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 constrains 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. 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
We briefly recall the three main concepts of OpenStream: written using Dynamic Single Assignment (DSA), i.e., each stream \( A \) specifies the amount by which the stream’s read or write access window is equal to its horizon, guaranteeing single pointer is shifted after task creation. The burst of a write access window is equal to its horizon, specifying the amount by which the stream’s read or write access window is equal to its horizon, guaranteeing single pointer is shifted after task creation. The burst of a write

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 \( h_{ts} \), and the burst \( b_{ts} \). 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 \( t_5 \) (\( b_{t_1,s} = h_{t_1,s} = 3, b_{t_2,s} = h_{t_2,s} = 2, b_{t_3,s} = h_{t_3,s} = 5 \) and \( b_{t_5,s} = 0, h_{t_5,s} = 3 \)). (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 \( t_5 \) and consumer \( t_3 \), 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,
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 $[I_s(t), I_s(t) + h_{t,s} - 1]$, 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:

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

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 \(\theta\) 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 \(\theta\). 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 \(\theta_o\) with trivially-parallel reads and, independently, writes, and obtains a purely sequential schedule \(\theta_h\) when stream \(s\) is bounded by \(l_s = 2\).

```plaintext
stream s;
parameter N;
for(k = 0; k < N; ++k)
task tw {
    // \(\theta_o(t_{w,k}) = 0; \theta_h(t_{w,k}) = 2k\)
    write two times to s; // \(I_s(t_{w,k}) = 2k\)
}
for(k = 0; k < N; ++k)
task tr {
    // \(\theta_o(t_{r,k}) = 1; \theta_h(t_{r,k}) = 2k + 1\)
    read two times from s; // \(J_s(t_{r,k}) = 2k\)
}
```

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

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

3.2 Non-causal schedules

A causal schedule \(\theta_c\) 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 \(\theta_c(t) < \theta_c(t’).\)

Consider the following example:

```plaintext
stream s;
parameter N;
task tsink {
    // \(\theta(t_{sink}) = N\)
    read once from s; // \(J_s(t_{sink}) = 0\)
}
for(k = 1; k < N; ++k)
task tw {
    // \(\theta(t_{w,k}) = N - k\)
    read once from s; // \(J_s(t_{w,k}) = k\)
    write once to s; // \(I_s(t_{w,k}) = k - 1\)
}
task tsink {
    // \(\theta(t_{sink}) = 0\)
    read once to s; // \(I_s(t_{sink}) = N - 1\)
}
```

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. For example, for \(l_s \geq 2\) the program...
has a valid schedule\(^3\). 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 successively, one after the other. Consider the following example:

\[\text{stream } s, t;\]
\[\text{task } \text{tws} \{\]
\[\text{task } \text{twt} \{\]
\[\text{task } \text{tcst} \{\]
\[\text{task } \text{tcts} \{\]
\[\text{task } \text{trs} \{\]
\[\text{task } \text{trt} \{\]

If stream \(s\) is minimized first, we find that \(l_s = 2\) is a valid solution if \(l_t \leq 5\). Conversely, if the size of stream \(t\) is optimized first, \(l_t = 3\) is feasible for \(l_s \geq 3\). Figure 5 shows the dependence graphs for each of these two situations. Although these solutions are both minimal for the component-wise order in \((l_s, l_t)\), they correspond to different global stream surfaces. Notice that, even though, \(l_s = 3\) and \(l_t = 3\) minimize the global surface for this simple example, a procedure minimizing each stream individually does not scale to the general case. In fact, 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].

Furthermore, one should notice that the global stream surface does not correspond to the maximum memory usage of the program. Let \(m_s\) and \(m_t\) be the number of live elements in streams \(s\) and \(t\) at a given instant during the execution. For \(l_s = 3\) and \(l_t = 3\), consider the following (causal) schedule: execute \(\text{tws}\) and \(\text{twt}\) simultaneously: \(m_s = 2\), \(m_t = 3\); then successively \(\text{tcst}\): \(m_s = 3\), \(m_t = 0\); and \(\text{tcts}\): \(m_s = 1\), \(m_t = 2\); and, finally, \(\text{trs}\) and \(\text{trt}\) simultaneously: \(m_s = 0\), \(m_t = 0\). Even though the global stream surface is \(l_s + l_t = 6\), at no point \(m_s + m_t = 6\), despite, individually, both \(m_s = l_s\) and \(m_t = l_t\) for some instant during the execution\(^4\). This shows that, even if a minimum global stream surface were to be found, this would still intrinsically be a conservative approximation of the actual memory requirements.

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,

---

\(^3\)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.

\(^4\)Otherwise, this would not be a minimal solution.
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, at runtime, the streams are implemented as bounded streams in accordance to the combination for which the schedule was built, the program is guaranteed to be able to execute 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 $M’ = M - M_{\text{off}}$ where $M_{\text{off}}$ 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 $t_{ct}$ and $t_{cts}$; and lastly $t_{rt}$ and $t_{rs}$. If, however, $M = 6$, the only possibility is $l_t = l_s = 3$ which, as we have seen, serializes the execution of $t_{cts}$ after $t_{ct}$.

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.
resources. We described how the current method is limited in its power due to the introduction of ‘spurious’ deadlocks, the difficulty of stream minimization, the overestimation of actual memory usage and deadlock detection undecidability.

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

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