Bounded Stream Scheduling in Polyhedral OpenStream

Nuno Miguel Nobre
The University of Manchester
Manchester, United Kingdom
nunomiguel.nobre@manchester.ac.uk

Andi Drebes
Inria and École Normale Supérieure
Paris, France
andi.drebes@inria.fr

Graham Riley
The University of Manchester
Manchester, United Kingdom
graham.riley@manchester.ac.uk

Antoniu Pop
The University of Manchester
Manchester, United Kingdom
antoniu.pop@manchester.ac.uk

Abstract

We consider OpenStream, a streaming dataflow language which supports the specification of concurrent tasks that communicate through streams. Streams, in the spirit of classical process networks, have no restrictions on their size. In order to deploy an OpenStream program on a chip, however, the size of the streams has to be bounded. This constricts the range of runtime behavior by restricting the schedules to a subset of parallel executions where the required memory never surpasses the available resources. In this paper we exploit an approach that, conservatively, certifies that augmenting the intrinsic dataflow dependencies of the program with stream bounding constraints does not deadlock the program: it cannot show the existence of a deadlock but can give a certificate for the absence thereof. The aim of this work is to study the limitations of this stream bounding strategy and to demonstrate how it can currently be used to determine if an OpenStream program can execute under the particular memory constraints of a given architecture.

Keywords Buffer bounding; Dataflow task-parallelism; OpenStream

1 Context

The industry is rapidly shifting to many-core architectures and specialized hardware accelerators, e.g., GPUs, FPGAs and ASICs. The differing capabilities of these devices increase the pressure on the programmer to develop both scalable and performance-portable code, hindering the programming process.

Task-parallel programming models are an answer to these scalability, performance-portability and productivity issues [3–8, 15, 18, 21, 24, 26, 28]. In particular, those based on streaming dataflow principles, e.g., StreamIt [18, 28], ΣC [15], OpenStream [26], have strong assets to follow this architecture scaling trend. Computations are encapsulated in tasks, the program's work units, that communicate through explicit, possibly unbounded data channels, or streams[1]. Task-parallel dataflow programs can be represented by directed graphs whose nodes are tasks and whose edges describe the producer-consumer task dependencies resulting from stream accesses. Each task's execution is triggered by the availability of its data operands, allowing latency to be naturally hidden, as well as precise, point-to-point synchronization of parallel activities. This is in stark contrast to fork-join and barrier-based synchronization patterns which require expensive global consensuses between workers. Hence, numerous opportunities for parallelism are naturally exposed [16, 19], including data, pipeline and task-parallelism [14]. This class of languages also enforce functional determinism at the language level, a desirable property inherited from Kahn Process Networks (KPNs) [17] which form the basis for most modern deterministic streaming languages.

In order to realize dataflow programs on a tangible chip with bounded resources, streams cannot be allowed to grow beyond the available memory on said device. This poses the question of whether a given specification of a dataflow program can execute under these restrictions. In this paper, we explore a strategy first suggested in [10], based on polyhedral model techniques, to answer such question for OpenStream programs. In essence, the dataflow, Read-after-Write (RaW) producer-consumer task dependencies are augmented with new, artificial, Write-after-Read (WaR) dependencies. These prevent tasks from writing to a full stream by delaying their execution to after other tasks first consume from the stream, thereby freeing some memory. This contrasts with the non-blocking, unconditionally successful writes to unbounded data channels of traditional KPNs and can introduce deadlocks. Furthermore, as even in the polyhedral subset of OpenStream, dependencies might be polynomial, deadlock detection is a semi-decidable, undecidable problem [10]. On the bright side, the polynomial extensions to the polyhedral model developed by Feautrier [12] can be leveraged to obtain a (polynomial) schedule, i.e., a task execution order, whose

[1] Tasks are also known in the literature as agents, actors, processes and filters. Likewise, data channels or lines are also known as buffers. We adhere to OpenStream's nomenclature, which designates these structures as streams.

IMPACT 2020, January 22, 2020, Bologna, Italy

existence certifies the absence of deadlocks. This enables an approach which, conservatively, can certify the execution of a dataflow OpenStream program on memory constrained hardware. The aim of this work is not only to study the limitations of this stream bounding strategy beyond its undecidability restraints, but also to outline some envisioned use cases.

Section 2 provides a brief description of OpenStream, its polyhedral fragment and the technique devised in [10, 12] to compute polynomial dependencies and schedules. Section 3 outlines the strategy used to bound streams and illustrates its limitations, while Section 4 sketches some guidelines for using it in practice. Section 5 discusses the most closely related work, before we conclude in Section 6.

2 OpenStream: Dataflow Task-Parallelism

OpenStream is a task-parallel streaming dataflow language implemented as an extension to OpenMP supporting the specification of fine-grained task, data and pipeline parallelism [26]. The fully fledged computational model underlying the operational semantics of the language is detailed in [25] and a simplified, partial-model is also defined in [11]. We briefly recall the three main concepts of OpenStream: streams, dataflow tasks and the control program. We present the restrictions placed upon OpenStreams programs for compatibility with the polyhedral model and review the technical solutions used for computing dependencies and handling polynomial schedules.

2.1 Streams and Tasks

A *stream* is a one-dimensional array of indefinite size, whose elements are of the same type. Each element of a stream is written using Dynamic Single Assignment (DSA), i.e., each stream element is written at most once. Conceptually, a stream s has a read pointer J_s and a write pointer I_s that define which elements of the stream are affected by subsequent read and write accesses. Streams themselves can also be grouped in arrays of arbitrary size and dimension.

A *task* t has a work function, i.e., an arbitrary sequence of instructions acting on local variables and on a finite set of streams. Access to each of the referenced streams is provided through windows. A window on a stream s is characterized by the access type it provides (read or write), its horizon $h_{t,s}$, and the burst $b_{t,s}$. The horizon is a positive integer specifying the size of the window. The burst is a non-negative integer specifying the amount by which the stream's read or write pointer is shifted after task creation. The burst of a write access window is equal to its horizon, guaranteeing single assignment, i.e., dataflow-only dependencies. The burst of a read access is either zero or equal to its horizon. A burst of zero corresponds to the peek operation. Figure 1 depicts how these concepts are used in OpenStream to express the flow of data between producer and consumer tasks.

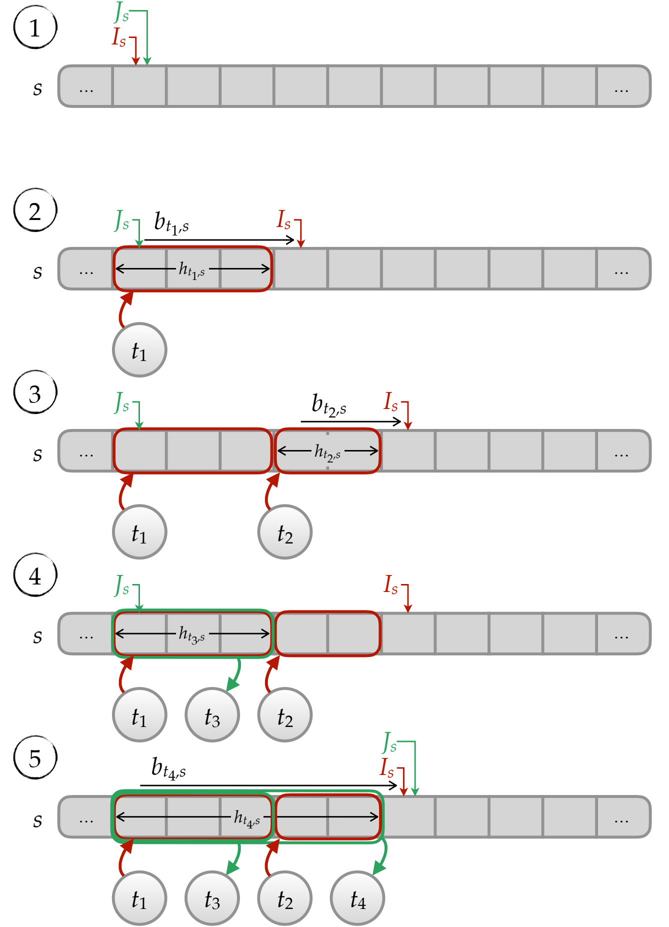

Figure 1. Illustration of stream accesses and the evolution of the read and write pointers for a stream s with equal bursts and horizons for all tasks, except t_3, and a zero burst for t_3 ($b_{t_1,s} = h_{t_1,s} = 3$, $b_{t_2,s} = h_{t_2,s} = 2$, $b_{t_4,s} = h_{t_4,s} = 5$ and $b_{t_3,s} = 0, h_{t_3,s} = 3$). (1) shows the initial state of the stream before any access. In (2) and (3), producers t_1 and t_2, respectively, are created and the write pointer I_s updated. Lastly, in (4) and (5), reader t_3 and consumer t_4, respectively, are created and, as the former only peeks on s, the read pointer J_s is advanced solely for t_4.

Stream accesses through windows determine the producer-consumer relationships: a task is a producer for another task if its output window overlaps with the other task's input window. Such relationships are captured in the task graph, such as the one in Figure 2 for the example of the stream accesses in Figure 1.

2.2 The Control Program

All task creations take place in the *control program*, an ordinary function containing task instantiation statements. Each such statement includes the respective list of read and write accesses to streams, each specified as a reference to a stream,

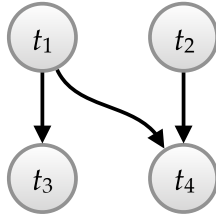

Figure 2. Task graph for the example of Figure 1. Task t_3 reads, without consuming, data produced by t_1 and must wait for its execution. Task t_4 consumes data produced by both t_1 and t_2 and, thus, must be executed after both t_1 and t_2 finish. Tasks t_1 and t_2, however, are dependency-free and can be executed concurrently.

a burst and a horizon. The order of task creation is defined by the order in which the control program executes the task instantiation statements. This order defines all producer-consumer relationships, since the read and write pointers at each task creation only depend on the bursts of prior task creations referencing the same streams. Note that the resulting producer-consumer relationships only *partially* define the order of task execution: unrelated tasks can execute in any order.

The control program for the example in Figures 1 and 2 is specified below using the compact OpenStream pseudo-code introduced in [10, 13]. This notation focuses on the dependencies between tasks and generally omits the tasks' work function. It uses an extended C syntax for operations directing the control flow; declarations for parameters, streams and stream arrays; and executable statements for task creation:

```
stream s;

task t1 {
  write three times to s;
}

task t2 {
  write two times to s;
}

task t3 {
  peek three times from s;
}

task t4 {
  read five times from s;
}
```

2.3 The Polyhedral Fragment of OpenStream

The analysis is confined to the subset of OpenStream in which the control program fits the polyhedral model, as first defined in [10]. We only consider programs that are deterministic by construction, i.e., programs for which the task creation order in the control program, and thus the interleaving of data in streams, is determined statically. A sufficient (but not necessary) condition is to require a sequential control program [25]. That is, the code of the tasks themselves can be arbitrary, but nested task creation is not allowed, i.e., tasks cannot instantiate other tasks. In addition, communication between tasks and the control program using shared variables and OpenMP-inherited mechanisms like `firstprivate` or `copyin` is not allowed. Control flow stems, exclusively, from the textual sequence of statements, affine conditional expressions and arbitrarily nested counted loops with affine bounds in the surrounding loop iterators and global parameters. The only executable statements are task creation statements. Window bursts and horizons may be numeric or parametric constants. Stream array indexing must be affine.

2.4 Computing Dependencies and Schedules

If the control program fits in the polyhedral model, we can statically compute closed form expressions for $J_s(t)$ and $I_s(t)$, the read and write pointers of a stream s at a point in the control program where task t which reads from or, respectively, writes to stream s is instantiated. As, at each task creation, these pointers are incremented by the burst, it is enough to count the number of task creations which read from or, respectively, write to s and precede the creation of t in the execution of the control program.

Since, within the constraints on the previous section, task creations can be represented as integer tuples in the iteration domain of their task instantiation statements and as these are polyhedra, their count can be obtained using the theory of Ehrhart polynomials [9] or Barvinok's Brion generating functions [30] which are both available in the barvinok library[2]. The result will, in general, be a polynomial.

As streams have the single assignment property, we need only compute dataflow or RaW dependencies. For a task t which writes to s through window $[I_s(t), I_s(t) + b_{t,s} - 1]$, and a task t' which reads from s through window $[J_s(t'), J_s(t') + h_{t',s} - 1]$, there is a dependency if the two windows overlap, i.e., if:

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

These dependencies inherit the polynomial nature of J_s and I_s and, thus, define semi-algebraic sets (which take the role of dependence polyhedra in the polyhedral model).

Finding schedules over these semi-algebraic sets can be achieved by leveraging Schweighofer's theorem [27] and Feautrier's approach [12] which, respectively, play the role of Farkas lemma and its algorithmic application in traditional tools. In a sentence, this entails finding a polynomial function that can be written as a non-negative linear combination of products of the polynomials that define the set. Unfortunately, deciding on the existence of a schedule over semi-algebraic sets is only semi-decidable. That is, if a schedule is found then, by construction, it is guaranteed to be valid and, thus, rules out the possibility of deadlocks. On the

[2] [Online] Available at http://barvinok.gforge.inria.fr.

contrary, if no schedule is found, it could be that the search space is limited by the order of the polynomial products considered. As we cannot possibly exhaust all linear combinations of higher degree in finite time, we cannot guarantee the non-existence of a schedule and, therefore, the existence of a deadlock.

3 Bounding Streams

We present a strategy to bound streams in OpenStream programs by introducing new WaR, 'back-pressure' dependencies in the polyhedral representation of the program. A few example OpenStream programs are presented which illustrate the procedure and its limitations.

3.1 Back-pressure Dependencies

If for a task execution schedule θ and a stream s there exists a positive integer l_s, for which all indices $i \leq j - l_s$ are dead (i.e. already both produced and consumed), whenever position j is written to by some task, then s is effectively bound by l_s for θ. This suggests that we can first append back-pressure dependencies that reproduce this behavior to Equation 1 and only then look for a schedule. To do this, we enforce all indices i to be consumed before j is written, i.e., we introduce a dependency from a task t which reads from s through $[J_s(t), J_s(t) + h_{t,s} - 1]$ to a task t' which writes to s through $[I_s(t'), I_s(t') + b_{t',s} - 1]$ whenever $\exists i \in [J_s(t), J_s(t) + h_{t,s} - 1]$ and $j \in [I_s(t'), I_s(t') + b_{t',s} - 1]$, such that $i \leq j - l_s$, which is equivalent to:

$$J_s(t) \leq I_s(t') + b_{t',s} - 1 - l_s. \tag{2}$$

Once the dependencies captured by both Equation 1 and Equation 2 have been added to the dependence relation, Feautrier's approach, as summarized in Subsection 2.4, can be leveraged to find a schedule with bounded streams.

The following example illustrates the result of this procedure. It starts with a dataflow program with a schedule θ_u with trivially-parallel reads and, independently, writes, and obtains a purely sequential schedule θ_b when stream s is bounded by $l_s = 2$.

```
stream s;
parameter N;

for(k = 0; k < N; ++k)
  task tw {                    // θu(tw,k) = 0; θb(tw,k) = 2k
    write two times to s;      // Is(tw,k) = 2k
  }

for(k = 0; k < N; ++k)
  task tr {                    // θu(tr,k) = 1; θb(tr,k) = 2k + 1
    read two times from s;     // Js(tr,k) = 2k
  }
```

Figure 3 illustrates how the back-pressure dependencies introduced for $l_s = 2$ force the execution of $t_{\text{w},k}$ to succeed $t_{\text{r},k-1}$ and precede $t_{\text{r},k}$.

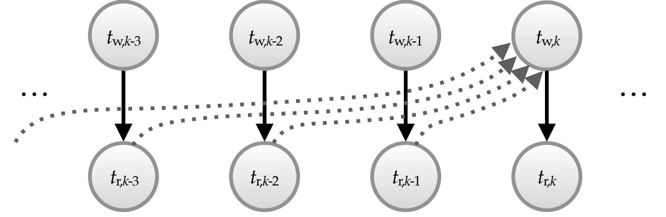

Figure 3. Fragment of the dependence task graph for the given example. This depicts the back-pressure dependencies for $t_{\text{w},k}$ as defined by Equation 2 for $l_s = 2$ (dashed) and the dataflow dependencies given by Equation 1 (solid).

We now provide a few examples that illustrate the limitations of this procedure.

3.2 Non-causal schedules

A causal schedule θ_c is a schedule for which writes to a given stream s occur in the same order as their write pointer indices advance, i.e., in the same order as the creation of the corresponding tasks. In other words, if t and t' are two producers in s, and t precedes t' in the control program, then $I_s(t) < I_s(t')$, but also $\theta_\text{c}(t) < \theta_\text{c}(t')$.

Consider the following example:

```
stream s;
parameter N;

task tsink {                // θ(tsink) = N
  read once from s;         // Js(tsink) = 0
}

for(k = 1; k < N; ++k)
  task tw {                 // θ(tw,k) = N − k
    read once from s;       // Js(tw,k) = k
    write once to s;        // Is(tw,k) = k − 1
  }

task tsource {              // θ(tsource) = 0
  write once to s;          // Is(tsource) = N − 1
}
```

The dataflow dependencies force the tasks in this program to be sequentially executed in the reverse order of their creation. Thus, this program only admits strict non-causal schedules (as the one given in the listing):

$$\forall t, t'. I_s(t) < I_s(t') \rightarrow \theta(t') < \theta(t).$$

However, any stream-bounding dependency of the type of Equation 2 requires at least 'partial causality', i.e., for all indices $i \leq j - l_s$ to be consumed before j is written, Equation 1 also implies that they have to be produced before j is written. This cannot happen in the present program, and any such dependency will add a cycle to the dependence task graph in Figure 4 and thus introduce a deadlock.

It is important to note that this does not imply that s cannot be bounded. In particular, for $l_s \geq 2$ the program has a

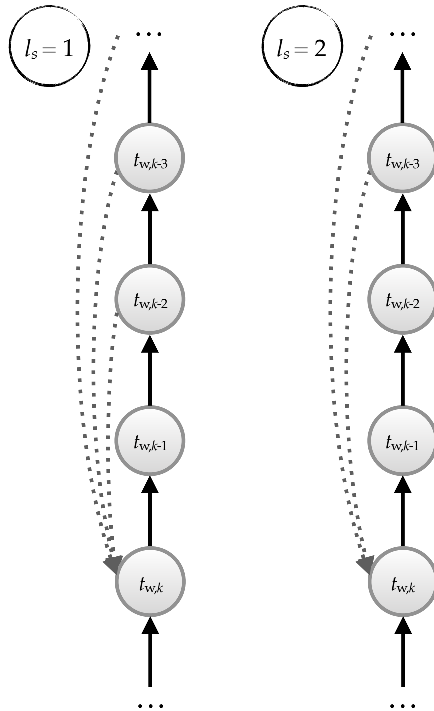

Figure 4. Fragment of the dependence task graph for the given example. This depicts both the back-pressure dependencies for $t_{w,k}$ for $l_s \leq 2$ (dashed) and the dataflow dependencies (solid).

valid schedule, e.g., the schedule provided in the listing[3]. The current strategy fails to capture this due to its implicit assumption that streams are filled in a (partially) causal order, which is not necessarily true for OpenStream programs. In summary, this shows that 'spurious' deadlocks can be introduced even for programs where streams can be effectively bounded, as the current approach introduces artificial back-pressure dependencies that go beyond bounding streams by implicitly enforcing 'some' causality.

3.3 Minimizing stream sizes

The programmer could be interested in determining the size l_s of each stream s such that the global surface of the streams $\sum_s l_s$ is minimized and there is no cycle in the dependence graph. Unfortunately, this is not equivalent to a multi-objective problem on the size of each individual stream. In other words, the sizes of different streams are not independent and therefore cannot be optimized separately and successively, one after the other. Consider the following example:

```
stream s, t;

task tws {
  write two times to s;
}

task twt {
  write three times to t;
}

task tcst {
  write once to s;
  read three times from t;
}

task tcts {
  write two times to t;
  read two times from s;
}

task trs {
  read once from s;
}

task trt {
  read two times from t;
}
```

If stream s is minimized first, noticing that t_{ws} requires $l_s \geq 2$, we find the smallest bound $l_s = 2$ is a solution if $l_t \geq 5$. The bound on stream s sequentializes the execution of t_{cst} after t_{cts}, requiring a large enough bound on stream t to prevent the introduction of the reverse dependency, as seen in Figure 5 (left). Conversely, if stream t is optimized first, t_{wt} requires $l_t \geq 3$, and we find $l_t = 3$ is feasible for $l_s \geq 3$. Figure 5 (center) shows the corresponding dependence graph.

Minimizing the global stream surface is a well-known problem in the context of KPNs, and has been proven NP-complete for as few as three tasks [20]. This example shows how, for OpenStream, like for KPNs, a strategy that tries to minimize each stream successively produces different results depending on the order streams are considered in. Next, we show how, unlike for KPNs, it is possible that none of the identified solutions minimizes the global stream surface.

In fact, although within the scope of the back-pressure dependencies strategy introduced in Subsection 3.1 these solutions are both minimal for the component-wise order in (l_s, l_t), neither solution minimizes the global stream surface. In reality, even though a situation where $l_s = 2$ and $l_t = 3$ apparently introduces a deadlock, as seen in Figure 5 (right), this is a 'spurious' deadlock of the type discussed in Subsection 3.2. For a (non-causal) schedule which executes each of the two (unilaterally) connected components of the dataflow dependence task graph one after the other, these bounds are both respected: sequentially execute t_{ws}, followed by t_{cts} and t_{rt}, and finally t_{wt}, followed by t_{cst} and t_{rs}.

Furthermore, one should note that the determined global stream surface does not correspond to the maximum memory usage of the program. Notice that, even if our strategy were able to identify $l_s = 2$ and $l_t = 3$ as the global stream surface minimizer, the sequential schedule described above provides

[3] Tasks execute atomically. When a producer task starts execution, it must already have enough space to write to the stream. If, as when $l_s = 1$, a producer relies on itself to free the stream, the program deadlocks.

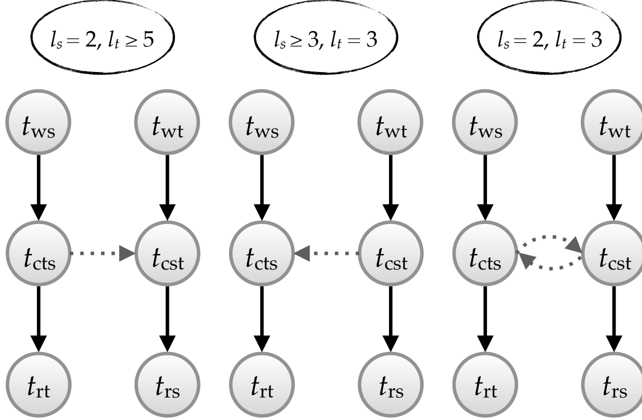

Figure 5. Dependence task graphs for the given example: (left) and (center) for the two minimal stream size solutions described in the text and (right) for the stream sizes that minimize the global stream surface but cannot be identified with the current strategy due to the 'spurious' deadlock. This depicts both back-pressure dependencies (dashed) and dataflow dependencies (solid).

an example of an execution where streams are never simultaneously fully utilized. In other words, at no point during the execution do stream s and stream t simultaneously hold two and three elements, respectively. This shows that, even if a minimum global stream surface were to be found, this would still be a conservative approximation of the actual memory requirements of the program.

4 Guidelines for Practical Applicability

Despite the undecidability of the scheduling problem and the 'spurious' constraints introduced by the stream bounding strategy, an approach can still be devised through which, given the resource constraints of a particular target architecture, one can, conservatively, decide on the execution feasibility of a particular OpenStream dataflow program.

Consider an arbitrary device, e.g., an FPGA, with some memory of total size M and an OpenStream program with n streams. We first compute all combinations of $(l_s)_{1 \leq s \leq n}$ such that $\sum_{s=1}^{n} l_s = M$ through any suitable method, e.g., stars and bars. Next, we build the dependence relation for each of these combinations by leveraging Equation 1 and Equation 2. Finally, we apply Feautrier's approach [12] in order to find a schedule. Once a schedule is constructed, we know that if we enforce the same dependencies at runtime, the streams are implemented as bounded streams in accordance to the combination for which the schedule was built, thereby guaranteeing successful execution on said device. Note that it is possible to take into account the resource requirements of tasks' work functions if these are known statically, as it is likely the case for FPGA kernels. In that case, the method just outlined is still applicable by replacing M with $M' = M - M_{wf}$ where M_{wf} are the collective memory requirements of the tasks' work functions.

Constraining a dataflow program to run on such devices can, however, diminish the amount of parallelism available. Let us reconsider the example in Subsection 3.3. If $M = 8$, we may set $l_s = 3$ and $l_t = 5$, essentially making the program free from back-pressure dependencies. The program can then execute in three waves: first t_{ws} and t_{wt} simultaneously; then t_{cst} and t_{cts}; and lastly t_{rs} and t_{rt}. If, however, $M = 6$, the only possibility is, within the proposed strategy, $l_s = l_t = 3$ which, as we have seen, serializes the execution of t_{cts} after t_{cst}.

On the other hand, these memory constrained devices may be massively parallel. Thus, even if some parallelism is lost due to memory constraints, the parallelism opportunities available might still greatly surpass the capabilities of many-core CPUs. In addition, the specialized hardware they expose to the programmer might also allow faster implementations of tasks' work functions, thereby compensating for the restrictions imposed by the constrained memory resources.

5 Related Work

Kahn Process Networks (KPNs) [17] underpin most modern languages based on streaming computing concepts, including OpenStream. Perhaps unsurprisingly, Parks [22] has shown KPNs' expressive power comes at the cost of undecidability for deadlock detection and bounded buffering guarantees.

However, despite the similarities, OpenStream and KPNs have subtle differences and the equivalence of the models is still not fully understood. Unlike KPNs, OpenStream allows streams to be read and written by several tasks; and, unlike OpenStream, KPNs allow conditional decisions on what streams to read from based on the data that circulates on the streams. Nevertheless, Cohen et al. [10] have shown the same undecidability properties hold for OpenStream programs.

This leaves a few options to the programmer willing to exploit data-centric programming models: (1) resort to models with restricted expressive power like Synchronous DataFlow (SDF) and Cyclo-Static DataFlow (CSDF) that statically provide deadlock-absence and bounded memory guarantees [23]; (2) resort to approaches like the one described here to, in practice, albeit conservatively, certify the execution of more expressive dataflow programs on devices with constrained memory resources; or (3) rely on automated process network generation from static control loop nests initially written in an imperative language, like Verdoolaege's Polyhedral Process Networks [29] which allow conservative estimation of the required buffer sizes for a given schedule, but are limited to polyhedral-amenable code. This choice will probably be ultimately guided by the application domain, but we believe that current and upcoming architectures require dataflow models that are more expressive - even than the polyhedral subset of OpenStream defined here - that allow irregular, dynamic communication patterns and the execution

of non-polyhedral code, e.g., as part of tasks' work functions. In this sense, the presented work builds on this insight and motivates future work in this direction.

Lastly, regarding the stream minimization problem of Subsection 3.3, the work in [1, 2, 20] on buffer minimization in the context of restricted KPNs, SDF and CSDF graphs might pave the way for the same kind of optimizations in OpenStream, or a subset of the language, in future work.

6 Conclusion

We showed how augmenting polyhedral OpenStream programs with back-pressure dependencies can be used to bound streams and statically, albeit conservatively, decide if said programs can be executed in devices with limited memory. We described how the current method is limited in its power due to the introduction of 'spurious' deadlocks, the difficulty of global stream minimization, the overestimation of actual memory usage and deadlock detection undecidability.

Acknowledgments

This work was supported by the EU FET-HPC grants ExaNoDe H2020-671578 and EuroEXA H2020-754337. A. Pop is funded by a RAEng University Research Fellowship.

References

[1] M. Benazouz, O. Marchetti, A. Munier-Kordon, and P. Urard. 2010. A new approach for minimizing buffer capacities with throughput constraint for embedded system design. In *ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010*. 1–8. https://doi.org/10.1109/AICCSA.2010.5586972

[2] Mohamed Benazouz and Alix Munier-Kordon. 2013. Cyclo-static DataFlow Phases Scheduling Optimization for Buffer Sizes Minimization. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems (M-SCOPES '13)*. ACM, New York, NY, USA, 3–12. https://doi.org/10.1145/2463596.2463602

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55 – 69. https://doi.org/10.1006/jpdc.1996.0107

[4] François Broquedis, Thierry Gautier, and Vincent Danjean. 2012. LIBKOMP, an Efficient OpenMP Runtime System for Both Fork-join and Data Flow Paradigms. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World (IWOMP'12)*. Springer-Verlag, Berlin, Heidelberg, 102–115. https://doi.org/10.1007/978-3-642-30961-8_8

[5] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. 2010. Concurrent Collections. *Sci. Program.* 18, 3-4 (Aug. 2010), 203–217. https://doi.org/10.1155/2010/521797

[6] D. Callahan, B. L. Chamberlain, and H. P. Zima. 2004. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* 52–60. https://doi.org/10.1109/HIPS.2004.1299190

[7] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/2093157.2093165

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.* 40, 10 (Oct. 2005), 519–538. https://doi.org/10.1145/1103845.1094852

[9] Philippe Clauss. 1996. Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs. In *Proceedings of the 10th International Conference on Supercomputing (ICS '96)*. ACM, New York, NY, USA, 278–285. https://doi.org/10.1145/237578.237617

[10] Albert Cohen, Alain Darte, and Paul Feautrier. 2016. Static Analysis of OpenStream Programs. In *6th International Workshop on Polyhedral Compilation Techniques (IMPACT'16), held with HIPEAC'16 (Proceedings of the IMPACT series)*. Michelle Strout and Tomofumi Yuki, Prague, Czech Republic. https://hal.inria.fr/hal-01251845

[11] Andi Drebes. 2015. *Dynamic optimization of data-flow task-parallel applications for large-scale NUMA systems.* Theses. Université Pierre et Marie Curie - Paris VI. https://tel.archives-ouvertes.fr/tel-01258451

[12] Paul Feautrier. 2015. The Power of Polynomials. In *5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*, Alexandra Jimborean and Alain Darte (Eds.). Amsterdam, Netherlands. https://hal.inria.fr/hal-01094787

[13] Paul Feautrier and Albert Cohen. 2018. On Polynomial Code Generation. In *8th International Workshop on Polyhedral Compilation Techniques (IMPACT'18)*. Manchester, United Kingdom.

[14] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGPLAN Not.* 41, 11 (Oct. 2006), 151–162. https://doi.org/10.1145/1168918.1168877

[15] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. 2011. ΣC: A Programming Model and Language for Embedded Manycores. In *Algorithms and Architectures for Parallel Processing*, Yang Xiang, Alfredo Cuzzocrea, Michael Hobbs, and Wanlei Zhou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 385–394.

[16] J. R Gurd, C. C Kirkham, and I. Watson. 1985. The Manchester Prototype Dataflow Computer. *Commun. ACM* 28, 1 (Jan. 1985), 34–52. https://doi.org/10.1145/2465.2468

[17] G. Kahn. 1974. The semantics of a simple language for parallel programming. In *Information processing*, J. L. Rosenfeld (Ed.). North Holland, Amsterdam, Stockholm, Sweden, 471–475.

[18] Kimberly Kuo, Rodric M. Rabbah, and Saman Amarasinghe. 2005. A Productive Programming Environment for Stream Computing. In *In Proceedings of the 2nd Second Workshop on Productivity and Performance in High-End Computing*.

[19] Ben Lee and A. R. Hurson. 1994. Dataflow Architectures and Multithreading. *Computer* 27, 8 (Aug. 1994), 27–39. https://doi.org/10.1109/2.303620

[20] Alix Munier-Kordon and Jean-Baptiste Note. 2005. A Buffer Minimization Problem for the Design of Embedded Systems. *European Journal of Operational Research* 164, 3 (Aug. 2005), 669–679. https://doi.org/10.1016/j.ejor.2004.01.041

[21] OpenMP Architecture Review Board. 2008. *OpenMP Application Program Interface Version 3.0.*

[22] Thomas Martyn Parks. 1995. *Bounded Scheduling of Process Networks.* Ph.D. Dissertation. Berkeley, CA, USA. UMI Order No. GAX96-21312.

[23] T. M. Parks, J. L. Pino, and E. A. Lee. 1995. A comparison of synchronous and cycle-static dataflow. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, Vol. 1. 204–210 vol.1. https://doi.org/10.1109/ACSSC.1995.540541

[24] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. 2009. Hierarchical Task-Based Programming With StarSs. *Int. J. High Perform. Comput. Appl.* 23, 3 (Aug. 2009), 284–299. https://doi.org/10.1177/1094342009106195

[25] Antoniu Pop and Albert Cohen. 2012. *Control-Driven Data Flow*. Research Report RR-8015. INRIA. 48 pages. https://hal.inria.fr/hal-00717906

[26] Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article 53 (Jan. 2013), 25 pages. https://doi.org/10.1145/2400682.2400712

[27] Markus Schweighofer. 2002. An algorithmic approach to Schmüdgen's Positivstellensatz. *Journal of Pure and Applied Algebra* 166, 3 (2002), 307 – 319. https://doi.org/10.1016/S0022-4049(01)00041-X

[28] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, London, UK, 179–196. http://dl.acm.org/citation.cfm?id=647478.727935

[29] Sven Verdoolaege. 2010. *Polyhedral Process Networks*. Springer US, Boston, MA, 931–965. https://doi.org/10.1007/978-1-4419-6345-1_33

[30] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. 2007. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica* 48, 1 (01 May 2007), 37–66. https://doi.org/10.1007/s00453-006-1231-0