no longer a niche problem. A number of new parallel programming languages are being actively developed and explored [2, 6, 8, 18, 22, 30]. These languages all employ new programming models to ease the parallel programming effort.

* This work was funded in part by the National Science Foundation, Award Numbers: 1240991 and 0917319

PPoPP'13, February 23–27, 2013, Shenzhen, China.

While designing and writing parallel programs is significantly harder than its sequential counterpart, debugging is even harder. This is due to the non-deterministic nature of parallel execution, and the accompanying difficulty of reproducing errors. Therefore the problem of static analysis of parallel programs is becoming critical. Some recent parallel languages also attempt to ease the debugging effort. For example, Titanium [30] can check for possible deadlocks at compile time. Most parallel constructs in X10 [22] are designed such that no (logical) deadlocks occur.

In this paper, we present static race detection for a subset of X10 programs. By providing race-free guarantee, we can prove determinacy of a program. The ability to detect races statically will allow programmers to correct them as they write the program, greatly improving their productivity. The subset of X10 we consider includes its core parallel constructs, `async` and `finish`. We also require that the loop bounds and array accesses to be affine to fit the *polyhedral model*, a mathematical framework for reasoning about program transformations [13]. Although it considers a restricted class of programs, this model has proven very effective and has found widespread use in automatic parallelization in high performance computing.

We improve upon previous techniques for static data race detection in two key directions:

- Our analysis is statement *instance-wise* and array *element-wise*. Most existing approaches (e.g., [9, 15]) analyze race of static statements and conservatively flags as race if two statements that may happen in parallel access the same variable. Our analysis will find sets of statement *instances* (i.e., statement executed when loop counters take specific values) that may happen in parallel, and only flags as race if they access the same *element* of an array.
- In comparison with other methods that support both instance-wise and element-wise analysis [5, 7], our work supports parallelism based on finish/async constructs, which are more expressive than *doall* type parallelism considered in prior work.

Consider the following code fragment that uses the `async` construct of X10. An `async` spawns a new activity to execute the enclosed statement. The spawning activity cannot proceed beyond the end of an enclosing `finish` block—another of X10's constructs—until all activities that it has spawned have terminated.

```
for (i in 0..N) {
   S0
   async S1
}
```

The above code sequentially executes `S0` at each iteration of the loop, and after executing `S0`, spawns a new activity to execute an instance of `S1`. Such parallelism cannot be expressed as *doall* type parallelism, and no clean way to express it as a *schedule* in the polyhedral model is currently available.

Our race detection is based on an adaptation of array dataflow analysis [11] from the polyhedral model. This allows us to have the same level of precisions as other work based on the polyhedral model, but is applicable to finish/async based parallelism. Specifically, the key contributions of this paper are:

- A very simple formulation of the operational semantics for this fragment of X10, significantly simplifying [16, 23]. This directly leads to a simple definition of the "happens-before" (HB) and "may-happen-in-parallel" (MHP) relation on statements.
- Characterization of the HB relation as an *incomplete lexicographical order*. This is the key to reuse techniques from the polyhedral model for X10 programs.
- Adaptation of Array Dataflow Analysis [11] for X10 programs. Array dataflow analysis answers the question, which instance of which statement produced the value being used for each statement instance. We have extended this analysis to handle X10 programs.
- Race Detection using array dataflow results. The key idea is that once we have solved the dataflow problem, we can identify the set of instances that cause a race by pinpointing the set of array cells which have multiple producers.
- Prototype implementation of a verifier for our subset of X10. If a program has races, our tool can tell precisely which statement instances are involved.

2. A Subset of X10

Our main interest is in intra-procedural analysis, and we wish to address the integration of finish/async concurrency with loops over array based data-structures in a pure form. Hence we will only consider assignment statements, sequencing, finish, async and for loops, and variables that range over integers and arrays of integers. This allows us to state certain properties of the key relations—Happens Before and its closely related May Happens in Parallel—in a pure form. Our formal treatment of the semantics can be extended *mutatis mutandis* with conditionals, local variables, potentially infinite loops, method calls, objects and functions etc., but we restrain ourselves to the subset we can analyze with polyhedral machinery.

The subset of X10 [22] we consider consists of the following control constructs:

- Sequence ($\{ST\}$): Composes two statements in sequence.
- Sequential `for` loop: We assume all loops have an associated loop iterator. X10 loops may scan a multidimensional iteration space. However, we assume that such loops have been expanded into a nest of unidimensional loops.
- Parallel activation, `async`: The body of the `async` is executed by an independent lightweight thread, called `activity` in X10.
- Termination detection, `finish`: An activity waits for all activities spawned within the body of `finish` to terminate before proceeding further. In addition, each program has an implicit `finish` as its top level construct.

We also require the program to fit the polyhedral model. The polyhedral model requires loop bounds, and array access to be affine expressions of the surrounding loop indices. Multidimensional arrays in X10 and Java are in fact trees of one-dimensional arrays. As such, they support many operations beyond simple subscripting. An example is row interchange in constant time. Detection of such uses is beyond the scope of this paper; see for instance [29] for an abstract interpretation approach.

Note that the full language permits some additional constructs: for instance the (conditional) atomic block `(when(c) S)` . This construct permits data-dependent synchronization in general and barrier-style *clocks* [23] in particular. The `at` construct permits computation across multiple places. We leave the integration of these statements into the analysis of this paper for future work.

2.1 Operational Semantics

We provide a simple, concise structural operational semantics (SOS) for the fragment of X10 considered in this paper. This semantics is considerably simpler than [23] because it eschews the "Middleweight Java" approach in favor of directly specifying semantics on statements. Unlike [16] there is no need to translate the statement to be executed into different kinds of tree-like structure; the information is already contained in the lexical structure of the statement and can be elegantly exploited using SOS based structural rules.

In this section, we present the semantics, characterize certain syntactic properties of statements (the happens-before and the may-happen-in-parallel relation), and relate them to behavioral properties. For simplicity of exposition, we chose to use a sequentially consistent memory model. In future work we expect to apply the methods of [24] to adapt the analysis techniques developed in this paper to relaxed memory models.

We assume that a set of (typed) locations `Loc`, and a set of values, `Val`, is given. `Loc` typically includes the set of variables in the program under consideration. With every d-dimensional $N_1 \times \ldots \times N_d$ array-valued variable `a` of type array are associated a set of distinct locations, designated `a(0,...,0), ..., a(N1-1, ..., Nd-1)`. The set of values includes integers and arrays.

A *heap* is a partial (finite) mapping from `Loc` to `Val`. For h a heap, l a location and v a value by $h[l = v]$ we shall mean the heap that is the same as h except that it takes on the value v at l. By $h(l)$ we mean the value e to which h maps l.

DEFINITION 2.1 (Expressions). *We assume that a set of* RHS expressions *(ranged over by $e, e', e_0, e_1, \ldots$) that denote values is defined. RHS expression include variables (e.g., `x`), literals (e.g., `0`), array accesses (e.g., `a(p)`), and appropriate operations over integers (e.g., addition). We also assume that a set of* LHS expressions *(ranged over by $a, a', a_0, a_1, \ldots$) that denote locations is defined. These include variables and array accesses.*[1] *We extend h to a map from RHS expressions to values and from LHS expressions to locations in the obvious way.*

DEFINITION 2.2 (Statements). *The statements are defined by the productions:*

(Statements) S	::=
T	Execute T.
$\{T\ S\}$	Execute T then S.
T	::=
$a = e$;	Assignment.
`for`$(x$ `in` $e1..e2)\ S$	Execute S for x in $e1 \ldots e2$.
`async` S	Spawn S.
`finish` S	Execute S and wait for termination.

[1] Thus, as is conventional in modern imperative languages, the notation `a(i)` is ambiguous. When used on the LHS it represents a location and when used on the RHS it represents the contents of that location.

DEFINITION 2.3 (Paths). *The set of paths $\mathcal{P}[\![S]\!]$ corresponding to a statement S is given as follows. For a set of paths U, we let xU stand for the set of paths xs, for $s \in U$.*

$$\begin{array}{rcl}
\mathcal{P}[\![a = e]\!] & = & \{\varepsilon\} \\
\mathcal{P}[\![\{S\,T\}]\!] & = & \{\varepsilon\} \cup 0\,\mathcal{P}[\![S]\!] \cup 1\,\mathcal{P}[\![T]\!] \\
\mathcal{P}[\![\texttt{for}(x\ \texttt{in}\ e1..e2)\ S]\!] & = & \{\varepsilon\} \cup x\,\mathcal{P}[\![S]\!] \\
\mathcal{P}[\![\texttt{async}\ S]\!] & = & \{\varepsilon\} \cup a\,\mathcal{P}[\![S]\!] \\
\mathcal{P}[\![\texttt{finish}\ S]\!] & = & \{\varepsilon\} \cup f\,\mathcal{P}[\![S]\!]
\end{array}$$

The statements for which the operational semantics is defined are assumed to satisfy some static semantic conditions (e.g., well-typedness). We omit the details. Note that in a `for` loop the index variable is considered bound. To avoid dealing with alpha renaming, we assume that in the statement under consideration no two loop index variables are the same.

Note that the set of paths for a statement is non-empty and prefix-closed, hence defines a tree. A path $p \in \mathcal{P}[\![S]\!]$ is *terminal* if it is not a proper prefix of any other in $\mathcal{P}[\![S]\!]$. For example, in Fig. 1 $[0, f, 0, i, a]$ is the (terminal) path for statement $S0$, and $[0, f, 1]$ is a (non-terminal) path.

From the definition of statements and paths, the only place that two paths may diverge is at a sequential composition. For a statement S, let sx and sy be two distinct paths in $\mathcal{P}[\![S]\!]$. Note that paths in $\mathcal{P}[\![S]\!]$ are *symbolic*, since loop iterators are variables. We define an *instance* of a path as $t = s\theta$, where θ is a substitution applied to $s \in \mathcal{P}[\![S]\!]$ mapping index variables to integers.[2] Now, two *path instances* sx and sy can diverge either when they follow different branches via sequantial composition, or when they have different values of loop iterators.

PROPOSITION 2.1. *For a statement S, let sx and sy be two distinct instances of paths in $\mathcal{P}[\![S]\!]$. Then either $x < y$ or $y < x$.*

We also introduce a notation to denote sub-statements. Let S be a statement and $s \in \mathcal{P}[\![S]\!]$. We use the notation $S\hat{}s$ to refer to the sub-statement of S obtained by traversing the path s from the root. Given a path instance $t = s\theta$, we define $S\hat{}t$ to be $(S\hat{}s)\theta$, that is, θ applied to the statement obtained by traversing the path s from the root of S. This definition is justified by the fact that θ is unique for each path instance.

DEFINITION 2.4 (Read and write set). *Let S be a statement, and $s \in \mathcal{P}[\![S]\!]$ a terminal path (or path instance). We let $rd(S,s)$ denote the set of locations read by $S\hat{}s$ and $wr(S,s)$ the set of locations written in $S\hat{}s$.*

Let s,t be two paths or path instances for S. We say s write-affects *t if $wr(S,s) \cap (rd(S,t) \cup wr(S,t))$ is non-empty. We say that s and t* conflict *if s write-affects t or* vice versa*. We say that t* self-conflicts *if $rd(S,t) \cup wr(S,t)$ is non-empty.*

For instance, let S be the statement $\texttt{for}(i\ \texttt{in}\ 0..10)\ a(i) = a(i) + 1$. Then the path $[\varepsilon]$ self-conflicts, as does $[i]$. But the path (instance) $[0]$ does not. In fact the paths $[i]$,$[j]$ do not conflict if i, j are distinct integers (in the given range.)

Note that $S\hat{}s$ may be a statement with free variables (e.g., parameters), hence the set of locations read/written may be symbolic (i.e., heap-dependent at run-time.)

Execution relation. As is conventional in SOS, we shall take a *configuration* to be a pair $\langle S,h\rangle$ (representing a state in which S has to be executed in the heap h) or h (representing a terminated computation.)

[2] Usually, we will be concerned only with path instances that satisfy the bounds conditions for the index variable. Note that given S, $s \in \mathcal{P}[\![S]\!]$, and an instance $t = s\theta$ note that θ can be recovered uniquely.

The operational execution relation $\longrightarrow$ is defined as a binary relation on configurations. We use the "matrix" convention for presenting rules compactly. A rule such as:

$$\frac{\begin{array}{c} c_0, \ldots, c_{p-1} \\ \gamma \longrightarrow \gamma_0 \mid \ldots \mid \gamma_{n-1} \end{array}}{\begin{array}{l} \gamma^0 \longrightarrow \delta_0^0 \mid \ldots \mid \delta_{n-1}^0 \\ \ldots \\ \gamma^{m-1} \longrightarrow \delta_0^{m-1} \mid \ldots \mid \delta_{n-1}^{m-1} \end{array}}$$

(with $p \geq 0, m > 0, n > 0$) is taken as shorthand for $m \times n$ rules: infer $\gamma^i \longrightarrow \delta_j^i$ from $c_0, \ldots, c_{p-1}, \gamma \longrightarrow \gamma_j$, for $i < m, j < n$.

The axioms and rules of inference are:

$$\frac{l = h(a), v = h(e)}{\langle a = e, h\rangle \longrightarrow h[l = v]} \tag{1}$$

$$\frac{\langle S,h\rangle \longrightarrow \langle S',h'\rangle \mid h'}{\begin{array}{rcl}
\langle \{S\,T\}, h\rangle & \longrightarrow & \langle \{S'\,T\}, h'\rangle \mid \langle T, h'\rangle \\
\langle \texttt{async}\ S, h\rangle & \longrightarrow & \langle \texttt{async}\ S', h'\rangle \mid h' \\
\langle \texttt{finish}\ S, h\rangle & \longrightarrow & \langle \texttt{finish}\ S', h'\rangle \mid h' \\
\langle \{\texttt{async}\ T\,S\}, h\rangle & \longrightarrow & \langle \{\texttt{async}\ T\,S'\}, h'\rangle \mid \langle \texttt{async}\ T, h'\rangle
\end{array}} \tag{2}$$

One can think of these rules as propagating an "active" tag from a statement to its constituent statements. The first rule says that if $\{S\,T\}$ is active then so is S (that is, any transition taken by S can be transformed into a transition of $\{S\,T\}$). The second rule says that if `async` S is active, then so is S. The third rule says the same thing for `finish` S. The fourth rule captures the essence of `async` (we call it the *"out of order"* rule). It says that in a sequential composition $\{\texttt{async}\ T\,S\}$, the second component S is also active. Thus one can think of `async` S as licensing the activation of the following statement (in addition to activating S).

The first `for` rule terminates execution of the `for` statement if its lower bound is greater than its upper bound.

$$\frac{l = h(e_0), u = h(e_1), l > u}{\langle \texttt{for}(x\ \texttt{in}\ e_0..e_1)\ S, h\rangle \longrightarrow h} \tag{3}$$

The recursive rule performs a "one step" unfolding of the `for` loop. Note that the binding of x to a value l is represented by applying the substitution $\theta = x \mapsto l$ to S, rather than by adding the binding to the heap. This is permissible because x does not represent a mutable location in S.

$$\frac{\begin{array}{c} l = h(e_0), u = h(e_1), l \leq u, m = l+1, T = S[l/x] \\ \langle T, h\rangle \longrightarrow \langle T', h'\rangle \mid h' \end{array}}{\begin{array}{l} \langle \texttt{for}(x\ \texttt{in}\ e_0..e_1)\ S, h\rangle \longrightarrow \\ \qquad \langle \{T'\ \texttt{for}(x\ \texttt{in}\ m..u)\ S\}, h'\rangle \mid \langle \texttt{for}(x\ \texttt{in}\ m..u)\ S, h'\rangle \end{array}} \tag{4}$$

We now define appropriate semantical notions.

DEFINITION 2.5 (Semantics). *Let $\stackrel{\star}{\longrightarrow}$ represent the reflexive, transitive closure of $\longrightarrow$. The operational semantics, $O[\![S]\!]$ of a statement S is the relation*

$$O[\![S]\!] \stackrel{def}{=} \{(h,h') \mid \langle S,h\rangle \stackrel{\star}{\longrightarrow} h'\}$$

Sometimes a set of observable *variables is defined by the programmer, and the notion of semantics appropriately refined:*

$$O[\![S,V]\!] \stackrel{def}{=} \{(h,h'\Big|_V) \mid \langle S,h\rangle \stackrel{\star}{\longrightarrow} h'\}$$

where for a function $f : D \to R$ and $V \subseteq D$ by $f\Big|_V$ we mean the function f restricted to the domain V.

Note in the above definition we have chosen not to restrict the set of variables over which the input heap is defined. In a more

complete treatment of the semantics, we would introduce the `new` operation which permits dynamic allocation of memory, and define the program as being executed in a heap that is initially defined over only the input array of strings containing the command line arguments.

DEFINITION 2.6 (Determinacy). *A statement S with set of observables V is said to be* scheduler determinate over V *(or just* determinate *for short) if $O[\![S,V]\!]$ represents the graph of a function, rather than a relation.*

S is said to be scheduler determinate *if it is scheduler determinate over the set of its free variables.*

2.2 Happens Before and May Happen in Parallel relations

We now establish two structural relations on statements, and connect them to the dynamic behavior of the statements.

DEFINITION 2.7 (Happens Before). *Given a statement S, two terminal path instances i, j in $\mathcal{P}[\![S]\!]$, we say that i* happens before *j, and write $i \prec j$, if for some arbitrary label sequences s,t,u, some sequence c over integers, and integers m_0, m_1 with $m_0 < m_1$:*

$$\begin{aligned} & i = s\,m_0\,c \wedge j = s\,m_1\,u \\ \textit{or,}\quad & i = s\,m_0\,c\,\mathtt{f}\,t \wedge j = s\,m_1\,u \end{aligned}$$

The definition does not explicitly mention `async` nodes. The label `a` may occur in i, but only in s or in t. In the first case the occurrence can be ignored because we are considering two paths that lie within the same `async`. In the latter case the occurrence may be ignored because it is covered by a `finish`. The intuition is that the "asyncness" of a node can never cause it to happen before some other node. But the "finishness" of a node can—it suppresses all downstream asyncs. This intuition is formalized in the next section.

The following proposition is easy to establish by reasoning about sequences.

PROPOSITION 2.2. *(Transitivity) If $i \prec j$ and $j \prec k$ then $i \prec k$. (Asymmetry) If $i \prec j$ then it is not the case that $j \prec i$. (Irreflexivity) For no i is it the case that $i \prec i$.*

Thus $\prec$ is a strict order. But it is not total. Consider $i = 0a$ and $j = 1$. It is not the case that $i \prec j$ or $j \prec i$.

DEFINITION 2.8 (May Happen In Parallel). *Given a statement S, two terminal path instances i, j in $\mathcal{P}[\![S]\!]$, we say that i* can start with *j* running*, if for some arbitrary label sequences s,t,u, some sequence c over integers, and integers m_0, m_1 with $m_0 < m_1$:*

$$\begin{aligned} & i = s\,m_0\,c\,\mathtt{a}\,t, \textit{ and,} \\ & j = s\,m_1\,u \end{aligned}$$

We say that i may happen in parallel with *j, and write $i\#j$, if j starts with i running, or i starts with j running.*

PROPOSITION 2.3. *Let S be a statement with two paths q, r. Then $q\#r$ iff $\neg(q \prec r) \wedge \neg(r \prec q) \wedge q \neq r$.*

The proof of the forward direction is easy. In the backward direction we need Proposition 2.1.

DEFINITION 2.9 (Race). *Given a statement S and two sub-statements T and U, we say that there is a race involving T and U, if for some legal instances $t = T\theta_t$ and $u = U\theta_u$, $t\#u$ and memory accesses by t and u conflict.*

2.3 Correspondence

We now establish the relationship between the HB and MHP relations and the transition relation. The formal language we are working with does not have conditionals, or local variables, or infinite loops. This means that every sub-statement will execute in every initial heap. Hence it is possible to characterize the MHP and HB relations in very simple terms.

The key idea in establishing the correspondence is to surface the path/time stamp of a statement in the transition relation. We label each step by the "reason" for the step—the path (from the root) to the substatement that triggers (is the base case for) the transition.

We proceed as follows. First we define *labeled statements*—each substatement is labeled with the path from the root. Next we label transitions. The rules are a straightforward adaptation of Rules 1—4. The only point worth noting is that in the recursive rule for `for`, the substitution $S[l/x]$ replaces x by l in the labels of all substatements of S as well.[3]

$$\frac{l = h(a), v = h(e)}{\langle a = e^s, h\rangle \longrightarrow^s h[l = v]} \tag{5}$$

$$\frac{\langle S,h\rangle \longrightarrow^s \langle S',h'\rangle \mid h'}{\begin{array}{rcl} \langle\{S\,T\},h\rangle & \longrightarrow^s & \langle\{S'\,T\},h'\rangle \mid \langle T,h'\rangle \\ \langle\{\mathtt{async}\ T\,S\},h\rangle & \longrightarrow^s & \langle\{\mathtt{async}\ T\,S'\},h'\rangle \mid \langle \mathtt{async}\ T,h'\rangle \\ \langle\mathtt{async}\ S,h\rangle & \longrightarrow^s & \langle\mathtt{async}\ S',h'\rangle \mid h' \\ \langle\mathtt{finish}\ S,h\rangle & \longrightarrow^s & \langle\mathtt{finish}\ S',h'\rangle \mid h' \end{array}} \tag{6}$$

$$\frac{l = h(e_0), u = h(e_1), u > l}{\langle \mathtt{for}(x\ \mathtt{in}\ e_0..e_1)\ S^s, h\rangle \longrightarrow^s h} \tag{7}$$

$$\frac{\begin{array}{l} l = h(e_0), u = h(e_1), l \leq u, m = l+1, T = S[l/x] \\ \langle T,h\rangle \longrightarrow^s \langle T',h'\rangle \mid h' \end{array}}{\begin{array}{l} \langle \mathtt{for}(x\ \mathtt{in}\ e_0..e_1)\ S, h\rangle \longrightarrow^s \\ \qquad \langle\{T'\ \mathtt{for}(x\ \mathtt{in}\ m..u)\ S\}, h'\rangle \mid \langle \mathtt{for}(x\ \mathtt{in}\ m..u)\ S, h'\rangle \end{array}} \tag{8}$$

Clearly this transition system is conservative over the previous one—it merely decorates each step with extra information.

THEOREM 1 (Characterization of HB). *Let S be a statement and q, r terminal paths in $\mathcal{P}[\![S]\!]$.*

If $q \prec r$ then for any heap h, in any labeled transition sequence starting from $\langle S,h\rangle$ containing q and r, (the transition labeled with) q occurs before (the transition labeled with) r.

(Converse) If for all heaps h and all transition sequences started from $\langle S,h\rangle$ containing q and r it is the case that q occurs before r then $q \prec r$.

The forward direction is proved by structural induction on S. The key case is sequential composition $U \equiv \{S\,T\}$ in which q is a path leading into S and r into T. Here, the only "out of order" transition possible is because of the "out of order" rule, which requires S be an `async`. But since $q \prec r$ we know that $q \equiv s0c$ or $q \equiv s0c\,f\,t$ and $r \equiv s1u$ (with s being the label for U). Hence S cannot be an `async` since its type is specified by the first label of $c\,f\,t$. In the converse direction, without loss of generality, let $q \equiv s0t$ and $r \equiv s1u$. If the first symbol in t that is not an integer is an `a`, then we show in the proof that the "out of order" rule can be used to construct an execution sequence in which r precedes q, contradicting our assumption. Hence $q \prec r$.

The proof of the following theorem is similar.

THEOREM 2 (Characterization of MHP). *Let S be a statement and q, r terminal paths in $\mathcal{P}[\![S]\!]$.*

If $q\#r$ then for any heap h there is a transition sequence starting from $\langle S,h\rangle$ containing q and r s.t. q occurs before r and another such that r occurs before q.

(Converse) If for all heaps h there is a transition sequence starting from $\langle S,h\rangle$ containing q and r s.t. q occurs before r and another such that r occurs before q, then $q\#r$ or $r\#q$.

[3] In Rule 7, s is the label for the whole `for` statement.

PROPOSITION 2.4. *Let S be a statement. If no two sub-statements are in a race, then S is determinate.*

The converse is not true. There may be a race but it may be *benign*, i.e., it does not affect the outcome of the program. Consider:

```
finish {
  async x=1; // S0=[f0a]
  x=1;       // S1=[f1]
}
```

Statements S0 and S1 are in a race (they may happen in parallel and their write sets overlap), however, the statement is determinate, it will always yield a heap which is the same as the initial except that maps `x` to `1`.

3. The "Happens-Before" Relation as an Incomplete Lexicographic Order

In this section, we formulate the "happens-before" relation, in a manner familiar from polyhedral analysis. In the polyhedral analysis of sequential languages, statement instances in a program are given unique time stamps represented as *integer vectors*. These vectors are ordered lexicographically—this order is sufficient to capture the idea of "happens-before" for a sequential language.

The strict lexicographic order is defined for two distinct such integer vectors u and v as follows:

$$u \ll v \quad \equiv \quad \bigvee_{p \geq 0} u \ll_p v, \tag{9}$$

$$u \ll_p v \quad \equiv \quad \left(\bigwedge_{k=1}^{p} u_k = v_k \right) \wedge (u_{p+1} < v_{p+1}) \tag{10}$$

As we saw in Section 2.1, the happens-before order in X10 must be sensitive to the presence of `finish` and `async` nodes. To take these constructions into account, we will use the paths of Section 2.1—vectors of integers, loop counters and the letters `a` and `f`—as time stamps. Polyhedral analyses can take loop counters symbolically, but reason about path instances, where loop counters take some integer value.

We will consider only terminal paths. The lexicographic order may be extended to paths simply by specifying how to order the additional symbols a and f, for instance by assuming that $\mathtt{a} < \mathtt{f}$ and that they occur later than integers and loop counters. This convention is irrelevant, since, by Proposition 2.1, we will never have to compare `a` or `f` to any other item in a path provided, we only compare *distinct* vectors.

Given a time stamp q, $|q|$ is its dimension, $q_i, 1 \leq i \leq |q|$ (sometimes written $q[i]$) is its i-th component, and $q[i..j], i \leq j$ is the vector whose components are $q_i, q_{i+1}, \ldots q_j$. A common shorthand for $q[i..|q|]$ is $q[i..]$.

A time stamp in which the loop counters have been replaced by integers denotes at most one instance of an elementary statement or *operation*. The admissible values are constrained to be within the enclosing loop bounds, which are assumed to be affine. The set of admissible values for the time stamps of statement S, the *iteration domain* of S, is written $\mathcal{D}_S$. Under the above hypothesis, $\mathcal{D}_S$ is a polyhedron.

We now reconstruct the "happens-before" relation as a "relaxed" lexicographic order. We start from the observation that:

$$\textbf{true} \equiv (q \ll r) \vee (q = r) \vee (r \ll q).$$

This suggests that $q \prec r$ be constructed as a case distinction:

- $q \ll r \rightarrow ?$
- $q = r \rightarrow$ **false**
- $r \ll q \rightarrow ?$

The case $q = r$ is obvious, since an operation cannot execute before itself. Let us now show that if $r \ll q$, then $q \prec r$ is impossible. In the notations of Definition 2.7, let s be the common prefix of q and r: $q = s.x.u$ and $r = s.y.v$. By Proposition 2.1, either $x < y$ or $y < x$ is true, and $r \ll q$ implies that $x > y$. Then, Definition 2.7 implies that $q \prec r$ cannot be true.

The conclusion is that in the above disjunction, only the first case has to be considered. This in turn can be expanded according to the definition (9) of $\ll$:

- $q \ll_0 r \rightarrow ?$
- $q \ll_1 r \rightarrow ?$
- $\ldots$
- $q \ll_n r \rightarrow ?$

The case distinction extends until $q \ll_k r$ is obviously false, i.e., when q_k and r_k are different integers, since all predicates $q \ll_{k'} r, k' \geq k$ contains the constraint $q_k = r_k$.

Let us now consider one of the cases $q \ll_p r$. The two time stamps have a common prefix $q[1..k] = r[1..k]$, and by the same reasoning as above, $q_{k+1} = 0$ and $r_{k+1} = 1$. We are then in a position to apply Definition 2.7. If the first letter in the vector $q[k+1..]$ is an `f` or if there is no letter, then $q \prec r$ is true, and otherwise is false.

The discussion above can be summarized by the following algorithm:

Algorithm H

- **Input**: Two paths q, r.
- **Output**: The constraint h in the loop counters of q, r (if any) which captures the precise conditions under which $q \prec r$.
- $h :=$ **false**
- $b :=$ **true**
- for $k = |q|$ downto 1:
  1. if $q_k = \mathtt{a}$ then $b :=$ **false**
  2. if $q_k = \mathtt{f}$ then $b :=$ **true**
  3. if $b \wedge k \leq |r|$ then $h := h \vee (q \ll_{k-1} r)$

Here, h denotes a disjunction of affine constraints, which is initialized to **false**, is augmented each time line (3) is executed, and is the "happens-before" predicate when the algorithm terminates.

In what follows, in the interest of compactness, we will allow sequences with more that two items, and timestamps containing integers larger than 1.

Example

Let us apply algorithm H to the example shown in Figure 1. The time stamps associated with each statement are as follows: `S0`: $[0, \mathtt{f}, 0, i, \mathtt{a}]$, `S1`: $[0, \mathtt{f}, 1, j]$, and `S2`: $[1]$.

We first ask if $S0 \prec S1$. Then $q = [0, \mathtt{f}, 0, i, \mathtt{a}]$ and $r = [0, \mathtt{f}, 1, j]$.

- We start from $k = |q| = 5$, $b =$ true, and $h =$ false.
- At $k = 5$, b becomes false, since $q_5 = \mathtt{a}$.
- Since q_k does not point to an `f` until $p = 1$, no changes occur. At $k = 1$, $q \ll_0 r \equiv q_1 < r_1$ is false, and hence $S0$ does not happen before $S1$.

Let us now ask the question if $S0 \prec S2$. Then $q = [0, \mathtt{f}, 0, i, \mathtt{a}]$ and $r = [1]$. Since $|r| = 1$, line (3) is never executed until $k = 1$. We reach $k = 1$ in the same state as in the previous example, but in this case $q \ll_0 r \equiv q_1 < r_1$ is true, and hence $S0 \prec S2$.

Although we have illustrated the algorithm with an example, the algorithm is not used in this fashion in the following sections.

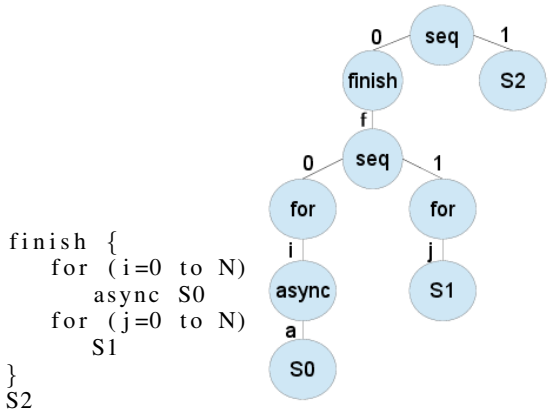

Figure 1: Example X10 code and its corresponding AST.

The important observation is that the algorithm only executes line (3) at a subset of the dimensions. Moreover, the subset is determined purely structurally, i.e., given the AST and two statements, one can find a subset I where lexicographic comparison should be performed. This leads to the following re-formulation of the algorithm as an *incomplete* lexicographic order:

$$q \prec r \equiv \bigcup_{k \in I} q \ll_k r, \tag{11}$$

It is well known that the $\ll_p$ are disjoint. A pair q, r being given, there is at most one k such that $q \ll_k r$. k is the rightmost index such that $q[1 \ldots k] = r[1 \ldots k]$. From this follows that $\prec$ is transitive.

The observation that the "happens-before" relation is an incomplete lexicographic order becomes important in the next section. Because of this property, we can formulate the dataflow analysis questions for X10 programs in a way that can be efficiently solved.

Lastly, the way we have constructed algorithm H clearly implies that:

PROPOSITION 3.1. *Algorithm H exactly implements the "happens-before" relation of Definition 2.7*

4. Dataflow Analysis

In this section, we present an adaptation of array dataflow analysis [11] for X10 programs, based on the "happens-before" relation as defined in the preceding section. The analysis is outlined in Figure 2.

Dataflow analysis aims at identifying, for each read access to a memory cell x, the *source* of the value found in x, i.e., the operation which wrote last into x. If the program is sequential, and fits into the polyhedral model [13] each read has a unique source which can be identified exactly [11].

However, the situation is different for parallel programs, since the actual execution order of operations may differ from run to run, due to scheduler decisions or hardware clock drifts. As a consequence, the content of some memory cell at a given step in the execution of a program may differ across runs. In other words, the answer to the dataflow question, which operation wrote last into x may be different.

This is called a race in software, or a hazard in hardware. The presence of a race condition in a program is usually a bug [4], which may be very difficult to diagnose and to correct, as it may manifest itself with very low probability.

[4] It ultimately depends on the intension of the programmer.

Input:
R, A, f_R: A read in a statement R. R reads from a shared array (or scalar) A with access function f_R.
$\mathcal{W}$: Set of statements W that write to A with access function f_W.
Output:
Q: Quast (Quasi-Affine Solution Tree [10]) that gives the producer of read by instances of R.
Algorithm:

1. **foreach** $W \in \mathcal{W}$
   (a) Compute Potential Sources $\Sigma_W(v)$ of W; the set of statement instances that (i) write to the same memory location, and (ii) the read do not happen before the write, parameterized by instances of R, v.
   (b) Compute Self Overwrites $\Sigma^-_W(v)$ of W; the set of writes overwritten by another instance of the same statement, parameterized by instances of R, v.
2. **foreach** $W \in \mathcal{W}$
   (a) Compute Validity Domain $Valid_W$ of $v \in R$; the set where v is valid for $\Sigma_W(v)$. An instance v becomes invalid if its write is overwritten by other statements.
3. Compute Q:
   $Q := \emptyset$
   foreach $W \in \mathcal{W}$
   (a) $Q := Q \cup \left((\Sigma_W(v) / \Sigma^-_W(v)) \wedge v \in Valid_W \right)$
   A write W is a producer of v if (i) it is a potential source, (ii) not overwritten by other instances of W, and (iii) not overwritten by writes of other statements.

Figure 2: Overview of array dataflow analysis for our X10 subset.

4.1 Potential Sources

Let us focus on an instance of statement R at time stamp v, which has a read of array A at subscript(s) $f_R(v)$. The potential sources are instances of a statements W at time stamp w, which write into A at subscript(s) $f_W(w)$. f_R and f_W are vector functions of the same dimension as A. The set of potential sources is defined by:

$$v \in D_R, \tag{12}$$
$$w \in D_W, \tag{13}$$
$$f_W(w) = f_R(v), \tag{14}$$
$$\neg(v \prec w) \wedge v \neq w \tag{15}$$

Constraint (12) and (13) respectively constraints v and w to set of legal time stamps (iteration domain) of R and W. Constraint (14) restricts to those with conflicting memory accesses; those access the same *element* of the array A. Lastly, constraint (15) removes writes that happen after reads ($v \prec w$), and write by the same statement instance ($v = w$). In a sequential program, $\prec$ is total, hence $\neg(v \prec w) \wedge v \neq w \equiv w \prec v$, which is the usual formulation [11].

Let $\Sigma_W(v)$ be the set of potential sources as defined by (12-14) and (15). If the source program fits in the polyhedral model, all constraints are affine with the exception of (15), which can be expanded in Disjunctive Normal Form (DNF). Hence, $\Sigma_W(v)$ is a union of polyhedra.

4.2 Overwriting

Let x be a write to the same memory cell as w. x overwrite w if in all executions, x happens between w and v, or $w \prec x \prec v$. It is clear that if an overwrite exists, v will never see the value written by w. Both

conditions are necessary: if one of them were not true, there would exists executions in which x happens before w, or v happens before x, and the value written at w would still be visible at v. This step is analogous to restricting the set of candidate sources to the most recent write in the original array dataflow analysis [11]. However, since the order is not total, the most recent write is not unique.

4.2.1 Self Overwrites

Write by a statement may be overwritten by other instances of the same statement. An instance w in $\Sigma_W(v)$ is a real source only if no other instance of W, x, overwrite w, i.e.,

$$w \in \Sigma_W(v) \wedge \neg\exists x \in \Sigma_W(v) : w \prec x. \tag{16}$$

This is exactly the definition of the set of upper bounds of Σ_W according to $\prec$. When $\prec$ is total, in the sequential case, (16) defines the unique maximum of P_W. In extreme cases, $\prec$ may be empty, and all tentative sources must be kept.

One possibility is to eliminate the existential quantifier in formula (16) using any projection algorithm, compute the negation and simplify the resulting formula. The drawback of this approach is its complexity: quantifier elimination in integers may generate expressions of exponential size, and so does negation.

Another possibility is to exploit the special form (11) of $\prec$. Since existential quantification distributes over disjunction, one has to compute $\exists x : w \ll_p x$ for each term which is present in $\prec$. Due to the very simple form of $\ll_p$, this set can be computed very efficiently using Parametric Integer Programming [10]. Simply solve the problem $\min\{x \in \Sigma_W(v) | w \ll_p x\}$ parametrically with respect to v and w. The result is a conditional expression (a *quast*) whose nodes bear affine constraints in the parameters, and whose leaves are either affine forms or the special term $\perp$, indicating that for some values of the parameters, the set above is empty. The disjunction of the paths leading to leaves not bearing a $\perp$ is the required projection. Then take the union of all such sets, denoted Σ_W^- in Figure 2, and subtract it from Σ_W.

4.2.2 Group Overwrites

Another case of overwrite happens when, for a given read, there are two possible writing statements, W_1 and W_2. A candidate source $w_1 \in \Sigma_{W_1}(v)$ is not visible to v if there exists $w_2 \in \Sigma_{W_2}(v)$ where $w_1 \prec w_2$.

This condition can be checked by inspecting the AST and Σ_W. In the AST, the paths from the root to W_1 and W_2 diverge at some `seq` or `async` or `finish` node. Assume that W_1 is to the left of W_2, then for each $w_1 \in \Sigma_{W_1}(v)$, one may associate w_2 such that $w_1 \prec w_2$. Let p be the common prefix of time stamps labeled to W_1 and W_2. Then $\forall k, 1 \leq k \leq p, w_1[k] = w_2[k]$ and $w_1[p+1] < w_2[p+1]$. Regardless of the remaining values beyond $p+1$, $w_1 \ll_p w_2$ holds. According to algorithm H, $w_1 \prec w_2$ if the uppermost parallel construct in W_1 below the common node is not an `async`, and hence w_1 is overwritten by w_2. Otherwise, w_1 and w_2 can happen in any order and w_1 is not overwritten. Similarly, there is no w_2 where $w_1 \prec w_2$ if W_2 is to the left of W_1, since we obtain $w_2 \ll_p w_1$.

However, this construction fails if for the considered value of v, $\Sigma_{W_2}(v)$ is empty. As a set, $\Sigma_W(v)$ is a function of v, and it may be empty for some values of v, i.e., for some values of v, R may have no sources from W. Let us define the *range* of W as:

$$\Omega_W = \{v | \Sigma_W(v) \neq \emptyset\}.$$

The source is in $\Sigma_2(v)$ if $v \in \Omega_2$, and in $\Sigma_1(v)$ if $v \in \Omega_1 \setminus \Omega_2$.

In the general case we must consider more than two writing statements. Let us define an order on the writing statements by the definition: $W < W'$ if W is to the left of W' in the AST, and if the uppermost parallel construct in the path to W from the common node is not an `async`. It is easy to prove transitivity of $<$. For each W, one may define a validity by the rule:

$$Valid_W = \Omega_W \setminus \bigcup_{w<W'} \Omega_{W'}.$$

The source is in Σ_W only if $v \in Valid_W$. Maximal elements in the $<$ order have no successors, hence for them $Valid_W = \Omega_W$, while some other statements may have empty validity sets, indicating complete overwriting.

5. Race Detection

Once we have Σ_W, Σ_W^-, and $Valid_W$ for all writers W, the output quast of the dataflow analysis may be computed. However, the resulting quast may not be well-defined: a read may have multiple sources when the program contains races. The detection of races using the array dataflow analysis result is discussed in this section.

5.1 Race between Read and Write

The set of potential sources $\Sigma_W(v)$ can be split in two sets according to whether $w \prec v$ is true or not. If $\neg(w \prec v)$, $\neg(v \prec w)$ and $v \neq w$ (recall constraint (15)), then the read and the write may happen in parallel. In other words, v and w are not ordered, and thus v may execute before w in some execution but w may precede v in the other. Hence there is clearly a race in this case.

Let $\Sigma'_W(v)$ be like $\Sigma_W(v)$ with (15) replaced by $\neg(w \prec v) \wedge \neg(v \prec w) \wedge v \neq w$. A non empty $\Sigma'_W(v)$ indicates a race. The emptiness of Σ'_W may be tested in many way, for instance by expanding its constraints to DNF and applying linear programming, or by submitting its definition to an SMT solver, like Yices, Z3, or CVC among others. If Σ'_W is found to be non empty, the compiler may issue a warning about statement R, and no further analyses are needed.

5.2 Race between Writes

Let $\Sigma^*_W(v)$ the set of sources after self overwrites have been removed. Then the source of a read at v is $\Sigma^*_W(v)$ if $v \in Valid_W$. However, the source may not be unique if the program has races. There are two types of races:

- Race between multiple writes by the same statement. If there exists a solution to the problem $v \in Valid_W, x, y \in \Sigma^*_W(v), x \neq y$, then x and y are involved in a race.
- Race between multiple writes by two statements. Two statements W_1 and W_2 have a race if: $Valid_{W_1} \wedge Valid_{W_2} \neq \emptyset$.

Both conditions can be checked by any SMT solver.

It is also important to note that it is not necessary to do all the above checks. For instance, if $W_1 < W_2$, their validity sets are disjoint by construction. Race detection of the first kind may be performed as we construct the sets, and the analysis may stop as soon as a race found. This approach may greatly reduce the complexity of the method.

5.3 Detection of Benign Races

The above approach is already sufficient to certify determinism of a program. However, additional analysis may be performed to flag questionable behavior of the program as warnings. For instance, our analysis detect array elements which are read but never written.

Another questionable behavior is *benign races*—races that do not influence the program determinacy. If two potential writes x and y may happen in parallel, x and y are in a race. However, if these writes are overwritten later or are not seen by any read, they are harmless. It might nevertheless be useful to warn the programmer of such behavior: a benign race can be taken as the indication of dead code. A way to handle them is therefore to do a backward recursive analysis starting from the output of the program.

5.4 Kernel Analysis

It is often the case that the full program is not polyhedral, while the core kernels are. In addition, due to the high cost of polyhedral analysis, it may not be practical to analyze the entire program.

The usual approach is to find "polyhedral parts"—subtrees in the AST that fit in the polyhedral model, and analyze them independently. Polyhedral methods are obvious candidates for such code fragments. For a finish/async language like X10, one must be more careful, since a subtree or a method may terminate but leave un-finished activities behind. Hence, to be handled with our methods, the sub-tree must satisfy the following properties in addition to the constraints of the polyhedral model:

- The uppermost parallel construct (in the path from the root of the sub-tree to each statement must be a `finish` if there is one.
- Similarly, let S be a statement that dominates statements in the sub-tree. Then the uppermost parallel construct in the path from the common prefix of S and the sub-tree to S must be a `finish` if there is one.

The above follows from algorithm H, and ensures that all statements before and after the sub-tree are ordered by the happens-before relation, with respect to the statements in the sub-tree.

6. Examples

In this section, we illustrate by examples the importance of two key strength of our approach; statement instance-wise, and array element-wise analysis. We specifically compare with the work by Vechev et al. [27] and with other polyhedral approaches [5, 7]. We are not aware of any other state-of-the-art static analysis techniques for race detection that perform instance-wise or element-wise analysis.

6.1 Importance of Element-wise

Let us first use an example similar to the one used by Vechev et al. [27]. The following code is a simplified example of a common case in parallel programming, where a shared array is accesses by multiple threads.

```
finish {
  async for (i in 1..N)
      B[i] = C[i];          // S0
  async for (j in N..2*N)
      B[j] = C[2*i];        // S1
}
for (k=1:2N)
     ... = foo(B[k], ...); // S2
```

The time stamps and iteration domains are:

- $S0$: $[0,\mathtt{f},0,\mathtt{a},i]$, $\mathcal{D}_{S0} = \{i | 1 \leq i \leq N\}$
- $S1$: $[0,\mathtt{f},1,\mathtt{a},j]$, $\mathcal{D}_{S1} = \{j | N \leq j \leq 2N\}$
- $S2$: $[1,k]$, $\mathcal{D}_{S2} = \{k | 1 \leq k \leq 2N\}$

The only read in the program is the read of `B` by $S2$. Our analysis returns the following answer to the question: which statement produced the value of `B[k]` at $S2$:

- If $1 \leq k \leq N \wedge k \leq 2N$ then $S0[k]$ is a producer.
- If $1 \leq k \leq 2N \wedge N \leq k$ then $S1[k]$ is a producer.

where $Sn[v]$ denote the instance of Sn when its loop counters take the value v. It concludes that there is a race by two writers since the two sources overlap at $k = N$.

For this example, both of the other approaches will find the race with similar precision. However, if an analysis is not element-wise, then the analysis only finds that there is a race with the entire array `B`. Assuming that the programmer is warned of this race and change the lower bound of the j loop to $N+1$, making the program race free, statement based approaches will still conservatively flag the array `B` to be in conflict.

6.2 Element-wise with Polyhedral

However, element-wise analysis in the work by Vechev et al. [27] is limited compared to polyhedral approaches, since they use an over-approximation. They require that any multi-dimensional arrays is reshaped into a 1D array, and the range of the 1D array to be represented with affine constraints. Furthermore, the renaming must be relative to what is called the `taskID` that identify an iteration of the loop ran by a thread.

For example, write to array `A` in the following code is expressed as writes to $\mathtt{A_i[j]}$, where i is the `taskID`.

```
for (i in 0..(N-1))
  async
     for (j in 0..(N-1))
        A[i][j] = ... // S0
```

Approaches based on the polyhedral model, including ours, represents the write to `A` as an affine function $(i, j \rightarrow i, j)$ from the iteration domain $\mathcal{D}_{S0} = \{i, j | 0 \leq i, j < N\}$.

Let us illustrate the difference with a slight modification to the first example.

```
{ finish {
    async for (i in 0..(N-1))
        B[2*i] = C[i];       // S0
    async for (j in 0..(N-1))
        B[2*j+1] = C[2*i];   // S1
  }
  ... = foo(B[N]); // S2
}
```

The difference is in the writes to `B`, which now do not conflict. Our analysis returns the following answer to the question, which statement produced the value read by read `B[N]` at $S2$:

- If $\exists e : 2e = N \wedge N \geq 2$ then $S0[N/2]$ is a producer.
- If $\exists e : 2e = N - 1 \wedge N \geq 1$ then $S1[(N-1)/2]$ is a producer.

Note that the parametric integer linear programming [10] step (Section 4.2.1) introduces a "new parameter" (existentially quantified variable). The intersection of the two validity sets is empty, and we conclude that the program is race free.

However, the over-approximation by Vechev et al. [27] will approximate the write by $S0$ to be $0 \leq i \leq 2N-2$, and the write by $S1$ to be $1 \leq i \leq 2N-1$. Clearly, the two approximations overlap, and hence their approach would conservatively flag the program to have race.

6.3 Importance of Instance-wise

The examples above can also be implemented using *doall* loops. When implemented as parallel loops, previous approaches [5, 7] based on the polyhedral model can verify its determinacy, and does not require extensions proposed in this paper.

Our work can also detect races in finish/async programs that cannot be expressed with *doall* parallelism. The following is a simplified example of a case when such parallelism may be used. The example is based on Gauss-Seidel stencil computation that performs updates in-place, and uses some of the values (`A[i-1][j]` and `A[i][j-1]`) computed at the current time step and others from the previous time step.

The following code fragment illustrates a possible use of `async` in a way that cannot be expressed as loop parallelism. Detail of the statement `S1` is not given to simplify the presentation, but some code corresponding to an asynchronous send is the motivation behind this example.

```
for (t in 1..T)
   finish for (i in 1..N-2) {
      //boundary conditions omitted
      for (j in start..end)
        A[i][j] =
         update(A[i-1][j], A[i][j-1],
          A[i][j], A[i+1][j], A[i][j+1]); // S0
      async S1(A[i][end]);                // S1
      //boundary conditions omitted
   }
```

The point we illustrate with this example is the importance of statement instance-wise analysis. At the granularity of (static) statements, the pair of statements $S0, S1$ may happen in parallel. This is a conservative approximation because $S0[t,i]$ may happen in parallel with $S1[t',i']$ when $t \geq t'$ and $i > i'$. With this precision, our approach find that the read of `A[i][end]` by $S1$ is always the value written by $S0$.

6.4 Benefits of Array Dataflow Analysis

Array dataflow analysis is, strictly speaking, an overkill for detecting races. The formulation used by Vechev et al. [27] focuses on finding conflicting memory accesses. Dataflow analysis goes one step further by eliminating some of the accesses that are guaranteed to be overwritten from the consumer's perspective. Consider the following example:

```
finish{
  async{
    x = f(); //S1
    x = g(); //S2
  }
  async{
    x = h(); //S3
    x = k(); //S4
  }
}
t = x;       //S5
```

The approach by Vechev et al. [27] will find that statements $S1$, $S2$, $S3$, and $S4$ are all in race since they all may happen in parallel and writes to x. In contrast, array dataflow analysis will show that the read of x at $S5$ has two potential sources, $S2$ and $S4$.

The output by Vechev et al. [27] grows in size as the number of statements in `async` increases, while the output of dataflow analysis does not. When our analysis is integrated to a programming environment, we believe that the preciseness and compactness of our analysis result will help the programmer more than simply detecting races.

Moreover, once the statement $S5$ is removed from the above example, the approach by Vechev et al. [27] would still detect a race, while our analysis would detect that the race is benign, and hence the full `finish` block is dead code.

7. Implementation and Evaluation

We have implemented[5] our analysis for the subset of X10 described in Section 2. We take a representation of the AST, where statements only specify arrays (or scalars) being read or written, disregarding the what the operation is. Once we detect polyhedral regions in X10 programs, equivalent information can easily be extracted from the internal representation of the compiler.

Analysis of loop programs to detect regions amenable for polyhedral analysis, frequently referred to as Static Control Parts (SCoPs), or Affine Control Loops (ACLs) is well established through efforts to integrate polyhedral parallelizers into full compilers [14, 19]. In addition, we require that array accesses `a[i]` and `a[j]` point to the same memory location iff $i = j$. In general, such guarantee require pointer analysis, which is outside the scope of this paper.

We use the Integer Set Library [28] in our implementation to perform polyhedral operations and to solve parametric integer linear programming problems. The analysis itself is written in Java, and Java Native Interface is used to call ISL.

Java Grande Forum Benchmark Suite

Although our key contribution is verification of finish/async programs, we are not aware of any set of parallel benchmarks that use the extra expressive power of finish/async in their polyhedral parts. We have demonstrated how our technique can handle such programs earlier with examples. In this section, we use Java Grande Forum benchmark suite [26] also used by Vechev et al. [27] to compare performance and applicability of our proposed analysis to their approach. The results are summarized in Table 1.

Out of the 8 benchmarks, 3 that were not handled by Vechev et al. [27] cannot be handled by ours either. SPARSE includes indirect array accesses, which falls out of the polyhedral model. Similarly, MONTECARLO and RAYTRACER cannot be handle by polyhedral analysis. All of the remaining 5 fit the polyhedral model, at least partially, and we were able to verify the determinacy of all parallel blocks. In fact, MolDyn has data dependent conditionals, which we approximate by assuming that both branches are always taken. Clearly, this gives a superset of the set of races, which the analyzer proved to be empty.

Although we present execution times of our method, and that of Vechev et al., we do not claim that our approach is more efficient. Their implementation is not directly comparable to ours, due to many reasons. For instance, they work on a lower level representation of the program (`Jimple`,) which create a large number of scalar variables, and necessitates loop and array re-construction. In Section 6, we have demonstrated that our method can detect races in more program instances.

8. Related Work

Our work may be placed in two different contexts, (i) as an extension to the polyhedral model for analyzing finish/async programs, and (ii) as an approach for statically verifying determinism of finish/async programs.

Array dataflow analysis was introduced by Feautrier [11] and further expanded by Pugh and Wonnacott [20]. Extensions beyond the polyhedral model were proposed by Pugh and Wonnacott [21] and by Barthou et al. [3]. As far as we know, time stamps were first introduced by Feautrier [12] as a trick for proving the existence of schedules for well-structured sequential programs. They were further exploited for specifying complex program transformations by Bastoul [4]. They are similar to the *pedigrees* proposed by Leiserson et al. [17], with the difference that pedigrees are computed at run time, while time stamps exist only at compile time.

Since the emphasis in the polyhedral literature is placed on automatic parallelization, there has been very little work on verifying already parallel programs. The work by Collard and Griebl [7] that presents array dataflow analysis for programs with *doall* parallelism is most closely related to our work. The key distinction is that we handle finish/async programs that can express parallelism not expressible by doall loops.

In the other context, one key question in reasoning about determinism is the question which statements (or statement instances) have a clearly defined order of execution. Analyses to answer this question for finish/async programs, closely related to our "happens-before" relation, have been presented by Agarwal et al. [1] and by Lee and Palsberg [16].

[5] Our implementation is available at `www.cs.colostate.edu/PolyhedralX10/`

Benchmark	`while` loop[1]	data-dep. `if`[2]	Time (s)[4]	Reference[5] Time (s) [27]
CRYPT	Y		7.6	54.8
CRYPT1	Y		0.24	-
CRYPT2	Y		0.24	-
SOR			1.85	-
SOR1			0.29	0.41
LUFACT1			0.35	1.94
SERIES	Y		1.25	-
SERIES1	Y		0.06	55.8
MOLDYN1[3]			0.35	24.6
MOLDYN2		Y	0.92	2.5
MOLDYN3			0.14	0.32
MOLDYN4			0.08	1.01
MOLDYN5		Y	0.08	0.34

Table 1: Performance of our implementation on JGF benchmarks [26]. Entries with the name followed by a number are verification of a parallel block that each contain a parallel loop surrounded by finish. All programs/blocks were verified to be determinate.

[1] Indicates that while loops were converted to for loops. These while loops are of the form:
`n=100; do { ... n--;} while (n>=0);`

[2] Indicates that data-dependent if statements were over-approximated by assuming both branches were always taken.

[3] MOLDYN require a final variable *pad* to be constant propagated due to expressions like: $i2 * pad$.

[4] Our experiments were conducted with 4-core Intel Core2Quad (2.83GHz) and 8GB of memory. We used Java 1.6, and ISL 0.10.

[5] These timing results are taken from their article [27] and were conducted with 4-core Xeon (3.8GHz) and 5GB of memory.

While Lee and Palsberg work at the level of a statement, Agarwal et. al. try to increase precision by exhibiting conditions on loop counters that guarantee (or forbid) parallel execution. However, these conditions use only equality and inequality, instead of the full power of affine constraints. The algorithms in these two papers are surprisingly complex when compared to algorithm H.

There is a separate body of work that address race detection with multi-threaded programs with locks (e.g., [9, 15]). These methods are not directly applicable to modern parallel languages where locks are rarely used.

The work by Vechev et al. [27] goes beyond the evaluation of the "may happens in parallel" relation and attempt to verify determinism of finish/async programs. Their analysis is also instance-wise and element-wise. The main difference is that their work use over-approximations of memory accesses, where our analysis is exact. In addition, we use array dataflow analysis to find races, but the information given by the analysis, which is more than enough to find races, can be used for other purposes.

Dynamic race detection (e.g., [25]) is a complementary technique to static analyses, and is more broadly applicable. However, dynamic analysis requires significant run-time overhead, and is subject to the well-known Dijkstra saying, that they can be used to prove the existence of races, not their absence.

The main drawback of our methods, compared to other approaches, is their restricted applicability; we require loops to be affine. Affine loop programs can be frequently found in scientific applications, which is an important target for emerging parallel programming languages. We believe that the increased precision more than compensates this restriction.

9. Conclusion and Future Work

This paper is a first step towards applying polyhedral analysis to finish/async programs. It has been written in the context of the X10 language. However, we expect our approach to be applicable to other languages with similar parallel constructs. For programs that fit in the polyhedral model, the analysis is exact, and as precise as can be. There are neither false positives nor false negatives. As a side effect, one can exploit the results of dataflow analysis for many other tasks, like scheduling and locality improvement, undefined variables detection, constant propagation and semantic program verification.

The approach in this paper can be extended in two directions:

- The X10 language has several control constructs which may create (or remove) races. Among them are clocks, a generalization of the classical barriers, the `atomic` modifier, and the `at` statement, which delegates a calculation to a remote place of the target system. Basically, all these constructs necessitate a new definition of the "happens-before" relation. The question is whether algorithm H can be extended to take care of them.

  Handling `atomic` and `at` constructs is a minor extension to the results presented here, but space constraints preclude an elaborate explanation. We are currently working on extending our analysis to handle clocks.

- Like all polyhedral analyses, our method applies only to a limited class of programs. Is there a possibility to remove some of these restrictions? A classical approach is to deal only with polyhedral subtrees of the AST, provided they don't interfere with the remnants of the program.

  One may also resort to approximations. The difficulty here is that since the source computation uses set *differences* (see for instance Section 4.2.2) over- and under-approximations are both needed. Depending on the quality of the approximations, the resulting analysis may have both false negatives and false positives. The problem will be to minimize their number.

References

[1] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 183–193, New York, NY, USA, 2007. ACM.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.W. Maessen, S. Ryu, G.L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, et al. The Fortress Language Specification. *Sun Microsystems*, 139:140, 2005.

[3] Denis Barthou, Jean-François Collard, and Paul Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[4] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, september 2004.

[5] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien, P. Quinton, and D. Wonnacott. ompverify: polyhedral analysis for the OpenMP programmer. *OpenMP in the Petascale Era*, pages 37–53, 2011.

[6] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[7] Jean-François Collard and Martin Griebl. Array dataflow analysis for explicitly parallel programs. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 406–413. Springer Berlin / Heidelberg, 1996.

[8] UPC Consortium et al. UPC language specifications. *Lawrence Berkeley National Lab Tech Report LBNL–59208*, 2005.

[9] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review*, 37(5):237–252, 2003.

[10] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[11] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[12] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[13] Paul Feautrier and Christian Lengauer. The polyhedral model. In David Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[14] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.N. Pouchet. Polly–Polyhedral optimization in LLVM. In *IMPACT 2011 First International Workshop on Polyhedral Compilation Techniques*, 2011.

[15] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 13–22. ACM, 2009.

[16] Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 25–36, New York, NY, USA, 2010. ACM.

[17] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *PPOPP'12*, pages 193–204, 2012.

[18] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[19] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.A. Silber, and N. Vasilache. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit*, page 2006, 2006.

[20] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *ACM SIGPLAN PLDI*, pages 140–151, 1992.

[21] W. Pugh and D. Wonnacott. Going beyond Integer Programming with the Omega Test to Eliminate False Data Dependencies. Technical Report CS-TR-3191, U. of Maryland, December 1992.

[22] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification version 2.2, March 2012. `x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[23] Vijay Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *CONCUR 2005 - Concurrency Theory*, pages 353–367, London, UK, 2005. Springer-Verlag.

[24] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 161–172, New York, NY, USA, 2007. ACM.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[26] L.A. Smith, J.M. Bull, and J. Obdrizalek. A parallel Java Grande benchmark suite. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 6–6. IEEE, 2001.

[27] Martin Vechev, Eran Yahav, Raghavan Raman, and Vivek Sarkar. Automatic verification of determinism for structured parallel programs. In *Proceedings of the 17th international conference on Static analysis*, SAS'10, pages 455–471, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] S. Verdoolaege. isl: An integer set library for the polyhedral model. *Mathematical Software–ICMS 2010*, pages 299–302, 2010.

[29] Peng Wu, Paul Feautrier, David Padua, and Zehra Sura. Instance-wise points-to analysis for loop-based dependence testing. In *International Conference on Supercomputing (ICS 2002)*, pages 262 – 273, June 2002.

[30] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, et al. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience*, 10(11-13):825–836, 1998.