Static Analysis of OpenStream Programs

Alain Darte
CNRS Compsys

Paul Feautrier
ENS-Lyon Compsys

Albert Cohen and Antoniu Pop
Inria Parkas

October 23, 2013

Executive Summary

The object of this report is to evaluate the possibility of applying polyhedral techniques to the parallel language OpenStream, which is developed by INRIA Parkas. When applicable, these techniques are invaluable for compile-time debugging and for improving the target code for a better adaptation to the target architecture.

OpenStream is a two-level language, in which a sequential control code directs the initialization of parallel task instances that communicate through *streams*. OpenStream programs are deterministic by construction, but may have deadlocks. If the control code is polyhedral, one may statically compute, for each task instance, its read and write indices for each stream. These indices may be polynomials of arbitrary degree. When linear, the full power of the polyhedral model may be brought to bear for dependence and dataflow analysis, scheduling and deadlock detection, and program transformations.

In the general case, one can think of two approaches: the first one consists in over-approximating dependences until problems become linear. In the second approach, one first leverage modern developments in SMT solvers, which allow them to solve polynomial problems, albeit with no guarantee of success. Furthermore, the task index functions have special properties that may be used to construct original analysis algorithms. Three preliminary results in this direction are 1) the proof that deadlock detection is undecidable in general, 2) a characterization of deadlocks in terms of dependence graphs, which implies that streams can be safely bounded as soon as a schedule exists with such sizes, and 3) a preliminary analysis of some solvable cases.

Contents

1 Introduction: State of the Art

1.1 Parallel Programming Languages: How and Why

The evolution of the peak performances of processors has, for a long time, be due to the constant increase of clock frequency, the exploitation of instruction-level pipelining and parallelism, and the introduction of hardware mechanisms for reordering instructions, hiding memory latencies, predicting conditional branches, etc. For the every-day programmer, this performance increase has been felt with no need to change programming paradigms. To bridge the gap between the (sequential) programming language and the low-level exploitation of processor architectures, compilation techniques (program transformations and optimizations) have been developed: loop unrolling, register allocation, instruction selection, peephole optimizations, optimizations beyond basic blocks, which are almost transparent for the user. As for parallel languages, they were used mostly for programming super-computers such as large-scale distributed memory systems.

In the last years however, with the parallelism becoming mainstream, the pressure has became much stronger on the compilers and on the programming languages. Embedded processors for example, designed with a simpler architecture to reduce power consumption and size, with less hardware support for runtime optimizations, still offer good performances, for some classes of applications, but only thanks to optimizing compilers (e.g., exploitation of VLIW or vector operations) or even macros visible to the programmer (e.g., multi-media instructions). The development of GPUs (graphic processors) has led to considerable performance gains, again for some applications, but at the price of a loss of programming productivity: programming in CUDA or OpenCL is a task for specialists. Programming FPGAs is feasible only for experts and at low level. Multicores, such as the Kalray MPPA, offer impressive computational power to a larger public but expose the user to the difficulty of parallel programming. Exascale computing, with difficult power and fault tolerance issues, leads to the same conclusion: it is not possible to exploit the peak performances of these machines for all applications (unlike "general purpose" processors) and by programming them almost transparently.

In other words, the development of these new hardware accelerators (FPGA, GPGPU, multicores), accessible to a larger public, but sometimes heterogeneous and always hard to program, has put a new pressure on programming languages and compilers to achieve the three "P": **portability**, **programmability**, **performance**. In this quest, giving up on the idea of automatic parallelization from sequential codes, a large number of research projects and HPC (high-performance computing) programming languages have been proposed, including early PGAS (partitioned global address space) languages (e.g., Co-Array Fortran or CAF, Unified Parallel C or UPC), APGAS (asynchronous PGAS) languages (e.g., X10, Chapel), languages focusing on heterogeneous platforms (e.g., Lime, OmpSs), streaming languages (e.g., SDF, SPDF, Σ-C, StreamIt, OpenStream), multi-threaded approaches (e.g., Cilk, threading building blocks or TBB), actor/object-based languages (e.g., S-Net, Charm++, CnC or Concurrent Collections, Swarm), runtime-based approaches (e.g., StarPU), source-to-source compilers (e.g., PIPS, Par4All), or even DSLs (domain-specific languages). This list is not even exhaustive. A survey on such languages has been organized in Lyon by the Compsys team, with the context of the ManycoreLabs project in mind, as part of the Spring 2013 thematic quarter on compilation, see the web site `http://labexcompilation.ens-lyon.fr/hpc-languages`.

All these approaches propose different trade-offs between the expressiveness of the language, the performance that can be obtained, manually or automatically, the style of programming. Each approach induces different analysis and compilation problems and relies on different interactions with the runtime system. For the Kalray MPPA, for which managing the core activities and the memory used to communicate between cores is mandatory, considering streaming-like languages has been proposed. A streaming language is more or less a language where tasks communicate explicitly, in a data-flow manner, through buffers, in general one-dimensional buffers similar in the spirit to FIFOs, following a semantics close to Kahn process networks. Even in this restricted class of languages, the expressiveness of the language induces different analysis and optimization problems: in general, the more expressive the language is, the more difficult it is to analyze. Important questions to address are:

- What is the semantics of the language?
- Is it possible for a program to induce deadlocks?
- Is is possible for a program to induce data races?
- If yes, is it possible to check these properties at compile-time?
- Is it possible to analyze (exactly or approximately) the flow of computations?
- Can we bound, statically, the size of the memory required for communications?
- Is it possible to change the granularity of the program, i.e., the size of the atomic computation with respect to communication?

The goal of this report is to explore these questions for the language OpenStream, proposed by the Parkas team, in the light of two other languages, CRP (Communicating Regular Processes), which, unlike OpenStream, manipulates multi-dimensional communicating buffers, and X10, whose recent analysis presents some (loose) similarity with the analysis of OpenStream.

1.2 Dependence Analysis for Parallel Languages

Optimizing compilers, and in particular parallelizing compilers, try to transform the source program into an equivalent program that is better adapted to the target architecture, or runs faster, or has more parallelism. Equivalence here means that the final output of the program (typically values of the program variables stored in memory) is unchanged by the optimization. The problem is that, in general, program equivalence is undecidable, so one needs a way to "understand" some key features of the program to guarantee that a restricted class of transformations is valid. This is the purpose of program analysis and, in particular, dependence analysis. Briefly speaking, dependence analysis consists in deriving a relation δ between program operations such that executing u before v for all operations (u, v) such that $u\ \delta\ v$ guarantees that the semantics of the program is preserved. A classic sufficient condition is to use Bernstein's condition, which states that if u is executed before v in the original program and both operations access the same memory cell, at least one as a write, then $u\ \delta\ v$. See [13] for more details.

For such an approach to be feasible, several concepts need to be defined (see also [14]). First, what is the semantics of the original program (is it deterministic?) and, in particular, what does it mean that u is executed before v? For a sequential language, by definition, all operations are done one after the other, so reads and writes are totally ordered. Furthermore, in imperative languages such as C or Fortran, this sequential order is explicit. For parallel languages, the situation is more complex: in languages such as OpenMP or X10 (at least in simpler forms), parallelism is explicit, thanks to the keywords of the languages, but the order of operations is only a partial order (and, for shared variables, data races may exist). For languages based on recurrence equations [23], the semantics of the program is explicit (i.e., it can be read directly from the syntax of the program), but the order of computations is implicit and is induced by single assignment on variables (writes must be done before reads). Even worse, there may be no such order, i.e., the specification of the program may induce deadlocks. OpenStream is an intermediate language as will be sketched in Sections 1.5 and 2.1: some sequential order, explicit in the program, is used to define an implicit order between writes and reads, in a data-flow manner similar to Kahn process networks. It is single assignment, deterministic, but with an implicit partial order of operations that may induce deadlocks.

The second concept to define is what are the memory cells "accessed" by a given operation. And what is an operation? This discussion may lead to many important problems such as pointer analysis, pure or non-pure functions, inter-procedural analysis, approximations, data races, control dependences, but even in the simpler case where an operation is an atomic modification of a well-identified memory structure, a key point to define dependences (the relation δ) is to be able to define precisely what is an operation, what is the order of these operations, what memory cells are concerned, and if multiple writes on shared variables (possibly inducing race conditions) are possible. In the case of OpenStream, data races are not possible as the language has the single assignment property (each memory cell is well-identified as a stream element, and is written only once). However identifying which memory cell is addressed by a given operation is complex (see Section 2.3), which makes dependence analysis hard, if not impossible.

1.3 X10 and the `async / finish` Languages

X10 [31] is a parallel programming language, developed at IBM Research (Yorktown Heights), in the context of an effort to increase programmer's productivity, sponsored by the US department of energy (DoE).

1.3.1 The Base Language

X10 is an object oriented language of the Java family. It has classes and methods, assignments and method invocation, and the usual control constructs: conditionals and loops. Dealing with method invocation necessitates inter-procedural analysis and is beyond the scope of this report. X10 has two kind of loops: the ordinary Java loop

```
for(initialization; tests; increment)
```

and an enumerator loop

```
for(x in range)
```

where the type of the counter `x` is inferred from the range type. A loop is polyhedral if `x` is of integer type and if the range is an integer interval. The bounds of the range must be affine functions of surrounding loops counters and integer parameters.

1.3.2 Concurrency

Concurrency is expressed in X10 by two constructs, `async S` and `finish S`, where `S` is an arbitrary statement or statement block. Such constructs can be arbitrarily nested, except that the whole program is always embedded in an implicit or explicit `finish`. 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 which were created inside `S` have terminated.

1.3.3 Synchronization

In some cases, it may be necessary to synchronize several parallel activities. This can be achieved using *clocks*. Clocks are created by `clocked finish` constructs. Activities are *registered* to the clock associated to the innermost enclosing `clocked finish` if created by a `clocked async` construct. An activity deregisters itself from its associated clock when it terminates. Synchronization occurs when an activity execute the `advance` primitive. This activity is held until all registered activities have executed an `advance`, at which time all registered activities are released.

Clocks can be seen as generalization of the classical barriers. The main differences are that activities may be distributed among several clocks which work independently, and that this distribution is dynamic as it can change when an activity is created or terminated.

1.3.4 Evaluation

X10 has a well developed open source compiler, which is freely available from IBM. X10 programs belong to the class of deadlock free programs. It is easy to see that `async` and `finish` cannot create a deadlock *by themselves*. There is always the possibility that an elementary statement, e.g., a function call, does not terminate. However, an X10 program may have races, for instance, when a memory cell may be written by several unordered operations. Races may be detected at run time, or, in favorable cases, at compile time [37].

1.4 Communicating Regular Processes

Modular compilation is a very well-known technique, which dates back to the early days of Fortran. For most sequential languages, the module is the function. In fact, provided one has designed a clever calling interface, functions can be compiled independently of each others [1]. Nevertheless, the compiler output is not an executable program. One needs another tool, the *linker*, whose goal is mainly to plug the addresses of the called functions into the callee code. Compilation is modular, but linking is not.

[1] In modern languages, the need for accurate type checking induces more complex relations between modules.

In the case of parallel programming, functions are not suitable as modules. If functions are handled as black boxes, then one may lose many opportunities for parallelization. If one opens the box, then modularity disappears. There is, however, another possibility: processes and network of processes. Process networks abstract from the behavior of message-passing architectures: each process sits in a processor and has its own private memory. Processes communicate only by sending messages over ports and through channels. Message passing libraries and languages abound. Libraries range from the basic socket system of Unix to MPI and BSP. Such systems have almost no restrictions on what the programmer can do and may suffer from non-deterministic behavior and deadlocks. The analysis and debugging of programs written using these libraries is very difficult.

Kahn Process Networks (KPN) [22] are an attempt to impose determinism by construction: channels are perfect FIFO queues, and each channel has only one reader and one writer. The static analysis of KPN is still difficult, because send and receive operations can only be correlated by counting messages, which may lead to non-linear counting functions and may even be impossible in the presence of conditionals.

In CRP (communicating regular processes), proposed in [12], processes are modules, but the semantics of channels is modified. A channel is an array of arbitrary dimension, which is used in write once/read many mode. This constraint is enough to insure determinism (the proof is rather technical and has been sketched in [13]). Read and write operations are now correlated by comparing array subscripts. To insure the possibility of precise analysis, subscripts must be affine functions of surrounding loop counters, i.e., the processes must be static control programs in the sense of [10].

1.4.1 Program Structure, Syntax, and Semantics

An application is a collection of function and process definitions. Several definitions can be collected in a module (usually a file); an application can be composed of an arbitrary number of modules. Like in C, process and function definitions are top level objects and do not nest. The basis of the syntax of CRP is ANSI C. There are, however, a few new keywords: `process, inport, outport, channel`. All these are reserved and are considered as additional "storage class specifiers" in the C grammar.

Arrays and other Data Structures There is an array constructor, `[]`, with the same properties as in C. The number of dimensions is static (number of `[]`), however the rules for sizes are much more permissive than in C. In fact, in many cases, the compiler infers the size of the array from the way it is used. Similarly, there is a structure constructor, with the same syntax and properties as in C. In the present version of the compiler, pointers are ignored.

Functions User-defined functions are inlined. Hence, recursion is forbidden. One may use black box functions, which are handled by the system as if they were pure (no modifications of the actual parameters or of global variables).

1.4.2 Processes, Channels, and Ports

Processes A *process* is a sequential program which can communicate with other processes through *channels* (see hereafter). All variables are local to one and only one process and are not visible from other processes[2].

Besides operative statements, a process can include process start statements, which have the same syntax as a void function call. Process start statements are not part of the control flow of the surrounding process. In effect, all the process start statements in an application are collected and executed immediately at application start time. One can define a process start graph, which must be a DAG.

The operating code of a process must be *regular*, or have static control [11] in the following sense:

- Statements are assignments statements and regular loop statements. All variables are considered part of some array, scalars being one-dimensional arrays of size 1.
- The only method of address calculation is subscripting into arrays of arbitrary dimension. The subscripts must be affine forms in constants and surrounding loop counters.

Some of these restrictions are quite natural when designing compute-intensive embedded systems with real time constraints. It is difficult, for instance, to predict the execution time of a `while` loop or of the traversal of a truly dynamic data structure. Other restrictions can be lifted by preprocessing (`goto` removal, inductive variable detection, subscript-like pointer detection, function inlining).

Channels A *channel* is the only medium of communication between processes. It can be viewed as a write-once/read-many multi-dimensional array of indefinite sizes. Each cell has a (virtual) full/empty bit. At application start time, all such bits are set to "empty".

- A write to an empty cell defines its value and sets the control bit to "full".
- A write to a full cell generates an error.
- A read of an empty cell stalls the reading process until the cell is filled.
- A read of a full cell is immediately satisfied.

There is no way of emptying a cell.

A channel may have any number of readers, and there are no constraints on the reading patterns. Reading is not destructive: a value remains available at least as long as some process may have some use for it.

[2]The model accepts read-only global variables (e.g., tables of constants). This facility is not discussed here for brevity sake.

Ports A port is an interface between a process and a channel. It allows, *inter alia*, that a process be instantiated several times, each instance being connected to different channels. Ports are only allowed as formal parameters to processes.

When connecting ports and channels, one must verify (statically) that the two entities have the same (data) type and dimension, that a channel has only one writer in single assignment mode, and that readers do not access undefined cells (*holes*) in a channel. Channels play the role of actual parameters to the port formal parameters.

The usual rule of visibility applies to ports and channels. If `P` is a process in which a channel `c` is defined, the only processes that can access `c` are `P` itself and processes which are started by `P` and have a port connected to `c`.

1.4.3 Evaluation

Within these restrictions, the CRP compiler, Syntol, is able to do a very precise analysis of a CRP application. It can compute dependences, construct a runtime schedule (when it finds one), and impose a maximum size on channels. Incidentally, the existence of a schedule is a proof that the application has no deadlock. Furthermore, all these analysis can mostly be done process by process, thus greatly improving the compilation time. The price to pay is that many programs do not fit into the CRP model. Researches are under way to enlarge the applicability of the CRP model, either by better preprocessing or by approximation methods.

1.5 OpenStream

The design of OpenStream builds on a previous streaming data-flow extension [27] to OpenMP. For a more detailed presentation, one may refer to [30], and to the formal model underlying the operational semantics of OpenStream [28].

1.5.1 The Base Language

In a nutshell, OpenStream allows the composition of tasks communicating through data-flow streams, as well as separate compilation. It also provides more general dynamic constructs to support complex data structures and unbounded fan-in and fan-out communications. Streams are strongly typed, and first-class. In particular, they may be freely combined with recursive computations and dynamic data structures, while preserving modular (separate) compilation. Variadic stream clauses allow to construct arbitrarily complex, dynamic, possibly nested task graphs.

It has been shown that OpenStream is sufficiently expressive to efficiently encode high-level parallel language features such as the memory regions of StarSs [26], as well as low-level point-to-point communication primitives such as futures [29, 30]. OpenStream also provides syntactic support for broadcast operations.

1.5.2 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 control flow regions are called *tasks* and inherit the OpenMP task

syntax. Without stream annotations, OpenStream tasks also have the same semantics as OpenMP tasks.

Unlike OpenMP, OpenStream allows to express the data flow between OpenMP tasks and to build a task dependence graph. Task graphs need be neither regular nor static, unlike the majority of the streaming languages. 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.

Despite its expressiveness, the programming model comes with specific conditions under which the functional determinism of Kahn networks [22] is guaranteed by construction. These conditions enforce a precise interleaving of data in streams derived from the control flow of the program responsible for spawning tasks dynamically, hereafter called the *control program*. One simple sufficient condition for determinism is that the control program is sequential. More general conditions exist, based on the partitioning of streams occurring in concurrent tasks of the control program [28], or based on the total ordering of concurrent control program tasks spawning tasks that operate on the same stream. The formal characterization of these more general conditions remains work in progress.

1.5.3 Synchronization

The syntactic extension to the OpenMP3.0 language specification consists in two additional clauses for the `task` construct: the `input` and `output` clauses presented on Figure 1. We provide an informal description of the programming model; see [28] for a formal, trace-based operational semantics.

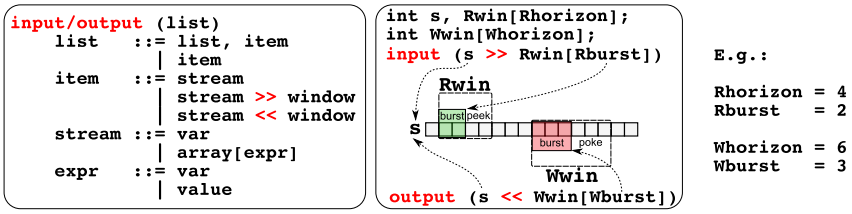

Figure 1: Syntax for `input` and `output` clauses (left) and illustration of stream access through windows (right).

Both clauses take a list of items, each describing a stream and its behavior with regard to the task to which the clause applies. If the item notation is in the abbreviated form `stream`, then the stream can only be accessed one element at a time through the same variable `stream`. In the second form, `stream >> window`, the programmer uses the C++-flavored `<< >>` stream operators to connect a sliding window to a stream, gaining access to multiple stream elements, within the body of the task.

Tasks compute on streams of values and not on individual values. To the programmer, streams are simple C scalars, transparently expanded into streams by the compiler. An array declaration (in plain C) defines the sliding window accessible within the task and its size, the *horizon*. The connection of a sliding window to a stream in an `input` or `output` clause allows to specify the *burst*, which is the number of elements by which the

```
#pragma omp task output (x)                          // Task T1
  x = ...;
for (i = 0; i < N; ++i) {
  int window_a[2], window_b[3];

  #pragma omp task output (x « window_a[2])          // Task T2
    window_a[0] = ...; window_a[1] = ...;
  if (i % 2) {
    #pragma omp task input (x » window_b[2])         // Task T3
      use (window_b[0], window_b[1]);
  }
  #pragma omp task input (x)                         // Task T4
    use (x);
}
```

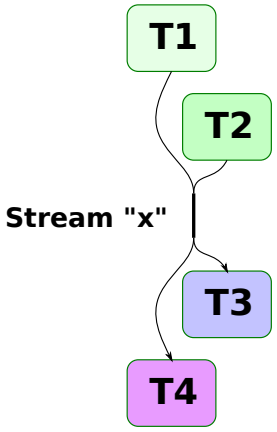

Figure 2: Example of `input`/`output` clause uses (left) and the resulting task graph (right).

sliding window is shifted after each *task activation* (see Section 1.6). In Figure 1, the input window `Rwin` would be shifted by two elements, while the output window `Wwin` would be shifted by three elements. The data-flow case corresponds to *horizon* = *burst*. In the more general case where *horizon* > *burst*, the window elements beyond the burst are accessible to the task; for an output window, 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, and is driven by the control flow of the main OpenMP program (see Section 2.3 for a more formal description of dependences between tasks).

The example in Figure 2 illustrates the syntax of the `input` and `output` clauses. Task T1 uses the abbreviated syntax to produce one data element for stream `x`. The semantics of stream operations is to interleave accesses, as illustrated on Figure 3, in task creation order. This order is determined by the flow of control spawning tasks, called *control program*. In our example, T1 introduces a delay in stream `x`. Task T2 is also a producer, adding two elements to stream `x` at each activation. Tasks can be guarded by arbitrary control flow, as is the case for T3, which reads three elements at a time and discards two elements. T4 also reads from `x`, interleaving its accesses to the stream with the accesses from T3. This interleaving is entirely determined by the schedule of the control program, in this case, it is a sequence (T4, T3, T4, T4, T3, ...).

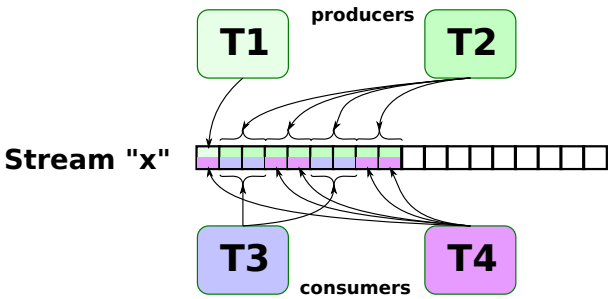

Figure 3: Interleaving of stream accesses for the tasks on Figure 2.

1.5.4 Evaluation

Higher expressiveness may improve productivity, but it often comes with performance overheads, impacting the compiler optimizations and increasing the complexity of the necessary runtime support. However, it is also an important asset: general point-to-point synchronizations alleviate the risk of overconstraining the execution: for example, task-level pipelines can hardly be expressed in simpler programming models like Cilk. Performance evaluation backing up this intuition can be found in [30].

The official repository and web site for OpenStream (`http://www.openstream.info`) centralizes the information for embedded and high-performance computing scenarios. The sources of the GCC-based OpenStream compiler are available with the command:

```
$ git clone git://git.code.sf.net/p/open-stream/code open-stream-code
```

on a Sourceforge GIT repository. Compiling GCC requires additional libraries, but the process is fully automated with regression testing and platform-specific auto-tuning of some of the runtime parameters. This version supports x86_32, x86_64 and ARM v7 instruction sets. The prototype MPPA (single-cluster) version is available on demand.

1.6 Other Data-Flow and Streaming Languages

The principal motivation for research into data-flow models comes from the incapacity of the von Neumann architecture to exploit large amounts of parallelism, and to do so efficiently in terms of hardware complexity and power consumption. The early data-flow architectures [7, 6, 36] avoid the von Neumann bottlenecks by only relying on local memory and replacing the global program counter by a purely data-driven execution model, executing instructions as soon as their operands become available. Programmer productivity is another important motivation: debugging concurrent applications with low-level threads is a daunting task, mostly because of the non-deterministic nature of races and deadlock-related errors. Data-flow languages closely follow the hardware model: the execution is explicitly driven by data dependences rather than control flow [21]. Data-flow languages offer functional and parallel composition of parallel programs preserving functional determinism. Recent data-flow architectures, execution models, and languages rely on the same principles, albeit at a coarser grain, executing sequences of instructions, or data-flow threads, instead of single instructions.

Among the most notable data-flow languages, Lucid [2] relies on the `next` keyword within loops to achieve a similar effect to advancing in a stream of data, by consuming or producing in a channel, or, in the synchronous languages domain, to the advancement of clocks on signals. Sisal [17] explicitly introduces the notion of stream, which is naturally very close to lists. If stream processing systems are understood as the parallel implementation of stream transformers, which is the functional interpretation of a process network mapping a set of input streams to a set of output streams, then any functional language can be used for stream programming. This corresponds to the lazy interpretation of functional languages; see [5] for a Haskell [20] implementation of Lucid Synchrone [4]. First-class streams of data improve expressiveness for a variety of communication and concurrency patterns such as broadcast, delays, and sliding windows. This was observed by data-flow computing pioneers, who designed I-structures as unbounded streams of futures to alleviate some of the overheads of a pure data-flow execution model [1].

As was already discussed, Kahn process networks (KPN) [22] form the basis for most deterministic languages based on stream computing concepts. In his survey [33] of stream processing, 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 a restricted form of Kahn process networks: static data-flow (SDF) and cyclo-static data-flow (CSDF) [24, 3]. 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 [16] 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 [18] for a restricted set of benchmarks that properly map on this model. On the other hand, data-flow synchronous languages such as Lustre [19] have been widely adopted in the certified design flow of reactive control applications. They offer determinism, deadlock freedom, bounded reaction time and memory. Unlike CSDF, they are not restricted to periodic activations and communications. Processes respond instantly and communicate through signals, also used to define a notion of time and causality. Signals differ from streams in that they are sampled rather than consumed.

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 not flawed, but data flow often needs to be predicated by complex control flow due to irregular events, as is the case in synchronous programs. Importantly, new approaches to streaming should try to preserve the strong properties provided by some of the existing models, like functional determinism or deadlock-freedom.

Outside the field of streaming languages, a rich set of constructs has emerged to express inter-task dependences in parallel languages. StarSs [26] is a pragma-based language to program distributed-memory and heterogeneous architectures; it supports both data-flow and control-flow programming styles, deriving inter-task dependences from the read and write accesses to array regions. SMPSs is one of the StarSs incarnations for shared-memory targets [25]. Unlike most alternative constructs, this choice of a control-flow-induced, data-centric definition of inter-tasks dependences guarantees the absence of deadlock. One of the downsides is that pipelining and data parallel execution require the programmer to explicitly expand/privatize the arrays used for synchronization. The

cost of dynamic dependence resolution is also slightly increased due to the management trees of array regions and matching task signatures within those trees. Independently, Habanero Java [32] introduced phasers, generalizing barriers and point-to-point synchronizations. Phasers ease the expression of complex inter-task dependences, but do not guarantee the absence of deadlock. Interestingly, and unlike most streaming languages, these parallel languages also support dynamic task creation. OpenStream was introduced in an attempt to reconcile some of the strengths of both streaming and dynamic task-parallel languages.

2 Dependence and Dataflow Analysis

The rest of this report deals specifically with the OpenStream language as defined in [30] and introduced in Section 1.5. Another source is the CDDF research report [28].

Any attempt to apply instance-wise analysis methods to OpenStream must start by identifying the key concepts: the set of statement instances or *operations*, and their execution order or *happens before* relation. For an example of such a characterization applied to the X10 language, see [37] as well as the summary given in Section 2.2.

OpenStream and CDDF are two-levels languages: task creation, then task activation/execution. A sequential control program direct the *creation* of tasks. Each created task waits until its *activation*, which means that all tasks it *depends on* (see hereafter) have terminated execution: it can now start its *execution* as soon as it is selected by the runtime scheduler. Both the control program and the task's code are written in a classical sequential language, C in this case. There are no constraints in the amount of work done either by the control program or the tasks. Communication between tasks is done through *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 integers, the horizon – the size of the window – and the burst – the amount by which the window is shifted right at each task creation. These numbers may be arbitrary data-dependent expressions. However, it does not seem possible to analyze OpenStream programs unless they can be expressed (in the code or after some analysis) as numerical, symbolic constants, or polynomial expressions (in this case, this relates to the concept of weighted sum in the integer set library ISL [34]).

The syntax of the language allows for arrays of streams, not to be confused with multi-dimensional streams, i.e., these arrays are similar to arrays of pointers, each pointer corresponding to a (one-dimensional) stream, and there are no windows associated to the array dimensions themselves. At the time of creation, tasks have access to all variables of the control program which are in 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. These constructs are inherited from OpenMP.

In this report, 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 `if` constructs, for example.) As a side effect, one obtains the creation order of task instances. For each stream and each task instance, one may compute a read or write index. One can then compute dependences between tasks. Since

```
#pragma omp task output(x)                    //T1
  x = ...;
for(i = 0; i<n; i++){
  int window_a[2], window_b[3];

  #pragma omp task output(x << window_a[2]) //T2
    window_a[0] = ...; window_a[1] = ...;

  if(i % 2){
    #pragma omp task input(x >> window_b[2])//T3
      use(window_b);
  }
  #pragma omp task input(x)                   //T4
    use(x);
}
```

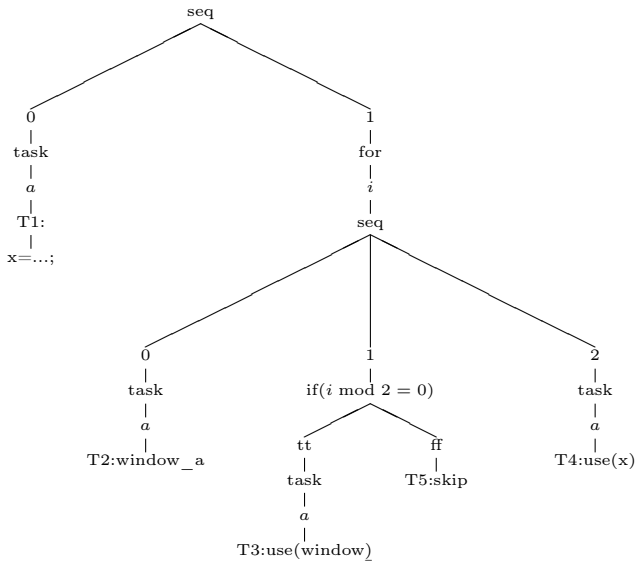

Figure 4: Program example (see Fig. 2) and corresponding AST.

streams are accessed in single assignment mode by construction, there are only read-after-write (RAW) or producer-consumer (PC) 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.

And now for the details.

2.1 Execution Order

2.1.1 The Abstract Syntax Tree

The AST has five kind of nodes:

- Sequence: `S1; ...; Sn` has n outgoing edges, labeled from 1 to n.
- Loop: `for(i=0; i<n; i++)` has one outgoing edge, labeled by i.
- Task: has one outgoing edge, labelled [3] by the letter a.
- If: has two outgoing edges, labelled tt and ff.
- Basic statement: has 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. As an example, consider the program on the left of Figure 4 (see also Section 1.5), which is an example borrowed from [30][Fig. 2]. Its AST is given on the right (statement labels added). The position vector of `S1` is $[0, a]$; the position vector of `S3` is $[1, i, 1, tt, a]$. The task that executes statement `S4` has position vector $[1, i, 2, a]$.

[3] for historical reasons, see Section 2.2, and for possible future extensions.

2.1.2 Orders of Tasks

A general OpenStream program, which allows for the creation of tasks within tasks (second-level tasks), has several execution levels:

- the creation order (a total order) of first-level tasks, which depends only on their position vectors,
- the activation order (a partial order) of first-level tasks, which depends on the dependences induced by stream accesses,
- the creation order of second-level tasks, which can be defined only if the execution order of first-level tasks is (at least partially) sequential, so that all tasks accessing a given stream are created sequentially,

and so on recursively. In this document, only first-level tasks are considered, i.e., no tasks can be created within tasks.

2.1.3 Creation Order

This is simply given by the lexicographic order of position vectors (excluding the terminal "a", which is not necessary here). For example, consider an instance of `S2`, $[1, i, 0, a]$, and an instance of `S4`, $[1, i', 2, a]$. The first one is created before the second one if and only if

$$1 < 1 \vee (1 = 1 \wedge i < i') \vee (1 = 1 \wedge i = i' \wedge 0 < 2) \vee ((1 = 1 \wedge i = i' \wedge 0 = 2) \equiv i \leq i'.$$

This creation order is denoted $\prec$ (strict order) and $\preceq$ (with equality) in what follows.

In this context, tests pose a difficulty. It is clear that the execution order of the opposite branches of a test is undefined. However, we want to use absence of order to represent parallelism. The trick is to associate to each elementary statement an iteration domain, and to stipulate that two position vectors can be compared only if their iteration domains intersect. This point is not elaborated in this document.

2.2 The Case of X10

The main tools for an instance-wise and element-wise dependence or dataflow analysis are a precise representation of the program operations and of their execution order or *happens-before* relation. X10 is simpler than OpenStream in that its execution order can be extracted from the program text (or its AST) in a straightforward manner, instead of being the result of a complex dependence calculation.

2.2.1 Paths and Iteration Vectors

Each statement *instance* is identified by a vector of integers, called an iteration vector. While in OpenStream it is enough to mark the creation of a task by the letter a, in X10 one has also to introduce another letter, f, for finish. The iteration vector is computed (symbolically) as a path from the root to a leaf in the AST. As the AST is traversed, values are appended to the vector based on the following rules:

- `Sequence`: Integer x when taking the x-th branch of a sequence.

- `For`: Loop iterator i.
- `Async`: a
- `Finish`: f

The iteration vector for a statement instance is obtained by instantiating the loop iterators to integer values. This is similar to the conventional iteration vectors used in the polyhedral literature, with the addition of a and f. Due to structural constraints, when two iteration vectors are compared, a and f are never compared to anything but themselves, and thus their order is irrelevant.

2.2.2 Happens-Before Relation

For sequential programs, the full lexicographical order denotes the execution order. In the presence of parallelism, the execution order is no longer total. Yuki et al. [37] showed that for the `finish/async` subset of X10 programs, the happens-before relationship can be expressed as an incomplete lexicographic order.

The strict lexicographic order of two distinct vectors u and v is defined as follows:

$$u \ll v \equiv \bigvee_{p \geq 0} u \ll_p v, \tag{1}$$

$$u \ll_p v \equiv \left(\bigwedge_{k=1}^{p} u_k = v_k \right) \wedge (u_{p+1} < v_{p+1}) \tag{2}$$

The incomplete version restricts the depths p that contribute to Equation (1) to some subset I. Intuitively, the set I is constructed such that the depths that do not contribute to the happens-before relation, due to concurrent execution, are removed. The relation $u \ll_p v$ is not considered if there is a "a" in u in a coordinate larger than coordinate $p+1$ and no "f" in between. The happens-before relation $\prec$ is then defined as follows:

$$u \prec v \equiv \bigcup_{k \in I} u \ll_k v \tag{3}$$

2.2.3 Race Detection through Dataflow Analysis

Using the iteration vectors and happens-before ordering, the authors of [37] developed an extension to the array dataflow analysis of [10] for X10 programs. Array dataflow analysis finds the statement instance that produced the value used by an instance of a read. Given reader and writer statements R and W, and memory access functions f_R and f_W, the set of potential sources is defined by:

$$r \in D_R, \tag{4}$$

$$w \in D_W, \tag{5}$$

$$f_W(w) = f_R(r), \tag{6}$$

$$\neg(r \prec w) \wedge r \neq w \tag{7}$$

where

- Constraints (4) and (5) restrict the statement instances to their domains (the sets of legal iterations),
- Constraint (6) restricts to those that access the same array *element*, and
- Constraint (7) excludes writers that happen after reads, and writes by the same statement instance.

In a sequential program, $\prec$ is total, hence $\neg(v \prec w) \wedge v \neq w$ is equivalent to $w \prec v$, which is the usual formulation of dataflow analysis as developed in [10].

The above gives a set of writer instances w that may be a producer for a read instance r for a single writer statement W. The proposed analysis proceeds by finding the most recent w among all statements that write to the same array. Since the happens-before relation is not total, the most recent w may not be unique, and there is a race when a producer cannot be uniquely identified.

2.2.4 Advance Counts

Informally, a clock may be considered to have as many associated counters as there are registered activities. When all participating activities execute an `advance`, synchronization takes place, and all counters are incremented. The statements that may-happen-in-parallel are restricted to those that are executed when the value of the counters match. In the case of a one-clock program, or inside an innermost `clocked finish`, this observation can be formalized as follows: let us write $\phi(x)$ for the number of `advance` that have happened before operation x, and let $\mathcal{A}$ be the set of `advance` instances in the program under study. Then:

$$\phi(x) = \mathrm{Card}\{a \in \mathcal{A} \mid a \prec x\}.$$

As a consequence, the happens-before relation for clocked programs, $\prec\!\!\prec$, must be completed as follows:

$$x \prec\!\!\prec y \equiv \phi(x) < \phi(y) \vee x \prec y,$$

in other words: x happens before y if fewer `advance` have been executed before x than before y, or when their `advance` counts are equal, if x would be executed before y in an unclocked program.

2.2.5 Counting Integer Points in Polytopes

For polyhedral iteration spaces, the question of counting advances can be cast as counting the number of integer points in polyhedra. Ehrhart [8] showed that the number of integer points in a polytope can be expressed as *periodic polynomials*. The work on X10 uses an algorithm proposed by Verdoolaege et al. [35] for computing such polynomials, which handles parametric polytopes.

2.2.6 Disproving Races

One may refine the dataflow analysis formulation developed for X10 programs without clocks (overviewed in Section 2.2.3) using the new happens-before relation for clocked programs. The only change required is to replace $\prec$ with $\prec\!\!\prec$. However, the fact that the ϕ functions are not affine in general creates a problem: parametric integer linear programming [9] can no longer be used, and there is no known alternative that can handle polynomials in integer variables.

Therefore, the proposed solution is to first detect races ignoring clocks and then later use constraint solvers to verify if the statement instances involved in a race can have equal ϕ values. One can prove that the problem is still undecidable: it is in fact equivalent to Hilbert's 10th problem, deciding whether a polynomials has integral roots (see [15]). However, modern SMT solvers, using heuristics and pattern matching approaches are able to solve most of the cases one encounters in practice.

2.3 The Case of OpenStream

2.3.1 Stream Indices

Let us write W_s (resp. R_s) for the set of tasks which have write access (resp. read access) to a stream s. To each instance t of a task τ and each stream s accessed by τ is associated a *burst* $b_{t,s}$. Each task instance has read or write access to a stream through a *window*. For a given task instance and stream, the position of the window (its index) is computed by the control program by summing the *bursts* of all preceding tasks instances.

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 always a nonnegative integer, and can only be null for an input stream [4] (the `peek` operation).

Let us write $I_s(t)$ for the index of output stream s at instance t of task τ. Let D_τ be the domain (set of instances) of task τ. We have the fundamental formula (see also [28]):

$$I_s(t) = \sum_{\tau \in W_s} b_{\tau,s} \mathrm{Card}\ \{x \in D_\tau | x \prec t\} \tag{8}$$

In the same way, if s is an input stream, the read index is given by:

$$J_s(t) = \sum_{\tau \in R_s} b_{\tau,s} \mathrm{Card}\ \{x \in D_\tau | x \prec t\} \tag{9}$$

Since for polyhedral programs D_τ is a polyhedron, and since $\prec$ is a disjunction of affine constraints, the necessary cardinal can be computed in closed form by familiar techniques and libraries. The result will usually be a polynomial, the degree of which is equal to the maximum 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:

[4] Output windows with a null burst/horizon are meaningless and would not even create a dependence.

Proposition 1 *If t and t' have write access to the same channel s, and if $t \prec t'$, then:*

$$I_s(t) + b_{\tau,s} \leq I_s(t'),$$

Proof Observe that the sets whose cardinals contribute to $I_s(t')$ are supersets of those contributing to $I_s(t)$, and that t belongs to the one but not to the other. ∎

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 (8) and (9) 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 like ISL [34].

2.3.2 Dependence Test

Two operations are in dependence if they both access the same memory location, and one of the accesses at least is a write. According to [30], the shared memory locations must belong to a stream. It would be possible to compute dependences on global variables, but this does not seem to be in the spirit of a streaming language.

As is well known, there are three kind of dependences:

- Producer-Consumer (PC) or RAW hazards or flow dependences,
- Producer-Producer (PP) or WAW hazards or output dependences,
- Consumer-Producer (CP) or WAR hazards or anti dependences.

The single assignment property implies that PP dependences do not exist. Also, there are no CP dependences as specified by the semantics of streams: by definition, writes always occur before reads.

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. Furthermore, if it was not satisfied on output windows, it would create holes in streams, which could never be filled by the single assignment property. Holes in input windows can only create spurious dependences and reduce parallelism. Another assumption is that each task has at most one window per stream. Otherwise, the semantics of this situation has to be defined, for example by considering that the windows sum up (sum of the bursts and horizons), or by considering the largest window (max of the bursts and horizons).

Let t be an instance of task τ that writes to stream s and 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:

$$I_s(t) \leq J_s(t') + h_{t',s} - 1 \wedge J_s(t') \leq I_s(t) + b_{t,s} - 1. \tag{10}$$

To these constraints, one must add conditions that express the fact that t and t' are legal iterations: $t \in D_\tau$ and $t' \in D_{\tau'}$. This condition enforces only that the writer of a given stream cell occurs before its readers. One can also impose a "Kahnian continuity" on streams (as in a FIFO), which means that a read can occur only if all stream cells with a smaller index have been written before. In this case, the condition becomes simply:

$$I_s(t) \leq J_s(t') + h_{t',s} - 1 \tag{11}$$

Indeed, this is equivalent to considering that the read window goes from 0 to $J_s(t') + h_{t',s} - 1$, thus the second inequality of Equation (10) is always satisfied.

Let us write $t\delta t'$ if this system of constraints is satisfied: δ is the *instance-wise dependence graph* of the subject program. The system $t\delta t'$ may be tested for satisfiability by any available tool. The result is a relation $\tau\Delta\tau'$, the *statement-wise dependence graph* of the subject program, where the dependence pair (τ, τ') is labelled by the set of instances that satisfy Equation (10) (or Equation (11) with the Kahnian continuity semantics), or an over-approximation of this set. The statement-wise dependence graph can be analyzed for parallelism. However, to apply an algorithm such as the Allen-Kennedy algorithm, some concept of depth has to be synthesized. Similarly, defining distance vectors may be interesting if the iteration domains of writes and reads are the same.

If the index functions are linear, i.e., if the control program has no deeply nested loops, a linear programming tool may be enough. If not, the use of an SMT solver like Z3, which can handle polynomials will be necessary. 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 t (the writer) must be executed before t' (the reader). 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, even in the absence of barriers.

3 Verification Problems in OpenStream

3.1 Deadlocks

The easiest method for proving the absence of deadlocks consists in building a schedule. If the I_s and J_s functions are linear, this can be done using standard algorithms like the Farkas algorithm. In the presence of polynomial functions, it may be that the special form of Equation (10) may simplify the construction of a schedule. It is not clear however if an equivalent of Farkas lemma (useful for generating schedules in the polyhedral model) for polynomial constraints exists, but one never knows.

The other possibility is to observe that there is a deadlock if one can find a cycle $t_0, t_1, \ldots, t_{n-1}$ such that $t_i\delta t_{i+1}$ and $t_{n-1}\delta t_0$. (This condition is necessary and sufficient if the number of operations is finite, or with the Kahnian continuity semantics, see Section 3.1.1.) Furthermore, the corresponding statements $\tau_0, \ldots, \tau_{n-1}$, form a cycle in the statement-wise dependence graph. There is a deadlock if the conjunction of the dependence relations is satisfiable. However, the number of such cycles (between instances) is

potentially unbounded except in trivial cases, hence this is only a semi-algorithm. Here again, the special form of δ may simplify the solution.

These two approaches are left for future work. We now give two preliminary results that characterize how difficult deadlock detection can be in the general case.

3.1.1 Deadlocks and Cycles between Task Instances

Consider the following example in an OpenStream-like format:

```
s, t streams;
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)` because of stream `t` and `b(i)` depends also on `a(i-1)` because of stream `s` as it depends "functionally" on `a(i)`. The following proposition explicits these situations in general.

Proposition 2 *If the control program generates a finite number of task instances, there is a deadlock if and only the graph of dependences among task instances has a cycle. The same is true for an infinite number of instances and the Kahnian continuity semantics. Without the Kahnian continuity semantics, if the number of task instances is infinite, there is a deadlock if and only if there is a task T and an infinite sequence of position vectors $(i_j)_{j\in\mathbb{N}}$ such that $i_j \ll i_{j+1}$ such that $T(i_j)$ depends on $T(i_{j+1})$, i.e., an infinite number of instances of the same task that depend on each other in the inverse order of their creation.*

> **Proof** Here is a sketch of the proof. A schedule is a function σ that assigns to each task instance t a nonnegative integer $\sigma(t) \in \mathbb{N}$ such that if a task instance t depends on a task instance u, then $\sigma(t) > \sigma(u)$, i.e., $\sigma(t) \geq \sigma(u) + 1$. This corresponds to assuming that each task has duration one (real durations can be arbitrarily large but tasks are assumed to have no deadlock internally, thus to execute in finite time and to take at least one unit of time [5]). As task creation does not depend on tasks dependences, there is no deadlock due to task creation, we only need to consider when tasks start executing. Using proof techniques similar to those used for systems of uniform recurrence equations [23], we can show the following properties.

[5] Tasks of duration 0 or infinitely small would allow for an infinite number of activations without implying deadlocks, which is not a reasonable model to work with.

- There is no schedule if and only if there is a task instance t such that the length $l(t)$ of a dependence path leading to t is not bounded.
- Each task instance depends (directly) on a finite number of other task instances. This is because a task instance reads only a finite number of streams and because bursts and horizons are numbers (thus finite). This is true also in the Kahnian 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 t and an infinite dependence path leading to t.
- The previous property shows that there is a deadlock if and only if there is no schedule. Indeed, if there is no deadlock, there exists an execution of tasks. This execution produces a schedule (considering that the smallest task duration is 1). Conversely, if there is a schedule, it is not enough to say that this schedule can be stretched to take the real duration into account as, given a task instance, there can be an infinite number of task instances scheduled before and thus the maximum of their duration can be infinite. However, one can build another schedule, called the free schedule, that executes each task instance as early as possible (this is without resource constraints): this is possible as the length of each path leading to it is finite and thus its total duration is finite too.
- Consider an infinite path leading to t, i.e., a sequence of task instances $(t_i)_{i\in\mathbb{N}}$ such that $t_0 = t$ and t_i depends on t_{i+1}. From $(t_i)_{i\in\mathbb{N}}$, one can extract an infinite subsequence $(t_{i_j})_{j\in\mathbb{N}}$ of task instances (which are thus in dependence by transitivity) that is nondecreasing with respect to the order of task creation $\preceq$, i.e., $i_j \preceq i_{j'}$ if $j \leq j'$. As there is a finite number of tasks, one can even select only instances of the same task T. In other words, if there is a deadlock, there is a task T and a sequence of position vectors $(i_j)_{j\in\mathbb{N}}$ such that if $j \leq j'$, $T(j)$ depends on $T(j')$ and $j \ll j'$ (lexicographic order). The converse is obviously true.

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 see what happens with the Kahnian continuity semantics with an infinite number of task instances. It is easy to see that given two tasks T and U, and two position vectors i and j, such that $U(j)$ depends (directly or by transitivity) on $T(i)$, then $U(j)$ also depends on $T(i')$ for all $i' \ll i$ if all task instances always write in the same streams (i.e., if $T(i)$ writes in `s`, so does $T(i')$). If there is a deadlock, there exist a task T and two position vectors i and j such that $i \ll j$ and $T(i)$ depends on $T(j)$, thus $T(i)$ also depends on $T(i)$ since $i \ll j$, which forms a cycle. The converse is of course true. ∎

3.1.2 Detecting Deadlocks is Undecidable

The following construction 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;

- if an OpenStream program has a spurious or a functional deadlock;
- if an OpenStream program has a stream causal schedule.

A functional deadlock is, as explained earlier, a situation where no created task can be executed, assuming the general semantics of dependences given by Equation (10). 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 related to race conditions in X10 presented by Paul Feautrier during the April 2013 french compilation days [15]. P and Q are two multivariate polynomials (with n variables). The code can use only horizons and bursts equal to 1 if large loop bodies are allowed, otherwise horizons and bursts can be used to emulate coefficients of the polynomials.

```
s, t streams;

for (x in D) {  /* D is the n-dimensional cube of size N in the first orthant */

  R1: read Q(x)-1 times in t;
  W1: write P(x)-1 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 the construction of [15], it is always possible to write affine loops so that R1 reads $Q(x) - 1$ 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 a dependence from $S(x)$ to $T(x)$. Concerning stream `t`, $T(x)$ writes in position $P(x)$ (without counting all previous iterations of x) and $S(x)$ reads in position $Q(x)$.

If $P(x) = Q(x)$, there is a functional dependence from $T(x)$ to $S(x)$. Otherwise, there exists a schedule for iteration x: execute $W1(x)$ and $W2(x)$, then $S(x)$, then $T(x)$, and finally $R1(x)$ and $R2(x)$. Thus, there exists a value of N with a deadlock if and only if there exists a (component-wise) nonnegative vector x such that $P(x) = Q(x)$ (undecidable, see [15], variant of Hilbert's 10th problem). This is for functional deadlocks.

For spurious deadlocks, i.e., with the Kahnian continuity semantics, there is a dependence from $S(x)$ to $T(x')$ for all $x \preceq x'$ because of stream `s`. For stream `t`, if $P(x) = Q(x)$ there is a functional deadlock and if $P(x) < Q(x)$ then the task instance writing in $Q(x)$ is one of the instances of $W2(x)$. But there is still a dependence from $T(x)$ to $S(x)$ since $T(x)$ writes in a smaller index than $W2(x)$, thus there is a deadlock. Finally, if

$P(x) > Q(x)$, there is a schedule and even a causal schedule: execute $W1(x)$, then $R1(x)$, then $S(x)$, then $T(x)$, then $W2(x)$, and finally $R2(x)$. In conclusion, there is a deadlock if and only if there exists a component-wise non-negative x such that $P(x) \leq Q(x)$. This leads to another variant of Hilbert's 10th problem: is there a component-wise nonnegative x such that $P(x) \leq 0$? If this problem was decidable, then one could also decide if $Q(x) = 0$ for a given polynomial Q. Indeed, $Q(x) = 0$ if and only if $Q^2(x) = 0$, which is also equivalent to $Q^2(x) \leq 0$.

3.2 Dataflow Analysis

Dataflow analysis for OpenStream is both trivial and impossible in general. The problem is, given a cell i in stream s, to find the position vector of the task that wrote $s[i]$. This vector and the associated task must satisfy the constraints:

$$I_s(t) \leq i \leq I_s(t) + b_{t,s} - 1 \ , \ t \in D_\tau.$$

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.

3.2.1 Looking for Some Solvable Cases

To try to better understand the function I_s and J_s and how to compute them, let us give a few remarks. By construction, the functions $I_s(t)$ and $J_s(t)$ have some hidden lexicographic properties, as stated in Proposition 1. Consider $I_s(t)$. Given a scalar k, there exists t_k such that the set of vectors t for which $I_s(t) \leq k$ has the following form $\{t \mid I_s(t) \leq k\} = \{t \mid t \preceq t_k\}$. Similarly, there exists a task instance t'_k such that $\{t' \mid J_s(t') \geq k\} = \{t' \mid t'_k \preceq t'\}$. Then, assuming $h_{t',s} = 1$ to simplify the notations, Equation (11) can be rewritten into:

$$\exists k \text{ s.t. } t \preceq t_k,\ t'_k \preceq t',\ t \in D_\tau,\ t' \in D_{\tau'} \tag{12}$$

The minimal dependence distance (when τ and τ' have same iteration domain) is the minimum of $t'_k - t_k$. The challenge is now to be able to compute t_k, i.e., more or less to inverse the function I_s (same for J_s). It also becomes clear from Equation (12) that a conservative approximation consists in building t_k (resp. t'_k) larger (resp. smaller), with respect to $\prec$, than the exact vector.

Property 1 also implies that t_k and t'_k can be defined one dimension at a time for a given domain D_τ. Indeed, if $t_k = (i_k, j_k)$ (in dimension 2 to simplify notations), then $t_k = \max_{\preceq}\{t \mid I_s(t) \leq k\}$, thus:

$$i_k = \max\{i \mid (i,j) \in D_\tau,\ I_s(i,j) \leq k\},\ j_k = \max\{j \mid (i_k,k) \in D_\tau,\ I_s(i_k,j) \leq k\}$$

Actually, for the computation of i_k, one can replace j by $f(i)$ where $f(i)$ is a function of i such that $(i,j) \in D_\tau$. This way, I_s is now a function of i only. If $f(i)$ is the smallest possible j (for example $j = 0$ on a rectangular domain starting at 0) then

$\max\{i \mid (i, f(i)) \in D_\tau,\ I_s(i, f(i)) \leq k\}$ is equal to i_k. In the general case, this maximum can be equal to i_k or $i_k - 1$. This variable elimination may make the computations simpler to solve. In some simple (and hopefully practical) cases, these computations can even be carried out in an affine way, and the dependences, thanks to Equation (12), can be computed in an affine way too. This is left for future work.

3.2.2 Impact of Barriers

There is not much about barriers in [30], but they certainly exist in OpenMP, and are a prominent feature in [28]. 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 functions I_s and J_s, and hence does not change stream dependences. However, it adds new pseudo-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 one must add 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.

4 Conclusion

This preliminary study of the OpenStream language has shown that, even if there is hope in solving more general but specific cases, two conditions must be satisfied to apply the full polyhedral model:

- The control program must conform to the polyhedral model: DO loops with affine bounds in outer loop counters and size constants.
- The index functions as defined by equations (8) and (9) must be affine; this in turn implies that bursts and horizons must be known numbers, and that the program has only one-dimensional loops.

If these conditions are satisfied, all polyhedral tools can be applied, including scheduling and hence efficient proof of deadlock absence (proving the existence of deadlocks seems much harder) and program transformations. The knowledge of a static schedule may help the compiler in the generation of efficient parallel code. However, it seems probable that the actual runtime scheduler will keep some dynamic features, if only to adapt to variations in the execution time of tasks. A knowledge of the properties of this scheduler will be necessary for stream sizing. However, Proposition 2 shows, as a by-product, that if streams are bounded statically and if a schedule still exists, then a dynamic schedule will not deadlock, i.e., will not end with a situation where no ready task can be executed.

To analyze programs beyond this very restricted set of polyhedral OpenStream programs, one may follow two directions: one may use affine approximations, or one may try to take advantage of the special properties of the I and J functions and invent new analysis algorithms, for example as suggested in Section 3.2.1. As an example of what one can do with approximations, dependences can be conservatively approximated by overestimating the J function and underestimating the I function. If these approximations

are affine, the polyhedral model can be applied. Finding affine over- and under-estimates is probably easy for bounded programs, but may be difficult or even impossible in the presence of infinite loops, as found in signal processing applications. Another approach is to observe that the I and J functions are not arbitrary polynomials. Their main property is that they are monotone increasing with respect to the lexicographic order of task activations, and hence invertible. However, finding a closed form inverse might be difficult. Another interesting property is that the counter of the innermost surrounding loop will always occur linearly in the task I and J function. How to exploit these properties, for instance to devise a scheduling algorithm, is an interesting research subject.

Transforming an OpenStream program in order to augment its granularity or to improve its locality is another interesting subject. Since OpenStream tasks are best considered as black boxes, the transformation must apply only to the control program. One possible approach would be to find conditions under which the I and J functions of the transformed program stay the same.

Lastly, one may wonder if the analysis techniques proposed in this report may be extended to more advanced features of OpenStream, like second-level tasks, variadic streams, or conditionals.

References

[1] Arvind, Rishiyur S. Nikhil, and Keshav Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[2] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Communications of the ACM*, 20(7):519–526, 1977.

[3] G. Bilsen, M. Engels, Lauwereins 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.

[4] Paul Caspi, Grégoire Hamon, and Marc Pouzet. Synchronous functional programming with Lucid Synchrone. In Stephan Merz and Nicolas Navet, editors, *Modeling and Verification of Real-Time Systems: Formalisms and Software Tools*, chapter 7, pages 207–247. ISTE, 2007.

[5] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 226–238, Philadelphia, Pennsylvania, United States, 1996. ACM.

[6] A. L. Davis. The architecture and system method of DDM1: A recursively structured data driven machine. In *5th Annual Symposium on Computer Architecture (ISCA'78)*, pages 210–215, New York, NY, USA, 1978. ACM.

[7] Jack B. Dennis and David Misunas. A preliminary architecture for a basic data flow processor. In *2nd Annual Symposium on Computer Architecture (ISCA'74)*, pages 126–132, 1974.

[8] Eugène Ehrhart. Sur les polyèdres rationnels homothétiques à n dimensions. *Comptes rendus de l'Académie des Sciences, Paris*, 254:616–618, 1962.

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

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

[11] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

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

[13] Paul Feautrier. Bernstein's conditions. In David Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[14] Paul Feautrier. Dependences. In David Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[15] Paul Feautrier. Array dataflow analysis for polyhedral X10 programs. In *French Compilation Days*, Annecy, April 2013.

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

[17] Jean-Luc Gaudiot, Tom DeBoni, John Feo, Wim Böhm, Walid Najjar, and Patrick Miller. The Sisal model of functional programming and its implementation. In *2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis (PAS'97)*, pages 112–123, Washington, DC, USA, 1997. IEEE Computer Society.

[18] Michael Gordon, William Thies, and Saman 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.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[20] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 27:1–164, May 1992.

[21] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, March 2004.

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

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

[24] 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.

[25] Vladimir Marjanovic, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Effective communication and computation overlap with hybrid MPI/SMPSs. In *15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, pages 337–338, Bangalore, India, 2010.

[26] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical task-based programming with StarSs. *International Journal on High Performance Computing Architecture*, 23(3):284–299, 2009.

[27] Antoniu Pop and Albert 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.

[28] Antoniu Pop and Albert Cohen. Control-driven data flow. Technical Report RR-8015, INRIA, July 2012.

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

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

[31] 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`.

[32] Jun Shirako, David Peixotto, Vivek Sarkar, and William Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *22nd Annual International Conference on Supercomputing (ICS'08)*, pages 277–288, 2008.

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

[34] Sven Verdoolaege. ISL: Integer set library. `http://freecode,com/projects/isl/`.

[35] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok rational functions. In *Algorithmica*, 2007.

[36] Ian Watson and John R. Gurd. A practical data flow computer. *IEEE Computer*, 15(2):51–57, 1982.

[37] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay 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.