Static Analysis of OpenStream Programs*

Albert Cohen
Inria, Parkas team
Ecole normale supérieure, Paris
Albert.Cohen@inria.fr

Alain Darte Paul Feautrier
Compsys team, LIP, UMR 5668 CNRS, Inria,
ENS-Lyon, UCB-Lyon
Firstname.Lastname@ens-lyon.fr

ABSTRACT

This paper studies the applicability of polyhedral techniques to the parallel language OpenStream [25]. When applicable, polyhedral techniques are invaluable for compile-time debugging and for generating efficient code well suited to a target architecture. OpenStream is a two-level language in which a control program directs the initialization of parallel task instances that communicate through *streams*, with possibly multiple writers and readers. It has a fairly complex semantics in its most general setting, but we restrict ourselves to the case where the control program is sequential, which is representative of the majority of the OpenStream applications. This restriction offers deterministic concurrency by construction, but deadlocks are still possible.

We show that, if the control program is polyhedral, one may statically compute, for each task instance, the read and write indices to each of its streams, and thus reason statically about the dependences among task instances (the only scheduling constraints in this polyhedral subset). These indices may be polynomials of arbitrary degree, thus requiring to extend to polynomials the standard polyhedral techniques for dependence analysis, scheduling, and deadlock detection. Modern SMT allow to solve polynomial problems, albeit with no guarantee of success; the approach of Feautrier [10] may offer an alternative solution. We also establish two important results related to deadlocks in OpenStream: 1) a characterization of deadlocks in terms of dependence paths, which implies that streams can be safely bounded as soon as a schedule exists with such sizes, 2) the proof that deadlock detection is undecidable, even for polyhedral OpenStream.

1. INTRODUCTION

For the every-day programmer, the performance increase of processors has been felt, for a long time, with no need to change programming paradigms. In the last years however, the development of more-difficult-to-program accelerators (FPGA, GPU, multicores), and even larger-scale platforms, has offered an impressive computational power to a larger public while exposing the users to the difficulty of parallel programming. The pressure to still achieve portability, performance, and productivity has become much stronger on compilers and programming languages. The question still remains to find the right trade-off between relying on optimized libraries, on static analysis and optimizations, or on runtime systems with more dynamic decisions, and how to make these three views collaborate.

Parallel programming is notoriously difficult. The reasons are multiple: in contrast to sequential programming, there is no unique model of parallel programming and computer, and it is difficult to visualize a process in which many events occur independently. Most importantly, parallel programs—like all programs—have bugs, which are difficult to track and correct. A concurrency bug may not be reproducible, or have a very low probability of occurrence. Hence the importance of creating parallel programs that are correct by construction, or whose correctness can be checked statically.

The last years have seen the emergence of many parallel programming approaches: low-level (MPI, OpenCL, CUDA), runtime-based (Kaapi, StarPU, TBB), vector and array languages (APL, HPF, ZPL, SaC), PGAS languages (Co-Array Fortran, Chapel, UPC, X10). The class of streaming dataflow languages (SDF, StreamIt, OpenStream, SigmaC), based on Kahn process networks (KPN), has the desirable property that *determinism* is enforced at the language level; it is also popular for the design of reactive systems. However, all types of bugs can only be excluded at the price of severely restricting the expressive power. For instance, there are deterministic languages with (deterministic) deadlocks (e.g., StreamIt) or non-deterministic deadlock-free languages (e.g., Cilk, StarSs, OpenMP, X10), and fragments of these are deterministic when shared variables are synchronized with task joins and/or dependences.

Similarly, due to familiar undecidability theorems, static checking can be obtained only at the price of severely restricting the expressive power of the base language. The polyhedral model is such a system of restrictions: in its classical setting, control statements are restricted to counted loops with affine bounds, program statements are restricted to simple calculations on scalars and array elements, and array subscripts are restricted to affine functions of the loop counters. Most programs do not fit these constraints, but it is often possible to isolate polyhedral fragments or to define a "polyhedral subset" of a language, either to develop analyses for this well-defined subset (when feasible) or to prove the

*This work was partly supported by the ManycoreLabs project PIA-6394 led by the manycore company Kalray.

IMPACT 2016
Sixth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2016, Prague, Czech Republic
In conjunction with HiPEAC 2016.

http://impact.gforge.inria.fr/impact2016

difficulty of an analysis or optimization of this subset, and thus of the general language. This has been done for the X10 language in [32] (for a subset where race detection is solvable by polyhedral techniques) and [33] (for a subset where it is undecidable). The aim of this paper is to similarly define a polyhedral subset of the OpenStream language [22, 25] and to use it to explore different questions such as:

- What is the flow of the computations in the program? Can it be described as a closed form relation, rather than a more general inductive one?
- Can we bound statically and safely the size of streams?
- Is it possible to change the granularity of the computations w.r.t. communications in streams?

The first question amounts to defining a form of dependence analysis for polyhedral OpenStream and is discussed in Section 2, after a definition of the language. The second and third questions are linked to the problem of deadlock detection, which is discussed in Section 3. They are motivated by the fact that OpenStream is a language with tasks whose schedule is fully guided by its runtime while grouping tasks and coalescing communications statically can only be done safely if it does not introduce deadlocks. Finally, Section 4 discusses related work, to position OpenStream in the landscape of streaming languages and to recall results obtained for the analysis of polyhedral X10. Section 5 summarizes our main contributions and some research directions.

2. THE OPENSTREAM LANGUAGE

The design of OpenStream builds on a previous streaming extension [22] to OpenMP. Source code, support tools, benchmarks, and bibliography can be found on `http://www.openstream.info`. For a more detailed presentation, one may refer to [25], and to the formal model underlying the operational semantics of OpenStream [23].

2.1 The Base Language

In a nutshell, OpenStream allows the composition of tasks communicating through dataflow streams, as well as separate compilation. It also provides more general dynamic constructs to support complex data structures and unbounded fan-in/fan-out communications. It has been shown that it is sufficiently expressive to efficiently encode high-level parallel language features such as the memory regions of StarSs [21], as well as low-level point-to-point communication primitives such as futures [24, 25]. OpenStream also provides syntactic support for broadcast operations.

2.1.1 Concurrency

OpenStream relies on programmer annotations to specify regions of the control flow that may be spawned as concurrent coroutines and delivered to a runtime execution environment. These regions are called *tasks* and inherit the OpenMP task syntax and, without stream annotations, the same semantics. OpenStream is a two-level language: a *control program* directs the *creation* of tasks, then each created task waits until its *activation*, which means that all tasks it *depends upon* (see hereafter) have terminated execution and it can now start its *execution* as soon as it is selected by the runtime scheduler. There are no constraints in the amount of work done by the control program or the tasks. At the time of creation, tasks have access to all variables of the control program in the current scope, using standard OpenMP mechanisms like `firstprivate` and `copyin`. Communication from tasks to the control program is through shared variables, under control of barrier synchronization, thanks to constructs inherited from OpenMP. For the polyhedral subset we consider, there is no communication from the tasks to the control program.

Despite its expressiveness, the OpenStream programming model comes with specific conditions under which the functional determinism of Kahn process networks [15] is guaranteed by construction. These conditions enforce a precise interleaving of data in streams derived from the control flow of the control program. One simple sufficient condition for determinism is that the control program is sequential, as in our polyhedral subset. More general conditions exist [23], which are not considered here.

2.1.2 Synchronization

OpenStream allows to express the flow of data between tasks through the concept of *streams*, inducing producer-consumer *dependences*. A stream is a virtual one-dimensional array of indefinite size, which can only be accessed through a sliding window. A window is defined by two nonnegative integers, the *horizon* (the size of the window) and the *burst* (the amount by which the window is shifted at each task creation). These numbers may be arbitrary data-dependent expressions. Our polyhedral fragment restricts them to numerical or symbolic constants, or polynomial expressions for static analysis purposes (see Section 2.2.2 for details). General OpenStream programs allow dynamic connections between tasks, multiple tasks interleaving their communications in the same streams, arbitrary and variable fan-in, fan-out, and communication rates in a dynamically constructed task graph. Also, unlike OpenMP, streams are first class objects of the language allowing for arbitrary task graph topologies. The definition of streams themselves is done thanks to two additional clauses for the `task` construct: the `input` and `output` clauses. The syntax uses the C++ style for stream operators, `<<` and `>>`. An array declaration (in plain C) defines the sliding window accessible within the task, as well as its size (the horizon). The connection of a sliding window to a stream in an `input` or `output` specifies the burst. The abbreviated form with no specified burst means a burst equal to 1. For an output clause, the burst and horizon must be equal. Task activation is enabled by the availability, on each input stream, of all horizon elements on the input window (see Section 2.2.1 for dependence analysis between tasks).

The example in Figure 1 (in pseudo-code to make it shorter) illustrates the use of the `input` and `output` clauses. The semantics of stream operations is determined by the *control program*, at the time of *task creation*. T_0 is a producer task, pushing two elements to stream `s` after execution (but the

```
  stream int s;
  int a[2], b[3], c[2];

T0 #pragma omp task output (s << a[2]);
  a[0] = 42, a[1] = 43; // write two cells in s

  for (i = 0; i < N; i++) {
    if (i % 2) {
T1 #pragma omp task output (s << b[3]), input (s)
      b[0] = b[1] = b[2] = foo(s); // read one, write three
    }
T2 #pragma omp task output (s), input (s >> c[2])
    s = bar(c[0], c[1]); // read two, write one
  }
```

Figure 1: Example of `input`/`output` clauses.

two slots are reserved at task creation). T_1 and T_2 are both producers and consumers of stream `s` and T_1 is created only every *even* iteration of the enclosing loop. Here, the values of the bursts and horizons imply that a single (sequential) execution is possible. The first created task instance in the loop is $T_2(0)$, which pops the two elements from `s` and writes a new one. This one is then read by $T_1(1)$, which adds three new elements. Two of them are read by $T_2(1)$, a new one is created, then $T_2(2)$ pops the two, and writes one, etc. In general, tasks can also be guarded by more complex control flow and, as shown here, tasks can have interleaved accesses through the same stream. Also, when burst < horizon, dependences can be even less intuitive.

2.2 Dependence Analysis for OpenStream

In this paper, we focus on programs for which the execution order of the control program is easily deduced from the abstract syntax tree (AST). (This is not the case for programs with `goto` or even non-affine `if` constructs, for example.) As a side effect, one obtains the creation order of task instances (Section 2.2.1). For each stream and each task instance, one can then compute a read or write index (Section 2.2.2). One can finally compute dependences between tasks (Section 2.2.3). In short, since streams are accessed in single assignment mode by construction, there are only producer-consumer (PC, or flow, or RAW) dependences. Two task instances are in dependence if, for some stream, the output window of one of them intersects the input window of the other. In this case, the writer must be executed first. The execution order of task instances is the transitive closure of the dependence relation.

2.2.1 Creation Order of Tasks

We restrict the study to a subset of OpenStream, where the control program fits the polyhedral model [11]. We consider unions and projections of polyhedra defined as Presburger formulas; *affine* is implicitly lifted to piecewise, quasi-affine expressions (integer division with a numerical constant). The code of the tasks themselves can be arbitrary, but without nested task creation (no tasks can be created within tasks). Control statements are restricted to counted loops with affine bounds, operations are restricted to simple calculations on scalars and array elements, and array subscripts are restricted to affine functions of the loop counters. Each task instance can then be identified by its *position vector*, following the AST labels, encoding:

- **Sequence**: S_1`;` `...;` S_n, with n outgoing edges, labeled from 1 to n.
- **Loop**: `for(i = 0; i < n; i++)`, with one outgoing edge, labeled by i.
- **Task**: with one outgoing edge labeled by the letter a.
- **Conditional**: two outgoing edges, labeled *tt* and *ff*.
- **Basic statement**: with no outgoing edge.

The position vector of a node is the list of labels encountered on the unique path from the AST root to the node. In the example of Figure 1, the position vectors of T_1 and T_2 are $[1, i, 0, tt, a]$ and $[1, i, 1, a]$, respectively. The *creation order* of tasks is then simply given by the lexicographic order of position vectors (excluding the terminal "a", not necessary here). For example, consider an instance of T_1, $[1, i, 0, tt, a]$, and an instance of T_2, $[1, i', 1, a]$. The first is created before the second if and only if $1 < 1$ or $(1 = 1 \wedge i < i')$ or $(1 = 1 \wedge i = i' \wedge 0 < 1)$, i.e., $i \leq i'$ (in the case both exist, which may not be the case due to conditionals). This creation order is denoted $\preceq$ (and $\prec$ if strict).

Note that, in this context, as often, conditionals pose a difficulty. The execution order of exclusive conditional branches is undefined. However, when the conditional expression is affine, one may associate an iteration domain to each statement in any of the branches, and state that two position vectors can be compared only if their iteration domains intersect. In the following, we restrict ourselves to such affine conditional expressions. In the example of Figure 1, T_1 is guarded by the quasi-affine expression $i \bmod 2 = 1$.

2.2.2 Stream Indices

Let W_s (resp. R_s) be the set of tasks with write (resp. read) access to a stream s. Each task instance t writes (resp. reads) s through a *window*, with an associated *burst* $b_{t,s}$. The position of the window (its index) is computed by the control program by summing the *bursts* of all preceding task instances that write (resp. read) the stream. To show the strong link with computations of cardinals (and generalizations), we first consider the case where a burst is a numerical or symbolic constant that can be extracted from the program text, in which case we can write $b_{\tau,s}$ instead of $b_{t,s}$ for any instance t of a task τ. A burst is a nonnegative integer, and can only be null for an input stream (the `peek` operation).

Let $I_s(t)$ (resp. $J_s(t)$) be the first index of output (resp. input) stream s written (resp. read) at task instance t. Let D_τ be the domain (set of instances) of task τ. We have the fundamental formulas (see also Definition 8 in [23]):

$$I_s(t) = \sum_{\tau \in W_s} b_{\tau,s} \mathrm{Card}\ \{x \in D_\tau | x \prec t\} \quad (1)$$

$$J_s(t) = \sum_{\tau \in R_s} b_{\tau,s} \mathrm{Card}\ \{x \in D_\tau | x \prec t\} \quad (2)$$

Since for polyhedral programs D_τ is a polyhedron, and since $\prec$ is a disjunction of (quasi-)affine constraints, the cardinal can be computed as a closed form by familiar techniques (Ehrhart polynomials [3] or Barvinok generating functions [31]) and their corresponding libraries (Polylib or barvinok). The result will usually be a polynomial, the degree of which is equal to the dimension of D_τ. However, these polynomials are not arbitrary, and their properties may be used to advantage for program analysis. For instance, the innermost loop counter in t will usually occur linearly in $I_s(t)$. The index function I_s is also, of course, related to the relation $\prec$, the task creation order, as follows:

PROPOSITION 1. *If $t \prec t'$ have write access to the stream s, then $I_s(t) + b_{\tau,s} \leq I_s(t')$ and, in particular, $I_s(t) < I_s(t')$.*

PROOF. Observe that the sets whose cardinals contribute to $I_s(t')$ are super-sets of those contributing to $I_s(t)$, and that t belongs to the first one but not to the other. Also, for write accesses, bursts are positive integers. □

Note that if bursts are not constants, Proposition 1 remains true with $b_{t,s}$ instead of $b_{\tau,s}$. From this follows directly that streams have the single assignment property, since the write windows for t and t', i.e., $[I_s(t), I_s(t) + b_{t,s} - 1]$ and $[I_s(t'), I_s(t') + b_{t',s} - 1]$, are always disjoint. Also, if bursts are not constants, Formulas (1) and (2) become

$$I_s(t) = \sum_{\tau \in W_s,\, x \in D_\tau,\, x \prec t} b_{x,s} \text{ and } J_s(t) = \sum_{\tau \in R_s,\, x \in D_\tau,\, x \prec t} b_{x,s}$$

When the bursts are polynomials in the control program loop counters, the resulting sums can still be evaluated at compile time by tools such as the barvinok library [30].

2.2.3 Dependences between Tasks

Two operations are in dependence if they both access the same memory location, and at least one of the accesses is a write. According to [25], the shared memory locations must belong to a stream. (It would be possible to compute dependences on global variables, but this is not in the spirit of a streaming language.) By definition, writes always occur before reads, which defines the semantics and constrains valid runtime schedules: there are no WAR (anti-) dependences. Furthermore, the single assignment property implies that WAW (output) dependences do not exist.

To simplify the analysis, let us assume that each task access all elements of its windows. This condition can easily be checked if windows are accessed with constant values. Holes in input windows emulate subsampling, which we conservatively approximate as windows accessing their entire range of elements, and there are no holes in output windows since this would leave undefined elements in streams.

Let t be an instance of task τ that writes to stream s and let t' be an instance of task τ' that reads from stream s, with horizon $h_{t',s}$. The write window is $[I_s(t), I_s(t) + b_{t,s} - 1]$ and the read window is $[J_s(t'), J_s(t') + h_{t',s} - 1]$. There is a dependence if these two segments overlap, i.e., if:

$$I_s(t) \leq J_s(t') + h_{t',s} - 1 \wedge J_s(t') \leq I_s(t) + b_{t,s} - 1 \quad (3)$$

To these constraints, one must add conditions expressing the fact that t and t' are legal iterations, i.e., $t \in D_\tau$ and $t' \in D_{\tau'}$. Condition (3) enforces only that the writer of a given stream cell occurs before its readers. One can also impose a "Kahnian continuity" condition on streams (as in a FIFO), which states that a read can occur only if all stream cells with a smaller index have already been written. This is equivalent to considering that the read window starts from 0, i.e., is $[0, J_s(t') + h_{t',s} - 1]$, and the condition becomes simply:

$$I_s(t) \leq J_s(t') + h_{t',s} - 1 \quad (4)$$

as the second inequality of (3) is always satisfied.

Let us write $t \ \delta \ t'$ if these constraints are satisfied, which may be tested by any available tool. The δ relation defines the *instance-wise dependence graph* of the program. The result is a relation $\tau \Delta \tau'$, the *statement-wise dependence graph*, where the dependence pair (τ, τ') is labeled by the set of instances that satisfy the inequalities in (3)—or (4) with the Kahnian continuity semantics—or an over-approximation of this set. The statement-wise dependence graph can then be analyzed for defining valid code transformations, in particular for bounding streams or changing the granularity of task instances (e.g., by changing bursts). Note that while these transformations may share similar effects on parallelism and locality as nested loop optimizations [19], they change the task graph structure. Applying "valid" loop transformations to the control program only changes the order of task creation but, by definition of their validity, not the task graph itself.

If the index functions are linear, a linear programming tool may be sufficient. If not, the use of an SMT solver like Z3 [6], which can handle polynomials, is necessary. Z3 uses heuristics and pattern matching and is able to solve undecidable problems in acceptable time, albeit without guarantee of success. Also, as in the case of ordinary dependences, one may relax the integrality constraints on t and t' and obtain conservative results. The advantage of this approximation is that solving polynomials in the reals is decidable, while looking for integer solutions is not. Other approximation schemes are to be explored.

Whatever the situation, if t and t' are in dependence, then the writer t must be executed before the reader t'. Observe that the dependence relation for OpenStream tasks is *not* a subset of the sequential creation order, as is the case for sequential programs. Hence, this raises the possibility of deadlocks, whose study is addressed in Section 3.

Note also that, if a stream cell is never written, Condition (3) does *not* generate a dependence to its readers, which can thus be understood—wrongly—as tasks without a predecessor with respect to this access. Indeed, unlike in a control-driven program where a read to an un-initialized variable is executed anyway and returns garbage, for OpenStream programs, the readers should wait forever, i.e., should never be activated, resulting in a form of deadlock. This case has thus to be checked as a special case. For the polyhedral fragment, it can be detected statically, by checking that, for each stream s, its largest write index $I_s(t)$ (see Equation 1) is greater than or equal to its largest read index $J_s(t')$ (see Equation 2). This can be done, for example, as follows.

- If there is a task $\tau \in W_s$ whose iteration domain is infinite, the write index $I_s(t)$ in s is unbounded since the write bursts are positive. Thus, there is nothing more to check, all cells of s have a producer. If not, by Proposition 1, one may compute τ_{last}, the lexicographic last instance of $\tau \in W_s$, e.g., by using PIP [7], then compute $I_s(\tau_{last})$. The result is, for each $\tau \in W_s$, a piecewise polynomial function of the parameters.
- Then, one can check that, for each instance t' reading the stream s, there is a task $\tau \in W_s$ such that $J_s(t') + h_{t',s} \leq I_s(\tau_{last}) + b_{\tau_{last},s}$. Such a check is undecidable in general. However, one can give two complementary semi-algorithms. If we find t' such that $J_s(t') + h_{t',s} > I_s(\tau_{last}) + b_{\tau_{last},s}$, for all $\tau \in W_s$, then we know that a producer is missing, and there is a deadlock. This can be checked again with an SMT solver. Conversely, with a generalization of the Farkas lemma to polynomial constraints [14, 27], one can try to prove, for each subdomain of the parameters, that $J_s(t') + h_{t',s} \leq I_s(\tau_{last}) + b_{\tau_{last},s}$ for all possible t' reading s. If yes, we know that, for this subdomain, there is no deadlock due to an absence of producer.

Note that, if bursts and horizons are constants for a given task, then we can also compute the last read for each given task as explained above for the last write. The resulting test is then a comparison between polynomial expressions of the parameters. This is still undecidable in general, but again, it can be proved or disproved with semi-algorithms involving parameters only. Also, for a fixed value of the parameters, the test boils down to the evaluation of polynomial expressions.

2.2.4 Additional Remarks

Let us add some remarks concerning dataflow analysis, i.e., the problem, given an element i in stream s, of finding the position vector of the task that wrote $s[i]$. For OpenStream, this is, again, both trivial and impossible in general. This vector and the associated task must satisfy the constraints $t \in D_\tau$ and $I_s(t) \leq i \leq I_s(t) + b_{t,s} - 1$. This is a constraint satisfaction problem, which may be solved if I_s is linear

or a low degree polynomial, but which seems impossible in general—this is a form of quantifier elimination, which is impossible in general for integers and polynomials. Due to its special form, it may be that the I_s function may be inverted, giving a closed form expression for i as suggested earlier. This is a subject for future work.

Finally, let us conclude this study on dependence analysis with a remark concerning barriers. There is not much about barriers in [25], but they certainly exist in OpenMP, and are a prominent feature in [23]. The semantics here is that when the control program executes a barrier, it stops until all created tasks have terminated. The first consequence is that the presence of barriers does not change anything in the $\prec$ relation, hence has no impact on the write and read functions I_s and J_s, and does not change stream dependences. However, it adds new "control-induced" dependences to the instance-wise dependence graph. If b is the position vector of a barrier and if $t \prec b$ and $b \prec t'$, then it adds a dependence from t to t' to the stream-based dependences. Note that this dependence is some variation on the lexicographic order, hence it fits in the polyhedral model and does not increase the difficulty of deadlock detection and scheduling.

3. DEADLOCKS IN OPENSTREAM

OpenStream programs have strong similarities with KPNs (Kahn Process Networks), but the equivalence of the two models, or of restricted versions, is not obvious. In OpenStream, unlike in KPNs, streams can be read and written by several actors, but in a single assignment manner and with a deterministic interleaving, fully expressed by the control program only. General KPNs are deterministic too, but the decision of reading in one stream or another can depend on the data that circulate on the streams. Because of these differences, it is not clear how to transfer the characterization and detection of deadlocks in KPNs to OpenStream programs. In fact, in [20], Parks showed that it is undecidable to detect if a KPN will deadlock (i.e., terminates). The argument is that BDFs (Boolean Data Flow) are a particular case of KPNs and because BDFs can simulate Turing machines (as shown by Buck [2]), they lead to the undecidable halting problem. The proof of Buck exploits the fact that, unlike SDFs (Synchronous Data Flow), BDFs have two actors *select* and *merge* that allow conditional token consumption and production, and these conditions can depend themselves on the history on streams—typically what was read/written on the tape of the Turing machine. In such a proof, undecidability comes from the complexity of the computation through the streams but does not say anything on the complexity of the stream *structure*. Similarly, one could imagine a KPN with a single process whose program is a universal Turing machine and reading an empty stream whenever the Turing machine halts. Such undecidability proofs do not give any insight for the situation of OpenStream where the readers and writers of a particular stream element do not depend on the computations themselves, only on the control program that creates tasks. Hence, we need to develop new proofs dedicated to OpenStream to answer the following questions:

- Which dependence structures lead to deadlocks?
- Can deadlocks depend on the execution order of tasks?
- Is it decidable to detect deadlocks?

Section 3.1 addresses the first two questions. To answer the last one, we use two ingredients: a particular stream and dependence structure exposed in Section 3.3 and, as for the problem of race detection in X10 (see Section 4.2), the fact that it is possible to encode, in a polyhedral fragment, multi-variate polynomials as the number of iterations in a set of affine (imperfectly) nested loops. This construction can find other applications and is recalled in Section 3.2.

3.1 Characterization of Deadlocks

The most intuitive method for proving the absence of deadlocks consists in building a schedule. A schedule is a function σ from the set of task instances to the nonnegative integers $\mathbb{N}$ such that $\sigma(t') \geq \sigma(t) + 1$ whenever t' depends on t. If a schedule exists, even if a "parallel front"—set of tasks t with same value $\sigma(t)$—is infinite, a runtime scheduler following σ has always some *ready* task instance to activate, thus does not lead to a deadlock. A ready task instance is an instance for which all predecessors in the dependence graph have terminated. However, remember that an instance reading a cell with no producer should not be considered as ready for activation. From now on, we thus exclude this situation, assuming it has already been considered as explained in Section 2.2.3. In other words, we assume that all stream cells that belong to the input window of some instance also belong to the output window of some instance so that a ready task instance can indeed be activated.

If the I_s and J_s functions are linear, checking for the existence of an affine schedule can be done using algorithms, standard in the polyhedral community, based on the affine form of the Farkas lemma. In case of polynomial functions, the special form of (3) may simplify the construction of a schedule. In the general case, extensions of the Farkas lemma can be used for generating schedules with polynomial constraints, as explored in [10]. Nevertheless, to better understand the equivalence between schedules and absence of deadlock situations, we need to characterize them in terms of dependence paths. Consider the following example in an OpenStream-like format:

```
    stream s, t;
c   read once in t;
    for (i = 0; ; i++) { /* infinite domain */
a     write once in s; read once in t;
b     write once in t; read once in s;
    }
```

Here, c depends on $b(0)$, which produces the first value of the stream t, while other values produced by $b(i)$ for $i > 0$ are read by $a(i-1)$. As for stream s, it induces a dependence from $a(i)$ to $b(i)$. In other words, for all i, $a(i)$ depends on $b(i+1)$, which depends on $a(i+1)$, etc. The program cannot start: an infinite number of tasks is created but none of them can execute. This is a case of deadlock where, in the graph defined by dependences among task instances, there is no cycle, but an infinite path. However, with the Kahnian continuity semantics, there is a cycle: $a(i-1)$ depends on $b(i)$ through stream t, and $b(i)$ depends on $a(i-1)$ through stream s as it depends "functionally" on $a(i)$.

As another simple example, consider:

```
   stream s;
   for(i = 0; ; i++)
a   read once in s; write once in s;
```

Each instance of a has a dependence on itself (Condition (3)), hence cannot be scheduled. This code has a deadlock.

The following proposition explicits these situations in general. We are not interested in the case of infinite programs where fairness in the runtime scheduler may be needed to

ensure all tasks make progress. We say that *a program has no deadlock* if, at runtime, whatever the tasks already activated by the runtime scheduler and for each task instance not yet activated, there is always an order of activations such that this task instance will be ready, and thus possibly activated.

PROPOSITION 2. *There is a deadlock if and only if there is no schedule, more precisely iff there is an infinite number of instances (possibly equal) of the same task that depend (possibly indirectly) on each other in the reverse order of their creation. If the control program generates a finite number of task instances, there is a deadlock iff the instance-wise dependence graph has a cycle. The same is true for an infinite number of instances and the Kahnian continuity semantics.*

PROOF. As task creation does not depend on dependences, there is no deadlock due to task creation, we only need to consider when tasks start executing. We show the following, using arguments similar to those used for the computability of systems of uniform recurrence equations (see [16, 5]).

Property 1. There is no schedule if and only if there exists a task instance t such that the length of the dependence paths leading to it is unbounded, i.e., $\sigma(t) = +\infty$ where

$$\sigma(t) = \sup\{\text{length}(\mathcal{P}) \mid \mathcal{P} \text{ dependence path leading to } t\}$$

Indeed, if $\sigma(t)$ is finite for any t, and if t' depends on t, then any path $\mathcal{P}$ leading to t, extended with the dependence from t to t', gives a path leading to t', thus $\text{length}(\mathcal{P}) + 1 \leq \sigma(t')$, and finally $\sigma(t)+1 \leq \sigma(t')$. Thus σ is a schedule. Conversely, if a schedule σ' exists, then, by induction on the length ℓ of a path leading to t, $\sigma'(t) \geq \ell$, thus $\sigma(t) \leq \sigma'(t) < +\infty$.

Property 2. Each task instance depends (directly) on a finite number of other task instances. Indeed, a task instance reads only a finite number of streams and bursts/horizons are finite. This is true also in the Kahnian continuity case as streams start at 0 (and not $-\infty$). This implies that there is no schedule if and only if there is a task instance with an infinite dependence path leading to it. Indeed, if such a task instance t exists then, of course, $\sigma(t) = +\infty$. Conversely, since $\sigma(t) = \max\{\sigma(u) \mid t \text{ depends directly on } u\}$, then $\sigma(t) = +\infty$ implies that $\sigma(u) = +\infty$ for at least one direct predecessor u of t. Continuing this way, one can construct, by induction, an infinite dependence path leading to t (this is nothing but König's lemma).

Property 3. The previous property implies that if there is no schedule, some task instance can never be ready in finite time (due to an infinite dependence path leading to it), whatever the tasks that have been activated so far, thus what we call a situation of deadlock. Conversely, if there is a schedule σ, the length of any dependence path leading to t is bounded by $\sigma(t)$, thus each task instance depends (even indirectly) on a finite number of other task instances. Thus, there exists an execution order that will make it ready.

Property 4. Now, consider an infinite dependence path leading to a task instance t, i.e., an infinite sequence of task instances $(t_i)_{i\in\mathbb{N}}$, such that $t_0 = t$ and t_i depends (directly) on t_{i+1}. Now, let i_0 such that t_{i_0} is the first created task instance in the path and i_0 is minimal. Then, by induction on j, define i_{j+1} such that $t_{i_{j+1}}$ is the smallest (following the creation order $\preceq$) task instance t_i with $i > i_j$ and i_{j+1} is minimal. The "smallest" exists because the order $\preceq$ has no infinite descent (the number of task instances created before a given task instance is finite). By construction, $t_{i_j} \preceq t_{i_{j+1}}$ as the first one is the smallest element in a larger set and t_{i_j} depends (possibly indirectly, by transitivity) on $t_{i_{j+1}}$ because $i_j < i_{j+1}$. Finally, as there is a finite number of tasks in the program, at least one task τ appears infinitely many times in this subsequence $(t_{i_j})_{j\in\mathbb{N}}$. In other words, if there is a deadlock, there is a task τ and an infinite sequence of instances $(\tau_i)_{i\in\mathbb{N}}$ of τ such that τ_i depends on τ_{i+1} in the non-strict reverse order of creation (i.e., $\tau_i \preceq \tau_{i+1}$).

This shows most of Proposition 2; when there is only a finite number of task instances, an infinite path traverses at least twice the same task instance, thus there is a cycle. It remains to consider the case of an infinite number of task instances, assuming the Kahnian continuity semantics. If a task instance $\tau(i)$ depends (directly or by transitivity) on a task instance $\tau'(j)$, then the task instance that directly depends on $\tau'(j)$ in this dependence path—a task instance reading a stream element written by $\tau'(j)$—also depends on $\tau'(k)$ for all $k \preceq j$ (this depends on the reasonable assumption that, in OpenStream, all task instances of a given task always write in the same streams). Then, by transitivity, $\tau(i)$ depends on $\tau'(k)$ for all $k \preceq j$. Finally, if there is a deadlock, as a particular case of Property 4 in this proof, there is a task τ and two position vectors i and j such that $i \preceq j$ and $\tau(i)$ depends on $\tau(j)$, thus $\tau(i)$ also depends on $\tau(k)$ for all $k \preceq j$, in particular $k = i$, which forms a cycle. □

This shows that finding a schedule is indeed a certificate ensuring the absence of deadlocks (in the sense defined previously). Now, consider a schedule σ. If a stream s is such that, for some positive integer ℓ_s, all indices $i \leq j - \ell_s$ of s are dead for σ (i.e., already produced and consumed) whenever index j is written, then s can be implemented as a "bounded stream" of size ℓ_s. Conversely, if s is implemented this way in the runtime execution environment, keeping track of the current smallest live index i and blocking writes from $i + \ell_s$ and beyond, then no deadlock will occur at runtime. Indeed, such a mechanism is equivalent to extending the instance-wise dependence graph with "back-pressure" dependences from τ' to τ, where τ' reads $s[i]$ and τ writes $s[j]$, with $i \leq j - \ell_s$. According to Proposition 2, since this new dependence graph has at least one schedule (e.g., σ is valid), it has no deadlock. Actually, there is a small subtlety: the previous proof assumed that each task instance depends on a finite number of task instances. With these additional dependences, this may not be true if an element e of s has an infinite number of readers. But in this case, with a finite number of computation resources, there cannot be a schedule with a bounded stream s anyway, as e can never be reused.

It is also important to note that OpenStream, in its simplest form studied here, only defines dependences among task instances, which are the only constraints for the runtime scheduler. Thus, standard loop transformations have no effect on the program execution, as they will not change dependences. What can be of interest however is to overconstrain the runtime, by *adding* dependences. For example, one can add artificial back-pressure dependences in the code. One can also increase the bursts and horizons to change the granularity of communications, or merge several tasks. Now, the validity of such code transformations is not anymore a problem of preserving dependences (as for standard languages) but of avoiding deadlocks. This is why deadlock detection is important and addressed in Section 3.3. Before, we need an extra ingredient, related to the expressiveness of polyhedral loops, that we recall in Section 3.2.

3.2 Construction of Counting Nested Loops

Our aim here is to recall a proof technique, borrowed from [33], which, given a multi-variate polynomial $Q(x)$ with nonnegative integer coefficients, builds a set of nested loops computing $Q(x)$, for some multi-dimensional parameter x, using only increments by constant integers. This technique shows how expressive simple affine nested loops can be. It is then enough to replace each increment d by the introduction of a task instance accessing some stream with a burst of d, and the corresponding read or write index increment after these loops is $Q(x)$.

Let us select one particular variable x_1 and write $Q(x) = Q(x_1, \vec{x_r})$ where $\vec{x_r}$, the vector of remaining variables, may be empty. The first *difference* Q_1 of Q in x_1 is:

$$Q_1(x_1, \vec{x_r}) = Q(x_1 + 1, \vec{x_r}) - Q(x_1, \vec{x_r}).$$

The following program computes $Q(x)$:

```
q = Q(0, x_r⃗);
for (i = 0; i < x_1; i++)
  q += Q_1(i, x_r⃗);
```

The degree of Q_1 is at most $m - 1$, where m is the degree of Q, and its coefficients are still nonnegative. A similar construction applied to Q_1 creates a second level loop involving the second difference of Q. Iterating at most m times results in a program where the increments are of degree zero in x_1, i.e., do not involve x_1. The construction is then applied recursively to the next variables in $\vec{x_r}$, and the final result is a program where all increments are positive integers. A complete example is given in [33].

3.3 Detecting Deadlocks is Undecidable

This section shows that it is in general undecidable (thanks to a reduction from Hilbert's tenth problem) to detect if an OpenStream program has:

- a functional deadlock;
- a spurious or a functional deadlock;
- a stream causal schedule.

A functional deadlock is a deadlock situation as exposed in Section 3.1, assuming the general semantics of dependences given by the constraints in (3). A spurious deadlock is a deadlock that arises only because of the Kahnian continuity semantics, i.e., if a read in a stream at a given index must wait for all writes in the stream at smaller indices. A causal schedule is a schedule where writes to a given stream occur in the same order as their indices, i.e., in the same order as the creation of the corresponding task: $\sigma(t) < \sigma(u)$ if t and u write to the same stream and $t \prec u$ (in this case, the index written by t is smaller than the index written by u).

The proof is based on the following construction, inspired by a similar proof about race conditions in X10 [33]. It uses the same link to Hilbert's 10th problem, the same ingredient for building polynomials (as presented in Section 3.2), but of course a different program structure as neither deadlocks and races, nor X10 and OpenStream, have particular connections. Hereafter, P and Q are two multivariate polynomials (with n variables), with nonnegative coefficients (actually, they are the positive and negative parts of a polynomial $R = P - Q$ used to relate to Hilbert's 10th problem). The code can use only horizons and bursts equal to 1, thanks to additional loops, or horizons and bursts can be used to emulate the constants appearing in the construction of Section 3.2.

In the following code, D is either the multidimensional first orthant scanned along diagonal hyperplanes (to prove undecidability for a program with infinitely many task instances) or the cube of size N in this orthant (to prove undecidability for a family of parametric programs with one parameter N).

```
    s, t streams;
    for (x ∈ D) {
R1    read Q(x) times in t;
W1    write P(x) times in t;
S     read once in t and write once in s;
T     read once in s and write once in t;
R2    read P(x) times in t;
W2    writes Q(x) times in t;
    }
```

Following Section 3.2, we can write affine loops so that R_1 reads $Q(x)$ times in t (same for the other polynomial expressions). The dependence graph has only one possible cycle, involving S and T, other tasks cannot induce deadlocks. For each iteration of x, there are $P(x) + Q(x) + 1$ writes and reads in stream t and one write and one read in stream s, thus functional dependences among task instances can only involve instances corresponding to the same iteration x. Because of stream s, there is always a dependence from $S(x)$ to $T(x)$. Concerning stream$t$, $T(x)$ writes in position $P(x)$ (if we start positions at 0, without counting all previous iterations of the x loop) and $S(x)$ reads in position $Q(x)$.

If $P(x) = Q(x)$, there is a functional dependence from $T(x)$ to $S(x)$, thus a deadlock. Otherwise, there always exists a schedule for iteration x: execute $W_1(x)$ and $W_2(x)$, then $S(x)$—which reads a value produced either by $W_1(x)$ or by $W_2(x)$—then $T(x)$, and finally $R_1(x)$ and $R_2(x)$. Thus, there exists a deadlock if and only if there exists a nonnegative vector x such that $P(x) = Q(x)$, i.e., $R(x) = 0$. This is undecidable as a variant of Hilbert's 10th problem (one can examine all possible signs of variables or replace x in Hilbert's problem by $x = x_1 - x_2$, where x_1 and x_2 are nonnegative). This is for functional deadlocks.

Now let us consider spurious deadlocks, i.e., with the Kahnian continuity semantics. If $P(x) = Q(x)$, there is still a functional deadlock. If $P(x) < Q(x)$, the situation is depicted in Figure 2: $S(x)$ reads in a position written by an instance of $W_2(x)$, beyond the position written by $T(x)$. Thus, with the Kahnian continuity condition given by (4), there is a (spurious) dependence (dotted arrow in the figure) from $T(x)$ to $S(x)$, thus a cycle (and therefore a deadlock).

Finally, if $P(x) > Q(x)$ (depicted in Figure 3), there is a schedule and even a causal schedule: execute successively $W_1(x)$, $R_1(x)$, $S(x)$—reading a value produced by $W_1(x)$—, then $T(x)$, $W_2(x)$, and finally $R_2(x)$. In conclusion, there is a deadlock (and here, equivalently, no causal schedule) if and only if there exists a nonnegative x such that $P(x) \leq Q(x)$. This leads to another variant of Hilbert's 10th problem: for a given polynomial R, is there a nonnegative x such that

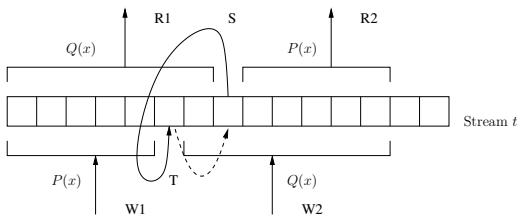

Figure 2: Spurious dependence and cycle.

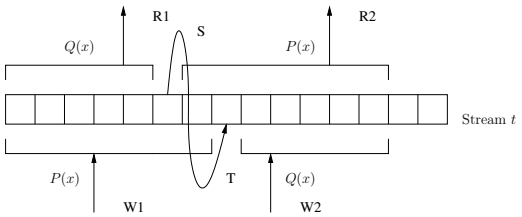

Figure 3: No deadlock, and even a causal schedule.

$R(x) \leq 0$? If this problem was decidable, then one could also decide if $R(x) = 0$, since $R(x) = 0$ if and only if $R^2(x) = 0$, which is also equivalent to $R^2(x) \leq 0$.

4. RELATED WORK

The work on OpenStream is yet another attempt to provide a safe and efficient environment for parallel programming. In Section 4.1, this effort is put in perspective with respect to many languages and systems with the same target. Section 4.2 discusses previous work on the X10 language, from which we have borrowed several methods and ideas.

4.1 Other Streaming/Dataflow Languages

There are two motivations for the design of streaming languages. The first one is that they fit well with important application domains, like reactive systems and signal processing applications. The second one is that they also fit with distributed architectures, processors implementing processes and network links implementing streams. They therefore have the potential of improving both programmer productivity and hardware performance.

Kahn process networks (KPN) [15] form the basis for most deterministic languages based on stream computing concepts. In his survey of stream processing [28], Stephens classifies stream processing systems based on three criteria: synchrony, determinism, and the type of communication channel. Fundamentally, stream-based models of computation all share the same structure, which can generally be represented as a graph, where computing nodes are connected through streaming edges. However, cyclic networks can lead to deadlocks or unbounded growth of in-flight data, which has spurred the development of restricted forms of KPN such as static data-flow (SDF) [18] and cyclo-static data-flow (CSDF) [1]. While processes in KPNs execute asynchronously and can produce or consume variable amounts of data, CSDF processes have a statically-defined behavior. With rates of production and consumption known at compile time, it is possible to statically decide whether the execution is free of deadlocks and to statically schedule the execution. It can also guarantee the absence of resource deadlocks when executing on bounded memory, a realistic restriction. SPDF [12] is another extension of SDF where production/consumption rates can be parametric. StreamIt is an instantiation of CSDF, building on the strong static restrictions of the underlying model to enable aggressive compiler optimizations. It achieves excellent performance and performance portability across a variety of targets [13] for a restricted set of benchmarks that properly map on this model.

It is also worth comparing our OpenStream polyhedral fragment with restricted classes of process networks amenable to polyhedral modeling. This includes the modular scheduling approach of Feautrier [9] and the automatic construction of process networks from static control loop nests by Kienhuis et al. [17] and Verdoolaege [29]. These process networks are meant as formal models and intermediate representations. They are generally extracted from imperative code rather than allowing the programmer to control the construction of the network. As a result, they only involve static analyses and decision procedures on (multi-dimensional) affine relations, although communications among Verdoolaege's PPN involve polynomial index expressions. Finally, note that the pionneering work of Clauss and Meister on spatial locality [4] does introduce polynomial expressions, but these are not the subject of further dependence analysis unlike our index expressions.

All of these diverse approaches to stream programming have the potential to help mitigate the memory wall, but they only apply to restricted classes of applications. Programs are generally considered built around regular streams of data, which fits the models where channels of communication are implemented as single-producer and single-consumer FIFO queues. We believe that the development of applications for current and upcoming multi- and many-core architectures requires a more general model, where communication patterns are not always regular or statically defined, but can occur and be exploited dynamically. The insight that the flow of data plays a central role in all programs is an essential one, but data-driven computations often need to be predicated by complex control flow due to data-dependent, sporadic events, as is the case in synchronous control programs. This complementarity of control and data flow is covered in depth in the Control-Driven Data Flow framework [23]. This paper explores the case of polyhedral OpenStream programs, supporting the insight that new approches to streaming should preserve the strong properties provided by some existing models, like functional determinism or deadlock-freedom.

4.2 Analysis of Polyhedral X10

The US Department of Energy led a research program with the aim of increasing parallel programming efficiency and productivity. The X10 language [26], developed at IBM Research, is one outcome of this program. It is an object-oriented language of the Java family.

Concurrency is expressed in X10 through two constructs, `async S` and `finish S`, where `S` is an arbitrary statement or statement block. The effect of `async S` is to create a new *activity* or lightweight thread, which executes `S` in parallel with the rest of the program. The effect of `finish S` is to launch the execution of `S`, then to wait until all activities created inside `S` have terminated. In some cases, it may be necessary to synchronize several parallel activities, which can be achieved with *clocks*. Clocks are an improved version of the classical barriers. They come in two flavors: named and implicit. It can be proved that, if only implicit clocks are used, an X10 program is deadlock-free. However, in contrast to OpenStream, it may have races, i.e., non-determinism. To cross a barrier, the program must execute an `advance` statement. The different activities that *registered* to the same clock cannot proceed until all of them have executed an `advance` statement. One may associate to each clock and each activity a counter that gives the number of `advance` statements executed so far by the activity and its ancestors since the creation of the clock.

Detection of races in X10 program is only possible, with today's techniques, for a polyhedral subset of the language, in

which tests and method calls are forbidden, data structures are restricted to arrays, and loops are restricted to counted loops. X10 is simpler than OpenStream in that its execution order or *happens-before* relation can be extracted from the program text (or its AST) in a straightforward manner, instead of being the result of a complex dependence calculation. As a result, the array dataflow analysis for polyhedral sequential programs [8] can be extended for polyhedral X10 programs [32]. Dataflow analysis finds the *source* of each value generated by the program. A race exists if a value has several possible sources; in that case, the program may not be deterministic.

Clocks may be used to remove a race and re-establish determinism. Informally, two instances can happen in parallel, therefore creating a race, only if they belong to the same phase of their enclosing clocks, i.e., if their counters are equal. However, even for polyhedral programs, when an `advance` is enclosed in several loops, its counter is a polynomial. Again, it can be obtained by classical counting algorithms but, as for deadlock detection in OpenStream, race detection then entails the resolution of polynomial equations in integers, and hence is undecidable (see [33] for details).

In fact, polynomials crop up whenever a program or a language needs to map a multi-dimensional object into a lower-dimensional one. For OpenStream, one maps a multi-dimensional sequence of values into a one-dimensional stream. For X10, a multi-dimensional set of operations is mapped into a one-dimensional sequence of `advance`. A more trivial example is the mapping of a multi-dimensional array into linear memory. The multi-dimensional channels of CRP [9] were designed to avoid this phenomenon.

5. CONCLUSION

This preliminary study has shown how an interesting fragment of the OpenStream language can be defined where polyhedral techniques are fully or partially applicable. This fragment revisits the traditional restrictions on imperative control flow in the context of the control program and of the task streaming clauses of an OpenStream program.

- On the bright side, dependences among task instances may be statically computed, and the formalization of the polyhedral fragment of OpenStream allowed to derive important undecidability properties about dynamic schedulability and the absence of deadlocks. Interestingly, these properties do not directly relate to the existing literature on (cyclo-)static dataflow graphs or Kahn networks, partly due to the yet incompletely understood simulation of one model into another. In particular, the undecidability of deadlocks derives from the polynomial nature of stream indexing alone, rather than the complexity of Boolean conditions as in Boolean dataflow [2].
- Dataflow analysis, scheduling for granularity control, and compilation-time deadlock detection, show a more ambivalent picture: our polyhedral fragment leads to polynomials of arbitrary degree as stream indexing functions. To enforce affine stream indexing, one may consider control programs with one-dimensional loops only; this enables all polyhedral tools but only to a very limited set of OpenStream programs.

We are considering multiple directions to deal with the polynomial constraints exposed in our polyhedral fragment. The most immediate one is to rely on affine approximations of the indexing functions, e.g., overestimating the range of the access indices J and I in the dependences. One may also take advantage of the special properties of I and J, such as their monotony w.r.t. the lexicographic order of position vectors (task activations), or the fact that the counter of the innermost loop in the control program always occur linearly. Alternatively, modern SMT solvers can handle polynomial problems, albeit with no guarantee of success. We believe a more promising approach is to explore polynomial extensions to native polyhedral techniques [10], with heuristics for the construction of low-degree polynomial schedules. These extensions were proposed, thanks to recent generalizations of the Farkas lemma (a key technique in polyhedral optimization) to polynomial constraints [14, 27], in order to extend automatic parallelization methods to programs involving—directly or indirectly—polynomials. The clocks in polyhedral X10 are an example; the stream-induced dependences of OpenStream bring another motivation to look beyond the affine form of the Farkas lemma.

One may also wonder if the analysis techniques proposed in this paper may be extended to more advanced features of OpenStream, like nested tasks, variadic streams, or data-dependent conditionals. More generally, this work also encourages further studies on the interactions between the semantics of parallel constructs, static dependence analysis, static scheduling, and dynamic scheduling, and the impact of the choice of the language constructs on both the programmer and the compiler.

Acknowledgments. This work was partly supported by the French "Investissements d'Avenir" ManycoreLabs grant. We are indebted to Antoniu Pop for the formalization of OpenStream's semantics and its properties. We would also like to thank the anonymous reviewers for their valuable feedback, and for pointing out the incomplete formalization of deadlocks resulting from the lack of writers.

6. REFERENCES

[1] G. Bilsen, M. Engels, L. R., and J. A. Peperstraete. Cyclo-static data flow. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP'95)*, pages 3255–3258, Detroit, Michigan, May 1995.

[2] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, University of California at Berkeley, 1993.

[3] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Application to analyze and transform scientic programs. In *ACM International Conference on Supercomputing (ICS'96)*, pages 278–285, 1996.

[4] P. Clauss and B. Meister. Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. *SIGARCH Comput. Archit. News*, 28(1):11–19, Mar. 2000.

[5] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization.* Birkhauser, 2000. ISBN 0-8176-4149-1.

[6] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pages 337–340. Springer-Verlag, 2008. Corresponding software:

https://github.com/Z3Prover/z3.

[7] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988. Corresponding software: www.piplib.org.

[8] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, Feb. 1991.

[9] P. Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34:459–487, 2006.

[10] P. Feautrier. The power of polynomials. In *5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*, Amsterdam, The Netherlands, Jan. 2015.

[11] P. Feautrier and C. Lengauer. The polyhedral model. In D. Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[12] P. Fradet, A. Girault, and P. Poplavko. SPDF: A schedulable parametric dataflow model of computations. In *Conference on Design, Automation and Test in Europe (DATE'12)*, pages 769–774, 2012.

[13] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, pages 151–162, San Jose, CA, Oct 2006.

[14] D. Handelman. Representing polynomials by positive linear functions on convex polyhedra. *Pacific Journal of Mathematics*, 132(1):35–63, 1988.

[15] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP'94*, pages 471–475. North Holland, 1974.

[16] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[17] B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving process networks from Matlab for embedded signal processing architectures. In *Conference on Hardware/Software Codesign (CODES)*, 2000.

[18] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–25, 1987.

[19] D. Padua. Parallelization, automatic. In D. Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[20] T. M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, Dec. 1995.

[21] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *International Journal on High Performance Computing Architecture*, 23(3):284–299, 2009.

[22] A. Pop and A. Cohen. A stream-computing extension to OpenMP. In *6th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC'11)*, pages 5–14, Heraklion, Greece, 2011. ACM.

[23] A. Pop and A. Cohen. Control-driven data flow. Technical Report RR-8015, INRIA, July 2012.

[24] A. Pop and A. Cohen. Work-streaming compilation of futures. In *5th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES'12, associated with ETAPS)*, Mar. 2012.

[25] A. Pop and A. Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–25, 2013.

[26] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification version 2.2, Mar. 2012. x10.sourceforge.net/documentation/languagespec/x10-latest.pdf.

[27] M. Schweighofer. An algorithmic approach to Schmüdgen's Positivstellensatz. *Journal of Pure and Applied Algebra*, 166:307–319, 2002.

[28] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[29] S. Verdoolaege. Polyhedral process networks. In *Handbook of Signal Processing Systems*, pages 1335–1375. Springer, 2013.

[30] S. Verdoolaege and M. Bruynooghe. Algorithms for weighted counting over parametric polytopes: A survey and a practical comparison. In M. Beck and T. Stoll, editors, *The 2008 International Conference on Information Theory and Statistical Learning*, July 2008.

[31] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok rational functions. In *Algorithmica*, 2007. Corresponding library: barvinok.gforge.inria.fr.

[32] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat. Array dataflow analysis for polyhedral X10 programs. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*, pages 23–34, 2013.

[33] T. Yuki, P. Feautrier, S. V. Rajopadhye, and V. Saraswat. Checking race freedom of clocked X10 programs. http://arxiv.org/abs/1311.4305, Dec. 2013.