Polyhedral Compilation for Multi-dimensional Stream Processing

JAKOB LEBEN and GEORGE TZANETAKIS, University of Victoria, Canada

We present a method for compilation of multi-dimensional stream processing programs from affine recurrence equations with unbounded domains into imperative code with statically allocated memory. The method involves a novel polyhedral schedule transformation called periodic tiling. It accommodates existing polyhedral optimizations to improve memory access patterns and expose parallelism. This enables efficient execution of programming languages with unbounded recurrence equations, as well as optimization of existing languages from which this form can be derived. The method is experimentally evaluated on 5 DSP algorithms with large problem sizes. Results show potential for improved throughput compared to hand-optimized C++ (speedups on a 6-core Intel Xeon CPU up to 10× with a geometric mean 3.3×).[1]

CCS Concepts: • **Software and its engineering → Compilers**; *Data flow languages*;

Additional Key Words and Phrases: Polyhedral compilation, multi-dimensional stream processing, digital signal processing, recurrence equations

ACM Reference format:
Jakob Leben and George Tzanetakis. 2019. Polyhedral Compilation for Multi-dimensional Stream Processing. *ACM Trans. Archit. Code Optim.* 16, 3, Article 27 (July 2019), 26 pages.
https://doi.org/10.1145/3330999

1 INTRODUCTION AND MOTIVATION

Stream processing programs (also called streaming programs) repeatedly perform the same computation to produce finite consecutive segments of output data streams, while consuming finite consecutive segments of input data streams. Such programs can operate on streams of data without a pre-determined length or virtually infinite streams. Frequently, these are high-volume streams that must be processed in real time with low latency, and hence optimization is crucial. Examples include processing of audio, video, and other media, as well as sensor arrays. Such programs typically exhibit very regular control flow and data dependencies. This provides many opportunities for static analysis and optimization. In addition, the programmer can be freed from certain tasks such as detailed scheduling or memory allocation, because the compiler can perform them with equal or better results. All this has given rise to various domain-specific programming languages and computational models for stream processing.

[1]New article, not an extension of a conference paper.

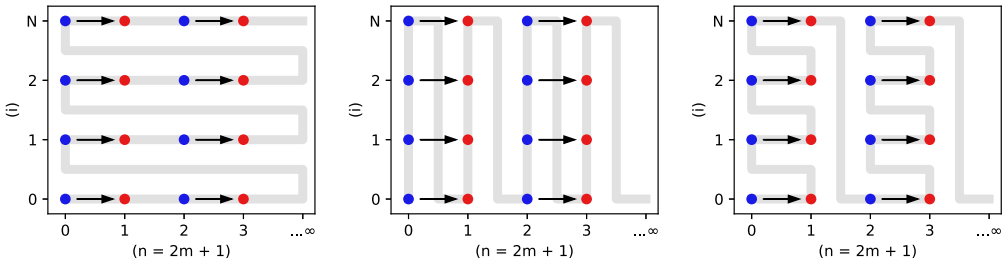
This work is supported by the National Science and Engineering Research Council of Canada.
Authors' address: J. Leben and G. Tzanetakis, University of Victoria, Department of Computer Science, 3800 Finnerty Road, Victoria, BC V8P 5C2, Canada; emails: jakob.leben@gmail.com, gtzan@cs.uvic.ca.

1544-3566/2019/07-ART27
https://doi.org/10.1145/3330999

The *dataflow* paradigm turned out to be a good fit for streaming programs, even though its original purpose was to utilize fine-grained parallelism supported by hardware dataflow architectures—independently of the concept of infinite streams (see Johnston et al. (2004) for a historical overview including languages such as Id, VAL, SISAL). While some dataflow models support data-dependent conditional and iterative structures required for general-purpose programming, we are particularly interested in more restricted models that exploit the regularity of streaming programs for static scheduling and memory allocation. In this group, the most attention has been given to those that represent streams as single-dimensional sequences (SDF (Lee and Messerschmitt 1987), StreamIt (Thies et al. 2002a), CAL (Eker and Janneck 2003)). Multi-dimensional streams can be represented in such models, albeit at the cost of abstraction, which makes this less natural for the programmer and restricts potential transformations in the compiler. There are a few models that address this with a more flexible multi-dimensional representation of streams, e.g., MDSDF (Murthy and Lee 2002), Array-OL (Boulet 2007). Languages based on systems of recurrence equations like ALPHA (Charot et al. 2004; Le Verge et al. 1991), PAULA (Hannig 2009), and Arrp (Leben 2016)) are more distant from the dataflow paradigm. Nevertheless, we think they deserve more attention in the context of stream processing: They naturally model multi-dimensional streams; moreover, recurrence equations are commonly used to mathematically describe streaming algorithms—especially in the domain of digital signal processing (DSP). Such languages therefore provide a direct path from mathematical description to executable code.

We present novel techniques for translation of streaming programs from the form of recurrence equations into an efficient imperative form with statically allocated memory, readily executable on general purpose machines. For this purpose, we leverage the *polyhedral model* (Feautrier and Lengauer 2011). While this model has lately been more associated with the compilation of static affine nested loop programs (SANLP) (starting with Feautrier (1991)), its origin was indeed in the compilation of recurrence equations (Karp et al. 1967; Rajopadhye 1989; Rajopadhye and Fujimoto 1990; Saouter and Quinton 1993). Since then, a large amount of research has explored the benefits of the polyhedral model for automatic optimization and parallelization. These optimizations improve data locality and consequently cache utilization, expose opportunities for parallelism (Bondhugula et al. 2008a), and minimize storage size (Bhaskaracharya et al. 2016). Whereas dataflow programming has witnessed a shift from fine-grained to coarse-grained dataflow due to performance concerns (Johnston et al. 2004), we believe the polyhedral model may allow efficient execution of fine-grained streaming programs via detailed refactoring of stream operators. Expressing entire programs in a single fine-grained streaming paradigm may increase code modularity and reuse.

There has been some research at the intersection of stream processing, recurrence equations and the polyhedral model. To the best of our knowledge though, *unbounded* recurrence equations representing infinite streams have only been translated to hardware (e.g., Rajopadhye and Fujimoto (1990)), whereas this work focuses on software generation. Other work is restricted to *finite* streams or has other limitations. For example, Thies et al. (2002b) translates SDF to unbounded recurrence equations without providing a further compilation method. Keinert and Teich (2011) treats MDSDF in the polyhedral model, but limited to individual (finite) image processing. Bhaskaracharya and Bondhugula (2013) optimizes dataflow programs in a subset of the LabVIEW language where only finite arrays are found. The Polyhedral Process Networks (Verdoolaege 2013) model is a hybrid between the polyhedral and the SDF models, and it has been derived from affine nested loop programs with (statically or dynamically) bounded domains (e.g., Turjan et al. (2004) and Nadezhkin et al. (2013)). The PAULA language (Hannig 2009) theoretically supports unbounded recurrence equations, although we are not aware of compilation methods that can handle them. Unbounded polyhedra also appear in literature on polyhedral transformations of data-dependent, dynamically computed, or non-linear control flow, but only as



(a) Pluto's schedule. (b) Simple feasible schedule. (c) Periodically tiled schedule.

```
for(int c1 = 0; ; ++c1)
    for(int c2 = 0; c2 < N; ++c2)
        x[c1 % 2, c2] = input(c2)
        if((c1-1) % 2 == 0)
            y = x[(c1-1) % 2, c2]
            + x[c1 % 2, c2];
            output(y, c2)
```

(d) Entire schedule in Fig. 1b.

```
for(int c2 = 0; c2 < N; ++c2)
    x[0] = input(c2)
    x[1] = input(c2)
    y = x[0] + x[1];
    output(y, c2)
```

(e) A period of schedule in Fig. 1c.

Fig. 1. Examples of polyhedral schedules and generated code.

overapproximations of finite iteration domains (the program terminates) (Benabderrahmane et al. 2010; Lengauer and Griebl 1995; Zhao et al. 2018). In contrast, our work deals with issues arising from truly unbounded iteration domains (the streaming program does not terminate) where the aforementioned techniques have no use.

This work addresses challenges specific to generation of executable code for general purpose hardware from unbounded multi-dimensional recurrence equations that represent infinite streams. Let us demonstrate these challenges using a program implementing a simple form of downsampling of multiple channels as an example. Let $x(n, i)$ be the input signal on the domain $0 \leq n,\ 0 \leq i < N$, where n represents time and has no upper bound, and i is the channel index. Let the output be $y(m, i) = x(2m, i) + x(2m + 1, i)$ on the domain $0 \leq m, 0 \leq i < N$. Figure 1(a) depicts a schedule for this program computed by the popular scheduling algorithm due to Bondhugula et al. (2008a), which maximizes data locality. Each blue dot represents the computation of an element of x, and each red dot both an element of x and an element of y. Black arrows represent data dependencies. The grey line traces the order of execution, starting from $\langle 0, 0 \rangle$. According to this schedule though, $y(0, 1)$ would never be computed, even as time goes towards infinity, because $y(m, 0)$ for all m up to infinity are scheduled earlier. We say that such a schedule is *infeasible*. To address this, we propose a scheduling technique called *periodic tiling*, which partitions the schedule into a sequence of finite equally shaped tiles (periods). As an example, Figure 1(c) is a periodically tiled variant of the schedule in Figure 1(a)—it computes one sample for all output channels before the next sample. The data locality granted by the original schedule is preserved within the periods of the transformed schedule.

Another problem concerns existing polyhedral code generation techniques. Even with a feasible schedule as depicted Figure 1(b), existing techniques such as Grosser et al. (2015) create infeasible loops to scan schedule points, like in Figure 1(d). Note that the iterator `c1` has no upper bound and would overflow at some point. The *periodic tiling* technique presented in this article circumvents this issue. Since it identifies the periodically repeating pattern in the schedule, code can be generated for a single finite period (without unbounded variables) and repeatedly executed by a host program as desired. Figure 1(e) shows an example.

An important issue in the compilation of single-assignment languages such as recurrence equations is storage allocation—especially when it involves unbounded arrays. Optimization is critical, since a trivial allocation storing each array element into a unique memory location would require an infinite amount of memory. It is constrained though by scheduling decisions such as the proposed periodic tiling. Therefore, we prove that periodic tiling always admits finite storage using a form of storage allocation generally known as *modular mappings* (Darte et al. 2003) and in particular algorithms due to Lefebvre and Feautrier (1998) and Bhaskaracharya et al. (2016).

To sum up, the main contribution of this article is a complete method for translation of streaming programs from unbounded recurrence equations into executable code with statically allocated memory via the polyhedral model. The central piece is a novel schedule transformation called *periodic tiling* and its integration with existing storage allocation and code generation techniques. As an additional contribution, we demonstrate the potential for combination of this method with existing polyhedral optimizations for data locality and parallelization as well as some well-known buffer indexing optimizations. We empirically evaluate their effect on throughput, buffer size and latency in 5 signal processing algorithms. Results show potential for improved parallel throughput scaling compared to hand-written C++. On an Intel Xeon CPU with 6 cores, we observe speedups up to 10× (geometric mean 3.3×) over hand-optimized C++, and up to 10× (geometric mean 4.2×) over hand-written C++ optimized by the Intel C++ compiler. We also evaluate the StreamIt language with the same algorithms and reveal its limitations with large problem sizes.

2 BACKGROUND

2.1 Polyhedra and Integer Sets

The polyhedral model describes various aspects of a program using convex polyhedra. In general, a convex polyhedron is a set of points $\langle v_1, \ldots, v_d \rangle \in \mathbb{R}^d$ that satisfy a finite number of affine inequalities in the variables v_i:

Definition 2.1 (Convex Polyhedron). A convex polyhedron is a set: $P = \{ \vec{i} \in \mathbb{R}^d \mid A\vec{i} \leq \vec{b} \}$ for some matrix $A \in \mathbb{R}^{m \times d}$ and vector $\vec{b} \in \mathbb{R}^m$

More specifically, the polyhedral model uses integer subsets of polyhedra called $\mathbb{Z}$-polyhedra. In the rest of this article, we refer to such a set simply as a *polyhedron* unless explicitly stated otherwise.

Definition 2.2 ($\mathbb{Z}$-Polyhedron). A $\mathbb{Z}$-Polyhedron is a set: $P = \{ \vec{i} \in \mathbb{Z}^d \mid A\vec{i} \leq \vec{b} \}$ for some matrix $A \in \mathbb{Z}^{m \times d}$ and vector $\vec{b} \in \mathbb{Z}^m$.

We are interested especially in unbounded sets. The Minkowski-Weyl theorem is useful for the treatment of unbounded polyhedra:

Theorem 2.3 [Minkowski-Weyl Theorem]. *For every convex polyhedron $P \subset \mathbb{R}^d$, there is a finite number of vectors $\vec{v_1}, \ldots, \vec{v_n}$ and $\vec{r_1}, \ldots, \vec{r_m}$ such that $P = conv(\vec{v_1}, \ldots, \vec{v_n}) + cone(\vec{r_1}, \ldots, \vec{r_m})$.*

In the above, $conv(\vec{v_1}, \ldots, \vec{v_n}) = \{ \sum_{i=1}^{n} \gamma_i \vec{v_i} \mid \gamma_i \geq 0 \wedge \sum_{i=1}^{n} \gamma_i = 1 \}$ is the bounded set of convex combinations of vectors v_i (a polytope), $cone(\vec{r_1}, \ldots, \vec{r_m}) = \{ \sum_{i=1}^{m} \lambda_i \vec{r_i} \mid \lambda_i \geq 0 \}$ is the unbounded set of conical combinations of vectors r_i (a cone) and $S + T = \{ \vec{p} + \vec{q} \mid \vec{p} \in S \wedge \vec{q} \in T \}$ is the Minkowski sum of two sets. Each element of the cone is a *ray* and represents an *infinite direction* defined as follows:

Definition 2.4 (Ray, Infinite Direction). A *ray* of a set $P \subset \mathbb{R}^d$ is any $\vec{r}$ such that $\vec{x} \in P \Rightarrow \vec{x} + \vec{r} \in P$. Two rays $\vec{r_1}, \vec{r_2}$ are *equivalent*, if $\vec{r_1} = \lambda \vec{r_2}$ with $\lambda > 0$. Each ray in a set of equivalent rays represents the same *infinite direction*.

We are particularly interested in $\mathbb{Z}$-polyhedra with a single infinite direction with the following properties:

Corollary 2.5. *A $\mathbb{Z}$-polyhedron P has a single infinite direction and a non-empty set of rays if and only if it is a subset of some polyhedron with a single infinite direction. For every ray $\vec{r}$ of P, an integer multiple $\lambda\vec{r}$ with $\lambda \in \mathbb{Z}^+$ is also a ray. P has a unique ray with the smallest length (smallest ray).*

Relations between $\mathbb{Z}$-polyhedra in the polyhedral model can result in sets that are not polyhedra but *Presburger sets* of the form

$$\left\{ \vec{i} \in \mathbb{Z}^d \mid \exists \vec{e} \in \mathbb{Z}^e \ : \ A\vec{i} + B\vec{e} \leq \vec{c} \right\},$$

where A and B are integer matrices and $\vec{c}$ is an integer vector. For example, an image of a $\mathbb{Z}$-polyhedron by an affine mapping is such a set. Such a set can always be described as an orthogonal projection of a $\mathbb{Z}$-polyhedron $\langle v_1, \dots v_n \rangle \mapsto \langle v_l, \dots v_m \rangle$, where $1 \leq l \leq m \leq n$ (from here on simply called a *projection*). This is obvious, since for any set $\{ \vec{i} \mid \exists \vec{e} \ : \ F \}$ with an affine formula F there is a $\mathbb{Z}$-polyhedron: $\{ \langle \vec{i}, \vec{e} \rangle \mid F \}$ where $\langle \vec{i}, \vec{e} \rangle$ is a vector with concatenated coordinates of $\vec{i}$ and $\vec{e}$. In this article, the reader will also encounter sets defined using a formula $P(\lfloor v/c \rfloor)$ for some predicate P and constant c. Such sets are definable as Presburger sets (or projections of $\mathbb{Z}$-polyhedra), since the formula is equivalent to the following, where q and r represent the quotient and remainder of the integer division:

$$\exists q, r \ : \ P(q) \wedge (v = cq + r) \wedge (0 \leq r < c).$$

The reader will be able to verify that the statements about $\mathbb{Z}$-polyhedra in Corollary 2.5 also apply to projections of $\mathbb{Z}$-polyhedra.

We denote elements of a relation $Q \subseteq \mathbb{R}^n \times \mathbb{R}^m$ by $\langle \vec{i}, \vec{j} \rangle$ for some $\vec{i} \in \mathbb{R}^n$ and $\vec{j} \in \mathbb{R}^m$. Sometimes, we consider such relations simply as subsets of $\mathbb{R}^{n+m}$ where $\langle \vec{i}, \vec{j} \rangle$ denotes the concatenation of coordinates of $\vec{i}$ and $\vec{j}$ and it may be called a point, a vertex, a ray, and so on. We also write $\langle i_1, \dots i_n \mid j_1, \dots j_m \rangle$ when spelling out each coordinate. Note also that the domain and range of Q are projections of the set of concatenations of $\vec{i}$ and $\vec{j}$. We denote the range of a relation Q by $\mathrm{ran}(Q)$.

2.2 Polyhedral Model

In the polyhedral model, a program is represented as a set of statements $s_1, s_2, \dots, s_n$ that read and write values in multi-dimensional arrays $a_1, a_2, \dots, a_m$. The set of all valid array indices D_a for an array a is called an *array domain* and is represented as a polyhedron. Each statement s has a set of *instances*, each instance corresponding to an index $\vec{i}$ in the *statement domain* D_s also represented as a polyhedron. Each statement instance computes a value of the statement function $f_s(\vec{i})$ and writes the result into an array at an index $w(\vec{i})$, where w is an affine function. The result may depend on values of other arrays at indices $r(\vec{i})$ where r is an affine function. The reads and writes of array values are commonly called *array accesses*. Given an access by each statement instance $\vec{i}$ at an array index $q(\vec{i})$, an *access relation* is the relation $\{ \langle \vec{i}, q(\vec{i}) \rangle \mid \vec{i} \in D_s \}$.

In this article, we focus on polyhedral models of stream programs where streams are represented as arrays with unbounded domains, and each array element is written only by a single instance of a single statement. Such models are easily derived from systems of affine recurrence equations. The latter are frequently used as a mathematical description of stream processing algorithms (including in this article). They are also the basis of programming languages such as ALPHA (Le Verge et al. 1991) and Arrp (Leben 2016). A system of recurrence equations is a set of variables V. Each variable

a is a function defined by a set of equations on disjoint domains $D_a^1, \ldots, D_a^n$ with the following general form:

$$\forall \vec{i} \in D_a^j : a(\vec{i}) = f^j(b_1(r_1(\vec{i})), \ldots, b_m(r_m(\vec{i}))), \tag{1}$$

where $b_1, \ldots, b_m$ are variables in V. A system of *affine* recurrence equations (SARE) is one where each D_a^j is a polyhedron and $r_1, \ldots, r_m$ are affine functions. The derivation of a polyhedral model from a SARE is straightforward. Each variable corresponds to an array with the domain $D_a = \bigcup_{j \in (1,n)} D_a^j$. Each equation defining a variable corresponds to a statement with the domain D_a^j and the function f^j. Each statement has an identity write relation and one read relation for each r_i: $\{ \langle \vec{i}, r_i(\vec{i}) \rangle \mid \vec{i} \in D_a^j \}$.

2.3 Scheduling

A *schedule* assigns a point in time and space (parallel processing unit) to each statement instance. A large amount of research into polyhedral optimization focuses on finding an optimal schedule that improves data locality and exposes parallelism. When the source program is given in a sequential form—e.g., static affine nested loop programs (SANLP)—it contains an inherent schedule; the goal of polyhedral transformations is to find a better one. In contrast, applicative languages like recurrence equations do not define a complete order of execution and the role of the compiler is to find one.

The earliest approaches search for two distinct affine functions—the *allocation* function (space) and the *timing* function (e.g., Rajopadhye and Fujimoto (1990)). It has been shown that a valid one-dimensional affine timing function does not exist for every program in the polyhedral model and a more general multi-dimensional time mapping has been proposed (Feautrier 1992a, 1992b). A popular approach used in the Pluto optimizer (Bondhugula et al. 2008b) finds an abstract multi-dimensional affine mapping in a single optimization framework; any dimension that does not carry dependencies can be interpreted as a distribution in space and others as a distribution in time. We use a generic definition of a schedule that accommodates all these approaches.

Definition 2.6 (Polyhedral Schedule). A polyhedral schedule Φ is a set of multi-dimensional functions $\phi_s : \mathbb{Z}^d \to \mathbb{Z}^n$. There is one ϕ_s for each statement s and each of them maps statement instances into the same space $\mathbb{Z}^n$. Viewed as a subset of $\mathbb{Z}^{d+n}$, ϕ_s is a polyhedron or a projection of one. The lexicographical order of points in $\mathbb{Z}^n$ is denoted by $\prec$ and defined as

$$\langle i_1, i_2, \ldots i_n \rangle \prec \langle j_1, j_2, \ldots j_n \rangle \iff i_1 < j_1 \vee (i_1 = j_1 \wedge \langle i_2, \ldots i_n \rangle \prec \langle j_2, \ldots j_n \rangle).$$

Any dimension may be interpreted as time or space. A point $\vec{i}$ is executed before a point $\vec{j}$ if $\vec{i} \prec \vec{j}$ and the first dimension in which they differ is a time dimension.

Definition 2.7 (Affine schedule direction and hyperplane). Consider an affine schedule where each output coordinate is defined as $\vec{i} \mapsto \vec{h} \cdot \vec{i} + h_0$. We call $\vec{h}$ a *scheduling direction*. The set of inputs $\vec{i}$ with the same value of $\vec{h} \cdot \vec{i}$ forms a *hyperplane*.

When searching for a schedule, a compiler is restricted by data dependencies between statement instances. In case of single-assignment languages, the only constraint is that an array value is written before it is first read (also known as a read-after-write dependence).

Definition 2.8 (Dependence relation). Given a write relation W and a read relation R, there is a dependence relation:

$$P = \{ \langle \vec{i}, \vec{j} \rangle \mid \exists \vec{a} : \langle \vec{i}, \vec{a} \rangle \in W \wedge \langle \vec{j}, \vec{a} \rangle \in R \}.$$

Definition 2.9 (Valid schedule). A schedule is *valid* when it *satisfies* all dependencies between statements. Let P be a dependence relation between statements s_1 and s_2. It is satisfied when

$$\forall \langle \vec{i}, \vec{j} \rangle \in P : \ \phi_{s_1}(\vec{i}) \prec \phi_{s_2}(\vec{j}).$$

Definition 2.10 (Dependence distance). Let an instance $\vec{j}$ of statement s_2 depend on an instance $\vec{i}$ of s_1. Then, $\phi_{s_2}(\vec{j}) - \phi_{s_1}(\vec{i})$ is a *dependence distance*.

Tiling is an important optimization technique. It refers to partitioning statement instances into regularly shaped groups and scheduling members of each group close together in time or space. Data locality is improved when grouped instances reuse the same data, which can improve cache utilization and minimize communication and synchronization between parallel processors. Since the origin of tiling in the polyhedral model in Irigoin and Triolet (1988), there has been a lot of research with diverse approaches. Sometimes, statement instances are first mapped into a common space, that space is tiled, and then the final space-time mapping is done (e.g., Dutta et al. (2006) and Hannig (2009)). In contrast, Griebl (2001) has proposed tiling after space-time mapping. The popular Pluto algorithm (Acharya and Bondhugula 2015; Bondhugula et al. 2008b) finds a multi-dimensional affine schedule such that simple rectangular tiling of the schedule range is possible, and the resulting tile indices can directly be interpreted as a distribution across space or time. More complex tile shapes and overlapped tiles have also been proposed (Krishnamoorthy et al. 2007).

This work proposes a technique called *periodic schedule tiling*, which is a one-dimensional tiling of the range of a multi-dimensional schedule (precise definition to follow). Its purpose is to partition an infinite schedule into a periodic sequence of finite parts. It is intended to be applied *after* space-time mapping and tiling for performance. It can be combined with a variety of scheduling and tiling approaches as long as they result in a schedule conforming to our Definition 2.6. In particular, our technique is compatible with schedules produced by the Pluto algorithm, which is also the basis for a number of other production and research compilers and optimizers: LLVM/Polly (Grosser et al. 2012), GCC/Graphite (Trifunovic et al. 2010), and PPCG (Verdoolaege et al. 2013).

2.4 Storage Allocation and Code Generation

The purpose of storage allocation is to map each element of an array in the polyhedral model to a memory location where it is stored. The amount of storage may be optimized by storing multiple elements into the same location. Such optimization can be classified as intra-array optimization (a storage location hosts elements from a single array) or inter-array optimization (a storage location hosts elements from multiple arrays). Since arrays in stream processing may be infinite, we are particularly concerned with intra-array optimization that allocates a finite buffer for each infinite array. We denote an intra-array optimizing storage function for an array a by γ_a.

Storage optimization is constrained by data dependencies and the schedule. Specifically, two elements can not be stored in the same location if they are live at the same time, according to the schedule. An element is live between the time it is written and the time it is last read. A pair of elements that are live at the same time represent a *storage conflict*. In intra-array optimization, this is denoted by $\vec{i} \bowtie \vec{j}$, where $\vec{i}$ and $\vec{j}$ are indices of two elements from the same array. A storage function γ_a is valid when it *satisfies* all conflicts, that is

$$\forall \vec{i} \in D_a, \forall \vec{j} \in D_a : \quad \vec{i} \bowtie \vec{j} \Rightarrow \gamma_a(\vec{i}) \neq \gamma_a(\vec{j}).$$

A popular class of intra-array storage optimizations is the class of *modular mappings*. A modular mapping characterized by a tuple $\langle M, \vec{e} \rangle$ is a storage function $\gamma_a(\vec{i}) = M\vec{i} \bmod \vec{e}$, where M represents an affine mapping and $\vec{e}$ the final multi-dimensional storage size. Modular mappings have

been studied extensively by Darte et al. (2003). The so-called *successive modulo technique* (SM) introduced by Lefebvre and Feautrier (1998) finds a suitable $\vec{e}$ when M is identity or otherwise given. We paraphrase its definition as given in Bhaskaracharya et al. (2016):

Definition 2.11 (Successive Modulo Technique). An algorithm that finds a valid storage size $\vec{e}$ for a modular mapping with a given M, defined as follows. Given a conflict $\vec{i} \bowtie \vec{j}$, let $|M\vec{i} - M\vec{j}|$ be a *conflict distance*. Starting with a set of conflicts, set the first component of the storage size $\vec{e}$ as the maximum distance of any conflict in the corresponding dimension plus 1. Remove all conflicts from the conflict set that have a distance larger than 0 in that dimension, since they are guaranteed to be satisfied. Repeat this for each following dimension.

More recently, Bhaskaracharya et al. (2016) proposed a technique called SMO that finds an M with the minimum number of storage dimensions as the primary objective. In combination with SM, fewer dimensions often result in a smaller total storage size.

Code generation is the task of converting a polyhedral model of a program with a schedule and storage optimization functions into imperative code. Using algorithms such as (Grosser et al. 2015; Quilleré et al. 2000), an abstract syntax tree (AST) is generated, which contains loops and conditional statements that visit each point in a polyhedral schedule and execute the associated statement instances. The induction variables (iterators) of the loops represent coordinates of the schedule points, and they are ultimately mapped to array indices through the storage optimization functions. We name this a *Polyhedral AST*:

Definition 2.12 (Polyhedral AST). A Polyhedral AST is an imperative AST generated from a polyhedral model with a given schedule and storage mapping.

This article extends the known AST generation techniques to avoid generating loops with unbounded iterators for streaming programs with unbounded array and statement domains.

3 PROBLEM STATEMENT

Before we define the problem, let us define the following terms:

Definition 3.1 (Access Schedule). Let A be an access relation and ϕ a schedule for the same statement. Then the composition of ϕ with the converse of A is an *access schedule* ϕ_A:

$$\phi_A = \phi \circ A^T = \{ \langle \vec{j}, \vec{t} \rangle \mid \exists \vec{i} : \langle \vec{i}, \vec{j} \rangle \in A \wedge \langle \vec{i}, \vec{t} \rangle \in \phi \}.$$

Definition 3.2 (Productive Schedule). A schedule is *productive* if each point in its range has a finite number of lexicographically preceding points.

Definition 3.3 (Rate-consistent Schedule). A schedule is *rate-consistent* if all unbounded access schedules for the same array have one and the same infinite direction.

Definition 3.4 (Admissible Input). An admissible input to our technique consists of a polyhedral model of a SARE and a schedule with the following properties:

- Array and statement domains have at most one infinite direction.
- Each statement instance only reads and writes a finite number of array elements, and there is an upper bound on this number across the entire program.
- The graph of statements given by the dependence relations (the *dependence graph*) is connected.
- The schedule is valid and rate-consistent.
- The conical hull of dependence distances does not contain $-\vec{r}$ for some ray $\vec{r}$ of the schedule range.

The problem addressed in this article is formally stated as follows:

Definition 3.5 (Polyhedral Compilation of a Stream Processing Program). Given an admissible input (Definition 3.4), find a transformation of the schedule and generate corresponding code such that:

(1) The schedule is productive.
(2) The program is executed in bounded, statically allocated memory.
(3) The generated code is free of unbounded loop iterators.

We now discuss and justify the admissible input (Definition 3.4). Arrays and statements with a single infinite direction are enough to model stream processing where time is the only infinite dimension. A program may also contain finite arrays that represent data independent of real time, and statements with bounded domains that write initial portions of infinite arrays. The restriction that each statement accesses a finite portion of arrays is obviously reasonable. It implies that all access relations have a single infinite direction.

We assume that program input and output is modeled using statements with side effects, just like in the example in Figure 1: an instance of an input (or output) statement transfers a finite portion of a stream from the world into an array (or from an array to the world). To ensure consistent side effects, we can enforce an execution order using a dependence relation, e.g., between each consecutive pair of statement instances. Note that by modeling input and output using statements rather than arrays, there is no so-called *live-in* and *live-out* arrays (arrays that are considered live during the entire program). Hence, storage optimization can operate uniformly on all arrays.

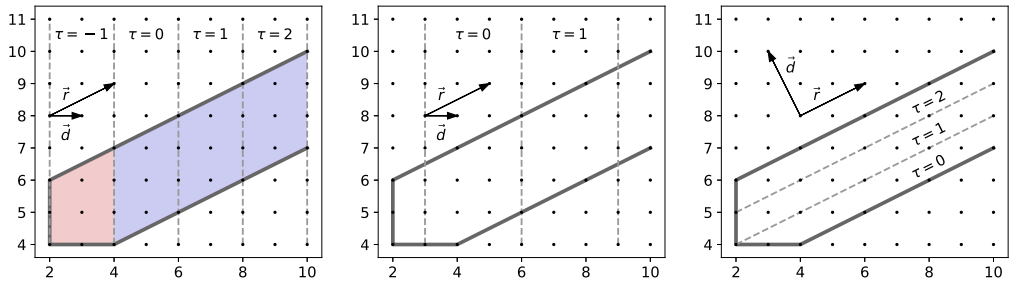
A rate-consistent schedule defined in 3.3 may be described informally and more intuitively as a schedule where any two dependent infinite statements iterate over each commonly accessed array "at the same rate." For example, consider two statements $a(i) = f(i)$ and $b(i) = a(2i)$ for $i \geq 0$. The schedule $\phi_a(i) = \phi_b(i) = i$ is not rate consistent: The infinite direction of the access schedule for the array a in the first statement is $\langle i, i \rangle$ and in the second statement is $\langle 2i, i \rangle$. In contrast, the schedule $\phi_a(i) = i, \phi_b(i) = 2i$ is rate-consistent. Some programs inherently prohibit rate-consistent schedules—for example, this equation with two accesses of array b: $a(i) = b(i) + b(2i)$, for all $i \geq 0$; regardless of the schedule, the infinite directions of the two access schedules are $\langle i, t \rangle$ and $\langle 2i, t \rangle$. The popular Pluto scheduling algorithm will likely find rate-consistent schedules. Namely, its objective is to minimize the upper bound on dependence distances. For two dependent statements, such a bound exists if and only if their schedule is rate-consistent. Hence, when the dependence graph of all the statements is connected, Pluto prioritizes globally rate-consistent schedules.[2]

A rate-consistent schedule and a connected dependence graph also imply that the ranges of all schedule statements have one and the same infinite direction. If the dependence graph is not connected, then the program may as well be split into separate programs, or artificial dependencies may be introduced to "synchronize" independent parts. The final restriction in Definition 3.4 reflects a reasonable expectation that dependencies do not predominantly point opposite to the infinite direction of the schedule.

4 PERIODIC SCHEDULE TILING

We present a transformation of an admissible schedule as defined in Definition 3.4, which ensures that the resulting schedule is productive, thus satisfying the requirement 1 in Definition 3.5. This method combined with our proposed storage allocation and code generation techniques also satisfies the other requirements. We name this method *periodic tiling*, since it partitions the infinite

[2] A proof and extended discussion regarding Pluto are published in the Addendum (Leben and Tzanetakis 2019).



(a) $\vec{d} = \langle 1, 0 \rangle$, $\mu = 4$, $\sigma = 2$. (b) $\vec{d} = \langle 1, 0 \rangle$, $\mu = 3$, $\sigma = 3$. (c) $\vec{d} = \langle -1, 2 \rangle$, $\mu = 4$, $\sigma = 2$.

Fig. 2. A periodic tiling 2(a) and two tilings that are not periodic: 2(b) and 2(c).

schedule into equally shaped finite tiles called *periods* with desirable properties. First, we define periodic tiling of any set in general in Section 4.1. Then, in Section 4.2, we define periodic schedule tiling in particular, prove some of its properties and provide an algorithm to find one. Finally, in Section 4.3, we discuss how periodic schedule tiling can be combined with tiling for performance. Without a loss of generality, we assume in the rest of the article that the infinite direction of a schedule is positive in all dimensions, to simplify notation.

4.1 Periodic Tiling

We define one-dimensional tiling similarly to (Irigoin and Triolet 1988) where it was defined as a map $\vec{i} \in \mathbb{Z}^d \mapsto \lfloor \vec{h} \cdot \vec{i} \rfloor$ with $\vec{h} \in \mathbb{Q}^d$. For the purpose of this work though, we expose the tile size as a separate parameter, and introduce the tile offset as an additional parameter:

Definition 4.1 (Tiling). A *tiling* of a set $S \subset \mathbb{Z}^d$ is a tuple $\langle \vec{d}, \mu, \sigma \rangle$. $\vec{d} \in \mathbb{Z}^d$ is a *tiling direction*, $\mu \in \mathbb{Z}$ is a *tiling offset*, and $\sigma \in \mathbb{Z}$ is a *tile size*. The function $\text{tile}(\vec{i}) = \lfloor (\vec{d} \cdot \vec{i} - \mu)/\sigma \rfloor$ assigns a *tile index* τ to each point in S. The set of all points with equal τ is considered a *tile* and denoted by T_τ.

Periodic tiling is a particular kind of tiling:

Definition 4.2 (Periodic Tiling). Consider a polyhedron P with a set of vertices V and a single infinite direction. Let $\vec{r}$ be any ray of P. A *periodic tiling* of P is a tiling $\langle \vec{d}, \mu, \sigma \rangle$ if there exists a ray $\vec{r}$ of P such that

$$(1)\ \vec{d} \cdot \vec{r} > 0 \qquad (2)\ \mu \geq \max \{ \vec{d} \cdot \vec{v} \mid \vec{v} \in V \} \qquad (3)\ \sigma = \vec{d} \cdot \vec{r}.$$

Similarly, if J is a projection, then a *periodic tiling* of $P' = J(P)$ is a tiling satisfying the conditions above with V replaced by $J(V)$ and with $\vec{r}$ replaced by a ray of P'. We also define a periodic tiling of a bounded polyhedron or its projection as a tiling satisfying only the condition 2 above. The tiles for $\tau < 0$ are called *prologue tiles* and those for $\tau \geq 0$ *periodic tiles*.

Corollary 4.3. *Following from Definitions 2.4 and 4.2: Let $\langle \vec{d}, \mu, \sigma \rangle$ be a periodic tiling with the smallest μ and σ for a given $\vec{d}$. Then $\langle \vec{d}, \mu', \sigma' \rangle$ is a periodic tiling if and only if it has an equal or larger offset $\mu' \geq \mu$ and an integer multiple size $\sigma' = k\sigma$, $k \in \mathbb{Z}^+$.*

Figure 2 depicts three different tilings of a polyhedron. Figure 2(a) is a periodic tiling, since (1) the tiling direction $\vec{d}$ is not perpendicular to the smallest ray of the polyhedron $\vec{r}$, (2) the offset μ is such that the vertex $\langle 4, 4 \rangle$—which is furthest in the tiling direction—lies at the beginning of the the tile $\tau = 0$, and (3) the size σ is an integer multiple of $\vec{d} \cdot \vec{r} = 2$. Prologue tiles are colored red, and

periodic tiles blue. Figure 2(b) is not a periodic tiling: μ is too small and σ is not an integer multiple of $\vec{d} \cdot \vec{r}$. 2(c) is also not a periodic tiling: $\vec{d}$ is perpendicular to $\vec{r}$, which generates unbounded tiles.

We turn the reader's attention to a number of properties of periodic tilings that we deem evident, although their mathematical proof is beyond the scope of this article. As shown later in the article, these properties turn out to be useful in solving the problem at the focus of this work:

Lemma 4.4. *A periodic tiling has the following properties:*

- *There is a finite number of prologue tiles, and zero or an infinite number of periodic tiles.*
- *Each tile is finite and each periodic tile has the same number of elements.*
- $\vec{i} \mapsto \vec{i} + \vec{r}$ *is an isomorphism between each pair of consecutive periodic tiles* T_τ *and* $T_{\tau+1}$*, which preserves the lexicographical order relation* $\prec$*. We call* $\vec{r}$ *the* tile distance.

4.2 Periodic Schedule Tiling

Definition 4.5 (Periodic Schedule Tiling). Let a schedule Φ be a set of statement schedules $\phi_s : D_s \to \mathbb{Z}^n$. Then, $\langle \vec{d} \in \mathbb{Z}^n, \mu, \sigma \rangle$ is a *periodic schedule tiling* if it is a periodic tiling of the range of each schedule ϕ_s, and additionally, if each related access schedule has a ray $\langle \vec{o}, \vec{r} \rangle$ where $\vec{r}$ is the tile distance. Let $\text{tile}(\vec{t})$ be the tile index according to a periodic schedule tiling. Then, a *periodically tiled schedule* is

$$\{ (\langle t_0, t_1, \ldots t_n \rangle \mapsto \langle \text{tile}(\vec{t}), t_0, t_1, \ldots t_n \rangle) \circ \phi_s \mid \phi_s \in \Phi \}.$$

A periodic tiling can turn a valid schedule into an invalid schedule (Definition 2.9). Due to (Irigoin and Triolet 1988), we have the following sufficient condition for validity:

Lemma 4.6. *A sufficient condition for validity of a periodically tiled schedule is:* $\vec{d} \cdot \vec{\delta} \geq 0$ *for all dependence distances* $\vec{\delta}$*.*

Theorem 4.7 [Existence of Periodic Schedule Tiling]. *There exists a valid periodic schedule tiling for any admissible input.*

Proof. A suitable tiling direction $\vec{d}$ must satisfy both Definition 4.2 and Lemma 4.6. To satisfy the former, $\vec{d}$ must be in the open halfspace $H(\vec{r}) = \{ \vec{x} \mid \vec{x} \cdot \vec{r} > 0 \}$ for some ray $\vec{r}$ of the schedule range. To satisfy the latter, $\vec{d}$ must be in the dual cone C^* of the conical hull C of dependence distances. There exists $\vec{d}$ satisfying both conditions unless $C^* \cap H(\vec{r})$ is empty. C is contained in the halfspace $\{ \langle x_1, \ldots \rangle \mid x_1 \geq 0 \}$, since only in that case all dependencies are satisfied. Therefore, $C^* \cap H(\vec{r})$ is empty only if C contains $-\vec{r}$. However, the set of admissible inputs excludes such a case.

A suitable tile offset μ is simply $\max\{\vec{d} \cdot \vec{v} \mid \vec{v} \in V\}$, where V is the union of vertices of all $\text{ran}(\phi_s)$.

The following proves the existence of a suitable tile size σ. All ranges of statement schedules have the same infinite direction, so there exists a vector $\vec{u}$ such that the ray of any schedule range is $k\vec{u}$ for some $k \in \mathbb{Z}^+$. Now, consider the set of all access schedules ϕ_{A_i} where $i \in [1, N]$, and each has a ray $\langle \vec{o}_i, \vec{r}_i \rangle$. Note that $\vec{r}_i$ must also be a ray of $\text{ran}(\phi_A)$, which is equal to some $\text{ran}(\phi_s)$ and so $\vec{r}_i = k_i \vec{u}$ for some $k_i \in \mathbb{Z}^+$. Let $\hat{k} = \text{lcm}_{i=1}^N k_i$ (lcm stands for least common multiple). Then, each access schedule has a ray $\langle (\hat{k}/k_i)\vec{o}_i, \hat{k}\vec{u} \rangle$, and $\hat{k}\vec{u}$ is also a ray of the schedule range. Therefore, $\sigma = \vec{d} \cdot (\hat{k}\vec{u})$ is the tile size of a periodic schedule tiling. This implies $\sigma = \text{lcm}_{i=1}^N (\vec{d} \cdot (k_i \vec{u})) = \text{lcm}_{i=1}^N (\vec{d} \cdot \vec{r}_i)$ where $\langle \vec{o}_i, \vec{r}_i \rangle$ is an access schedule ray. □

Figure 3 shows an example program with a periodically tiled schedule. This is a typical stencil program, except that it has an infinite number of steps $n \in [0, \infty)$. At every step $n > 0$, the statement s_5 computes an array of values $u(n, i)$ for $0 \leq i < N$. Each of these values is computed from

Statement	Effect	Domain
$s_1(n)$:	$x(n) \leftarrow \text{input}(n)$	$0 \le n$
$s_2(n)$:	$u(n, 0) \leftarrow 0$	$0 \le n$
$s_3(n)$:	$u(n, N-1) \leftarrow 0$	$0 \le n$
$s_4(i)$:	$u(0, i) \leftarrow x(0)$	$0 < i < N-1$
$s_5(n, i)$:	$u(n, i) \leftarrow x(n) + u(n-1, i)$ $+ u(n-1, i-1) + u(n-1, i+1)$	$0 < n \wedge 0 < i < N-1$
$s_6(m)$:	$y(m) \leftarrow u(2m, K)$ $\text{output}(m) \leftarrow y(m)$	$0 \le m \wedge K = \lfloor N/2 \rfloor$

(a) Program on three arrays $x(n)$, $y(m)$ and $u(n, i)$ where $0 < n$, $0 < m$ and $0 \le i < N$.

$$\phi_{s_1}(n) = \langle n, n+1, 1 \rangle$$
$$\phi_{s_2}(n) = \langle n+1, n+2, 2 \rangle$$
$$\phi_{s_3}(n) = \langle n+1, n+N-1, 0 \rangle$$
$$\phi_{s_4}(0, i) = \langle 0, i, 4 \rangle$$
$$\phi_{s_5}(n, i) = \langle n, n+i, 3 \rangle$$
$$\phi_{s_6}(m) = \langle 2m, 2m+K, 5 \rangle$$

(b) Φ = A schedule computed by ISL.

$$\phi_{s_1}(n) = \langle \lfloor (n-1)/2 \rfloor, n, n+1, 1 \rangle$$
$$\phi_{s_2}(n) = \langle \lfloor n/2 \rfloor, n+1, n+2, 2 \rangle$$
$$\phi_{s_3}(n) = \langle \lfloor n/2 \rfloor, n+1, n+N-1, 0 \rangle$$
$$\phi_{s_4}(0, i) = \langle -1, 0, i, 4 \rangle$$
$$\phi_{s_5}(n, i) = \langle \lfloor (n-1)/2 \rfloor, n, n+i, 3 \rangle$$
$$\phi_{s_6}(m) = \langle m-1, 2m, 2m+K, 5 \rangle$$

(c) $\Phi' = \Phi$ periodically tiled with $\vec{d} = \langle 1, 0, 0 \rangle$, $\mu = 1$, $\sigma = 2$

Schedule	Vertices	Access Schedule	Ray
$\text{ran}(\phi_{s_4})$	$\langle 0, 1, 4 \rangle$, $\langle 0, N-2, 4 \rangle$	$s_4(i)$ writes $u(0, i)$: $\{\langle 0, i \mid 0, i, 4 \rangle \mid 0 < i < N-1\}$	None.
$\text{ran}(\phi_{s_5})$	$\langle 1, 2, 3 \rangle$, $\langle 1, N-1, 3 \rangle$	$s_5(n, i)$ reads $u(n-1, i)$: $\{\langle n, i \mid 1+n, 1+n+i, 3 \rangle \mid 0 \le n \wedge 0 < i < N-1\}$	$\langle 1, 0 \mid 1, 1, 0 \rangle$
$\text{ran}(\phi_{s_6})$	$\langle 0, K, 5 \rangle$	$s_6(m)$ reads $u(2m, K)$: $\{\langle m, K \mid m, m+K, 5 \rangle \mid \exists e : m = 2e \ \wedge \ 0 \le m\}$	$\langle 2, 0 \mid 2, 2, 0 \rangle$

(d) A few of the statement schedules and their vertices, a few of the access schedules and their rays.

Fig. 3. Example of periodically tiled schedule.

an input value $x(n)$ obtained by statement s_1 and values of u at previous steps. Each instance of statement s_6 samples and outputs a value from u. To make the example less trivial, s_6 happens only at every second step. For consistency with other examples, the output values are stored in the intermediate array y, but the store and the output are modeled as a single statement for brevity. Statements s_2, s_3 and s_4 initialize u at boundaries. s_4 is the only statement with a bounded domain. An initial schedule computed using ISL is shown in Figure 3(b). This schedule undergoes periodic tiling, resulting in the schedule in Figure 3(c). To help illustrate the procedure, vertices of some of the schedules and rays of some of the access schedules are given in Figure 3(d). Figure 4 graphically presents the ranges of the original statement schedules, their rays, and the tiles of the periodic tiling. The periodic tiling direction $\vec{d} = \langle 1, 0, 0 \rangle$ is chosen, because it is not perpendicular to the rays. The tile offset $\mu = 1$ is the maximum of $\vec{d} \cdot \vec{v}$ for vertices v of schedule ranges. The tile size $\sigma = 2$ is the least common multiple of $\vec{d} \cdot \vec{r}$ for all rays $\langle \vec{o}, \vec{r} \rangle$ of access schedules.

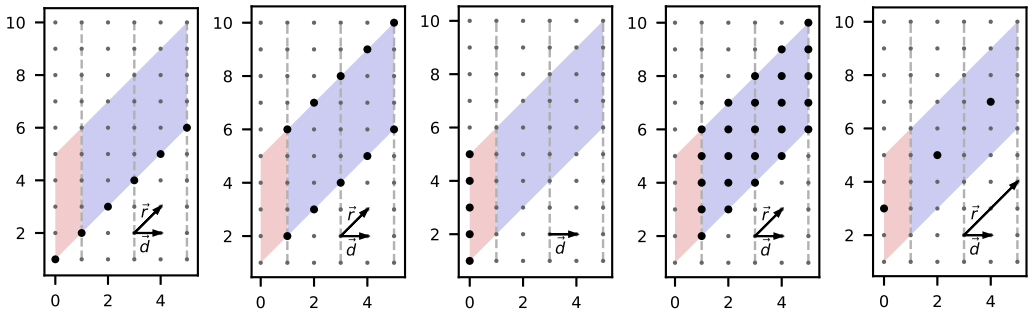


Fig. 4. First two dimensions of schedule Φ introduced in Figure 3, with $N = 7$. Dashed lines indicate tile boundaries in periodically tiled schedule Φ'. The prologue tile is red and periodic tiles are blue. Each subfigure highlights schedule for a particular statement using bold dots; from left to right: ϕ_{s_1}, $(\phi_{s_2} \cup \phi_{s_3})$, ϕ_{s_4}, ϕ_{s_5}, ϕ_{s_6}.

ALGORITHM 1: Find smallest periodic tiling

```
1:  procedure PeriodicTiling(𝒜, Φ, C)
2:      d⃗ ← TilingDirection(Φ, C)
3:      μ ← 0, σ ← 1
4:      for each A_{s,a} in 𝒜 and φ_s in Φ do
5:          φ_a ← φ_s ∘ A^T_{s,a}
6:          μ_a ← TileOffset(φ_s, d⃗)
7:          μ ← max(μ, μ_a)
8:          if ¬ IsBounded(φ_a) then
9:              σ_a ← TileSize(φ_a)
10:             σ ← lcm(σ, σ_a)
11:         end if
12:     end for
13:     return ⟨d⃗, μ, σ⟩
14: end procedure
15: procedure TilingDirection(Φ, C)
16:     r⃗ ← SmallestRay(ran(φ_s)) for any φ_s ∈ Φ
17:     Return any d⃗ ∈ C* ∩ H(r⃗), preferably a standard basis or one with smallest L¹-norm.
18: end procedure
19: procedure TileOffset(φ_s, d⃗)
20:     V ← Vertices(ran(φ_s))
21:     μ ← 0
22:     for each t⃗ ∈ V do
23:         μ ← max(μ, d⃗ · t⃗)
24:     end for
25:     return μ
26: end procedure
27: procedure TileSize(φ_a, d⃗)
28:     ⟨i⃗, t⃗⟩ ← SmallestRay(φ_a)
29:     return d⃗ · t⃗
30: end procedure
```

Algorithm 1 finds a periodic schedule tiling given a set $\mathcal{A}$ of access relations $A_{s,a}$ (between a statement s and array a), a set Φ of schedules ϕ_s (one for each statement s), and a set of conflict distances C. The algorithm includes a heuristic to find a valid direction with small coordinates, which is likely to result in simpler code. It finds the smallest tile offset and size for this direction. Although finding optimal parameters is outside the scope of this article, Section 4.3 discusses how the tiling direction, offset and size relate to various performance objectives. The algorithm includes a few auxiliary procedures. SmallestRay returns the smallest ray of an integer set with a single infinite direction. IsBounded returns whether an integer set is bounded. Vertices returns all vertices of an integer set; in case of a projection of a polyhedron, it returns the projections of the vertices of the polyhedron. Integer set relations are treated as integer sets in the combined dimensions of the domain and range. These operations can be easily implemented using the Integer Set Library (ISL).

We now prove that periodic schedule tiling satisfies the requirement 1 in Definition 3.5:

Theorem 4.8. *A periodically tiled schedule is productive.*

Proof. According to the definition of periodically tiled schedule Definition (4.5) and the schedule execution order (see Section 2.3), and assuming that the infinite direction of the schedule is positive in all dimensions, an entire tile T_τ is executed after the tile $T_{\tau-1}$. Further, a finite number of non-empty tiles precedes any tile, and each tile has a finite number of points, as stated in Lemma 4.4. Therefore, each point is preceded by a finite number of points and the schedule is productive. □

We prove a couple other properties of periodically tiled schedules that support the proposed storage allocation (Section 5.1) and code generation techniques (Section 5.2).

Lemma 4.9. *Consider a pair of corresponding schedule points $\vec{t_1}$ and $\vec{t_2}$ from two periodic tiles, according to the tile isomorphism in Lemma 4.4. Also consider the converse of an access schedule ϕ_A^T and the sets of array accesses $\phi_A^T(\vec{t_1})$ and $\phi_A^T(\vec{t_2})$. For each ϕ_A, there exists $\vec{o}$ such that $\vec{a} \mapsto \vec{a} + \vec{o}$ is an isomorphism for each such pair of corresponding array access sets.*

Proof. We know from Lemma 4.4 that each pair of corresponding schedule points has a distance $\vec{r}$. So, the above is true if there exists $\vec{o}$ such that $\langle \vec{a}, \vec{t} \rangle \in \phi_A \Rightarrow \langle \vec{a}, \vec{t} \rangle + \langle \vec{o}, \vec{r} \rangle \in \phi_A$. This is indeed true, since there exists $\langle \vec{o}, \vec{r} \rangle$, which is a ray of ϕ_A (from Definition 4.5) and it follows from Definition 2.4. □

Lemma 4.10. *All access relations of an array have a common distance $\vec{o}$ described in Lemma 4.9.*

Proof. Let $\langle \vec{o}_i, \vec{r} \rangle$ and $\langle \vec{o}_j, \vec{r} \rangle$ be the rays of two access schedules of an array, with $\vec{r}$ the tile distance in a periodic schedule tiling. In an admissible input, these two rays must be equivalent (Definition 3.3). So, $\langle \vec{o}_i, \vec{r} \rangle = k\langle \vec{o}_j, \vec{r} \rangle$, which is only true if $k = 1$ and $\vec{o}_i = \vec{o}_j$. □

4.3 Combining Periodic Tiling with Tiling for Performance

The goal of periodic schedule tiling is to ensure that a streaming program with an unbounded polyhedral model can be executed productively and in finite memory. This section demonstrates how periodic tiling can be combined with tiling that improves data locality and parallelism while also considering the effect on input-output latency—an important aspect of stream processing systems. It is sufficient for the purpose of this article to show that simple heuristics for choosing periodic tiling parameters allow exploiting existing polyhedral optimization techniques for streaming programs. Optimization of the periodic tiling parameters however is beyond the scope of this article.

We use the schedule in Figure 3(b) as an example. To tile for data locality, one usually chooses a tile size T and prefixes the schedule with inter-tile schedule dimensions: $\phi_{s_5}(n, i) = \langle \lfloor n/T \rfloor, \lfloor (n + i)/T \rfloor, n, n + i, 3 \rangle$, and similarly for other statements. This generates parallelogram tiles shown in Figure 5(a) (with $T = 4$). This tiled schedule can be further periodically tiled, for example, with $\vec{d} = \langle 1, 0, 0, 0, 0 \rangle$, $\mu = 1$, and $\sigma = 1$, producing a prologue and a period as shown in Figure 5(a). Note that choosing a direction $\vec{d}$, which is non-zero only in the inter-tile schedule dimensions ensures that the prologue and periodic tiles consist only of entire original tiles (they do not split any tiles).

Although the above approach improves data locality within tiles, it does not support any parallel execution of tiles, since each inter-tile schedule dimension carries some data dependencies. This is a known problem, and Bondhugula et al. (2008b) provide a simple solution: prefixing the tiled schedule with a dimension equal to the sum of all the inter-tile dimensions. In our example, this yields $\phi_{s_5}(n, i) = \langle \lfloor n/T \rfloor + \lfloor (n + i)/T \rfloor, \lfloor n/T \rfloor, \lfloor (n + i)/T \rfloor, n, n + i, 3 \rangle$. This admits a periodic tiling with $\vec{d} = \langle 1, 0, 0, 0, 0, 0 \rangle$, $\mu = 6$, and $\sigma = 2$, which is shown in Figure 5(b). Within each period, we have two schedule hyperplanes in the direction that carries all dependencies, and so all subtiles on such a hyperplane can be executed in parallel (an example is marked with the thick black line).

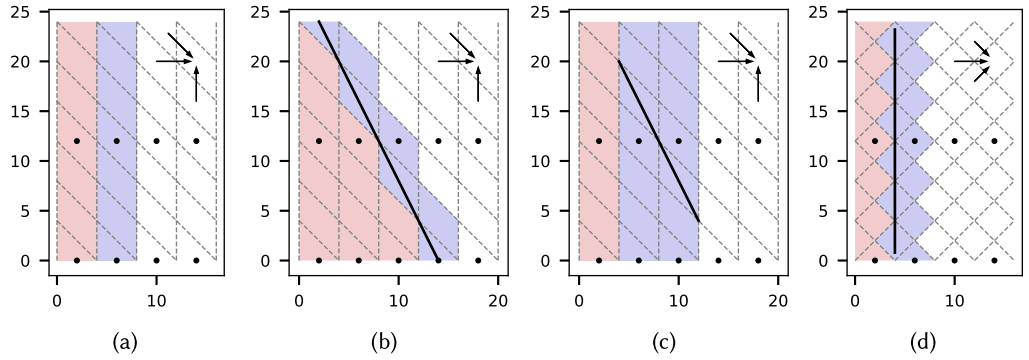


Fig. 5. Combination of periodic tiling and performance tiling for program in Figure 3 with $N = 24$. Horizontal axis represents $n = 2m$ and vertical i. Prologue tiles in red, the first periodic tile in blue. Bottom row of dots marks sub-tiles where input occurs, and top row marks sub-tiles where output occurs. Arrows depict dependencies between tiles. The bar connects a set of sub-tiles within a period that can execute in parallel.

There are alternative periodic tilings, though. Note that the schedule in Figure 5(b) increases input-output latency in comparison to Figure 5(a)—tiles that perform input are executed much earlier than the dependent output tiles. It is particularly problematic that the latency depends on the domain size (parameter N). This can be improved while retaining some tile parallelism using a periodic tiling $\vec{d} = \langle 0, 1, 0, 0, 0, 0 \rangle$, $\mu = 1$, and $\sigma > 1$, where σ controls the trade-off between latency and parallelism. This is shown in Figure 5(c) for $\sigma = 2$. An even better solution is possible with an altogether different schedule based on the "diamond tiling" presented in Bandishti et al. (2012). For example, a schedule with $\phi_{s_5}(n, i) = \langle \lfloor (n + i)/T \rfloor, \lfloor (n - i)/T \rfloor, n + i, n - i, c \rangle$ (and similarly for other statements) produces diamond-shaped tiles as shown in Figure 5(d). The direction $\langle 1, 1, 0, 0, 0 \rangle$ now carries all the inter-tile dependencies. Therefore, we can choose this as the periodic tiling direction $\vec{d}$ with $\mu = 1$ and $\sigma = 2$. This allows tile parallelism within a period as well as minimal latency.

5 STORAGE ALLOCATION AND CODE GENERATION

5.1 Finite Storage Using Modular Mapping

Storage optimization is critical when modeling streams as infinite arrays: a trivial storage allocation mapping each array element into a unique memory location would require an infinite amount of memory. We prove that finite storage is achievable for periodically tiled schedules using well-known intra-array storage optimization techniques, thus satisfying the requirement 2 in Definition 3.5:

Theorem 5.1. *The successive modulo technique (SM) yields a modular mapping $\langle M, \vec{e} \rangle$ with a finite storage size $\vec{e}$ for any periodically tiled rate-consistent schedule and any M.*

Proof. Recall Definition 2.11 of SM and note that it yields finite storage as long as all storage conflict distances are finite. There may be an infinite conflict distance only if: (1) an element of an array is live for an unbounded amount of time, and (2) an unbounded number of other elements are accessed during that time. Recall the constant distance of array accesses across schedule tiles (Lemma 4.10). If the offset is non-zero, then no element is live for an unbounded amount of time, so the first necessary condition is not true. If the offset is zero, then only a bounded number of elements are accessed during the entire schedule, so the second necessary condition is not true. □

5.2 Periodic Polyhedral AST

As demonstrated by the example in the Introduction, when applying existing techniques for generation of a Polyhedral AST to a periodically tiled schedule, the resulting code can contain unbounded loops. To address this, we propose to generate code in a form called *Periodic Polyhedral AST* without unbounded quantities. This form consists of traditional Polyhedral AST parts generated from parts of the periodically tiled schedule using existing techniques, with a few modifications:

Definition 5.2 (Periodic Polyhedral AST). An AST that consists of an infinite sequence $(q, p, p, p, \ldots)$, where q and p are Polyhedral ASTs generated from a polyhedral model with a periodically tiled schedule and modulo contracted buffers. Specifically, q is generated from the prologue tiles of the schedule, and p is generated from the first periodic tile. In addition, p has the following properties:

- Each array access $a(\vec{i})$ is mapped into a buffer access $b((\vec{i} + \vec{d}) \bmod \vec{e})$, where $\vec{e}$ is the buffer size and $\vec{d}$ represents an offset for all accesses to this buffer.
- Initially, $\vec{d} = 0$, and p updates $\vec{d}$ to a new value $(\vec{d} + \vec{o}) \bmod \vec{e}$, where $\vec{o}$ is the inter-tile distance of buffer accesses according to Lemma 4.10.

A Periodic Polyhedral AST represents a solution to the final requirement in Definition 3.5: generating code without unbounded quantities. To support this, we prove the following:

Theorem 5.3. *A Periodic Polyhedral AST is semantically equivalent to a Polyhedral AST of a periodically tiled schedule.*

Proof. The isomorphism of tiles in a periodic schedule tiling that preserves the order relation (Lemma 4.4) proves the semantic equivalence except for the difference in the mapping of array accesses to buffer accesses in the periods p of the Periodic Polyhedral AST. We prove that each repetition τ of p is equivalent to the part of the Polyhedral AST corresponding to the tile τ as follows. Let $a(\vec{i})$ be an access in the first periodic tile. Due to Lemma 4.9, each following periodic tile τ has a corresponding array access $a(\vec{i} + \tau\vec{o})$. In the Polyhedral AST, the corresponding buffer index is $(\vec{i} + \tau\vec{o}) \bmod \vec{e}$, and in the Periodic Polyhedral AST it is $(\vec{i} + \vec{d}_\tau) \bmod \vec{e}$, where $\vec{d}_\tau = \tau\vec{o} \bmod \vec{e}$. The buffer accesses in the Polyhedral AST and the Periodic Polyhedral AST are equivalent, since

$$(\vec{i} + \tau\vec{o}) \bmod \vec{e} = (\vec{i} + (\tau\vec{o} \bmod \vec{e})) \bmod \vec{e}. \qquad \square$$

Corollary 5.4. *Evident from Definition 5.2, a Periodic Polyhedral AST has no unbounded loop iterators.*

5.3 Buffer Performance Optimization

Our empirical evaluation shows that buffer index expressions can have a significant impact on performance. In addition to the intrinsic cost of the mathematical operations involved, some operations can also obstruct a compiler's ability to vectorize a loop. The general form of a buffer index, as described in Section 5.2, is rather complex and involves the costly modulo operation: $(\vec{i} + \vec{d}) \bmod \vec{e}$. Furthermore, arrays in stream processing are often accessed sparsely, for example, when a stream is processed in windows with a hop size larger than 1. This may result in $\vec{i}$ expanding to an expression that involves additional division and multiplication. This section presents techniques for simplification of index expressions. Most of these techniques are well known and used in practice, but we summarize them here and place them into the context of polyhedral compilation. In the rest of this section, i, d, o, and e represent individual components of the vectors $\vec{i}$, $\vec{d}$, $\vec{o}$, and $\vec{e}$.

Redundant Buffer Dimensions. If the buffer size in some dimension equals 1, then that dimension of the buffer can be removed, which simplifies indexing.

Redundancy of Modulo for Buffer Size. In any given loop, modulo may be redundant if $i + (\tau o \bmod e) < e$ for all i in the loop and all $\tau \geq 0$. If this condition is not satisfied, then we may be able to extend the buffer size e to satisfy it. If o is 0, then it can easily by satisfied by extending the buffer size to the maximum value of i.

Replacing Modulo with Bitmasking. When the buffer size e is a power of two, modulo can be replaced with bitmasking, since $x \bmod e = x \,\&\, (e - 1)$. If the minimal buffer size is not a power of two, then it can be extended to the next power of two. This increases the buffer size by no more than two times in a single dimension. If multiple dimensions are extended this way, then the increase can be as large as 2^n, where n is the number of extended dimensions.

Shifting Data within Buffer. We can shift data within a buffer by $-o$ at the end of each period, instead of updating the access offset d. By also extending the buffer size to the span of all access indices within a period, we get rid of both d and the modulo. The complexity of data shifting is in $O(N)$ where N is the number of elements reused across periods. This can be improved by further increasing buffer size and shifting data only every M periods. The buffer index then requires the offset d, but not the modulo. d is updated every period, while every M-th period it is reset to 0 and the data is moved by $-Mo$. The asymptotic complexity of data shifting is then in $O(N/M)$, which is in $O(1)$ when $M \geq N$.

Loop Invariant Code Motion. When accessing a multi-dimensional array in a multi-level loop nest, complex index expressions may be avoided in the innermost loop using loop-invariant code motion (hoisting). Although loop invariant code motion is a well-known optimization technique, our evaluation reveals that existing C++ compilers sometimes fail to move array index expressions even when it would have significant benefits.

6 EMPIRICAL EVALUATION

We integrate the proposed compilation techniques into a single framework, which is empirically evaluated with the goal of supporting two claims:

(1) Efficient executable code can be generated for stream processing applications in the form of recurrence equations and similar applicative languages.
(2) High-volume multi-dimensional stream processing applications in general benefit from optimizations enabled by polyhedral techniques.

We evaluate the framework on stream processing kernels exhibiting a variety of mathematical operations, data layouts and data dependence patterns. The input to our framework are the kernels expressed in the language Arrp (Leben 2016). The implementation resembles textbook definitions of algorithms using recurrence equations as close as possible. The output of our framework is C++ with OpenMP annotations, which is then translated into executable code using a general-purpose C++ compiler. We compare this to several alternative implementations of the same algorithms (see details in the following subsections):

- *Auto-optimized C++ (abbreviated as C++ AO)*: Hand-written C++ in a conventional form, compiled using the highest degree of automatic optimization in C++ compilers.
- *Hand-optimized C++ (abbreviated as C++ HO*: Manually optimized version of the C++ source code used in the auto-optimized C++.
- *StreamIt*: Implementation in the StreamIt language.

Unfortunately, due to limitations of the available StreamIt compiler related to the large problem sizes used in this evaluation, we were unable to achieve any parallelism in StreamIt programs, and we were only able to compile 4 out of 5 programs. In many cases, the compiler did not terminate within 1 hour or it ran out of memory.

All the data generated in our experiments as well as instructions and source code required to replicate the experiments are published in Leben (2019).

6.1 Algorithms

This section describes the evaluated algorithms. Each algorithm uses a scaling parameter N that controls the volume of data streamed through. The input is denoted by x and the output by y. Discrete time is denoted by n and is theoretically in the range $-\infty < n < \infty$, although implementations begin at $n = 0$. For brevity, we omit definitions of outputs at domain bounds.

filter-bank: A bank of N finite impulse response filters (FIR) of N-th order, operating on a one-dimensional input stream. b denotes a predefined array of coefficients.	$y[n,i] = \sum_{k=0}^{N-1} b[i,k]x[n-k], \quad 0 \le i < N$
max-filter: A bank of N max filters of N-th order. Each filter operates independently on one of the N input channels.	$y[n,i] = \max_{k=0}^{N-1} x[n-k,i], \quad 0 \le i < N$
autocorrelation (ac): Short-term autocorrelation of windows of $W = 5N$ samples with 3/4 window overlap, for lags $0 \le l < W$ samples:	$y[n,l] = \sum_{k=0}^{W-1} x[1/4Wn+k]x[1/4Wn+k+l]$
wave1d: FDTD scheme for 1-dimensional wave equation.[3] $b_0, b_1, \ldots$ denote predefined coefficients.	$u[n,i] = b_0u[n-2,i] + b_1u[n-1,i] + b_2(u[n-1,i-1] + u[n-1,i+1]) + x[n], \quad 0 \le i < N^2$ $y[n] = u[n,N/2]$
wave2d: FDTD scheme for 2-dimensional wave equation.[3] $b_0, b_1, \ldots$ denote predefined coefficients.	$u[n,i,j] = b_0u[n-2,i,j] + b_1u[n-1,i,j] + b_2\big(u[n-1,i-1,j] + u[n-1,i+1,j] + u[n-1,i,j-1] + u[n-1,i,j+1]\big) + x[n], \quad 0 \le i < N,\ 0 \le j < N$ $y[n] = u[n,N/2,N/2]$

6.2 Algorithm Implementation and Evaluation

6.2.1 Source Code. The Arrp, StreamIt and hand-written C++ source code is published in (Leben 2019). In addition to hand-written sources, C++ code also appears as the output of the Arrp and StreamIt compilers. In all cases, it has a structure described in Section 5.2: a prologue followed by a repeated execution of a period. While conventionally hand-written C++ is directly sent to a C++ compiler to evaluate automatic optimization, the same code is used as a starting point for hand-optimized C++. Rather than restructuring the entire program, we only modify the contents of the period function using loop transformations, OpenMP pragmas, and so on, while aiming for the highest throughput. This means the result is also the best we could hope for if we applied existing polyhedral techniques to the original C++ code.

In the polyhedral model of Arrp, we insert statements that write data from the outside world into input arrays x and send data from the output arrays y to the outside world, as described in

[3]Used, for example, in physical modeling of musical instruments (Bilbao 2009).

Table 1. Scheduling Directions and Tile Sizes for the Arrp Code. Commas Separate Values for Each Schedule Dimension

Algorithm	Scheduling Directions	Tile Sizes
filter-bank	$n + k,\ k,\ i$	128, 32, 64
max-filter	$n + k,\ k,\ i$	256, 64, ∞
ac	$1/4Nn + k,\ 1/4Nn,\ l$	$1/4N$, 1, 50
wave1d	$n,\ n + i$	256, 1024
wave2d	$n,\ n + i,\ n + j$	32, 32, 256

Section 3. In C++ code, these statements appear as calls to `input` and `output` functions, which transfer an element or array of elements (see Figure 1 as an example). Hand-written C++ and C++ generated by the StreamIt compiler follow the same pattern.

6.2.2 Scheduling and Tiling. After constructing a polyhedral model of the Arrp code, the Arrp compiler computes an initial schedule using the Integer Set Library (ISL) with a variation of the Pluto algorithm. The schedule is then traditionally tiled. We evaluated a large number of manually selected tile sizes in the range of 4 to 4,096 units in each schedule dimension, as well as leaving some dimensions untiled (tile size equals ∞). We report results for the tilings yielding the highest throughput. The autocorrelation algorithm is a special case where we choose a tile size based on the parameter N to align the tile boundaries with the input window boundaries. Table 1 lists the automatically selected scheduling directions and manually selected tile sizes. The scheduling directions refer to the index variables in the above algorithm definitions.

The tiled schedule is further subjected to *periodic tiling*, which allows extraction of a prologue and a period (see Section 5.2). For the filter-bank, max-filter, and autocorrelation algorithms, we simply choose the standard basis vector for the first dimension of the schedule as the direction for periodic tiling and use the smallest periodic tiling size and offset. In wave1d and wave2d though, data dependencies preclude parallel execution of sub-tiles using the default inter-tile schedule. This is alleviated with the approach depicted in Figure 5(c) and explained in Section 4.3.

We have experimented with tiling and other schedule modifications in hand-optimized C++, but found only limited benefits, due to the limitation of transformations to a single iteration of the period as described in Section 6.3. For auto-optimized C++, the highest degree of schedule transformations is enabled using C++ compiler options as described in Section 6.2.5. When compiling StreamIt, we have not found any options related to scheduling that would improve performance.

6.2.3 Parallelization and Vectorization. The Arrp compiler inserts OpenMP pragmas into generated C++ code to explicitly request loop parallelization and vectorization. Vectorization is requested on the innermost loops that carry no dependencies and parallelization on the outermost such loops. One exception is the autocorrelation algorithm. The outermost parallelizable loop in the code for period corresponds to the second scheduling direction ($1/4Nn$) and it has only 4 iterations—due to the windowed input processing with a hop size equal to 1/4 of a window. To increase parallelism, we parallelize the loop for to the third scheduling direction l, which also carries no dependencies but has a much larger number of iterations.

In hand-optimized C++ code, we insert OpenMP pragmas for explicit parallelization and vectorization. For auto-optimized C++, automatic parallelization and vectorization is enabled using C++ compiler options. See Section 6.2.5 for details. The StreamIt compiler was unable to compile the programs with parallelization enabled.

6.2.4 Storage Allocation and Buffer Implementation. The Arrp compiler contracts infinite arrays using modular mappings $\langle M, \vec{e} \rangle$ with an identity matrix M and storage size $\vec{e}$ determined using a

variation of the successive modulo technique. Array elements accessed by a group of statement instances executed in parallel are treated as storage conflicts.

Furthermore, each algorithm is evaluated using three different buffer implementations using various optimizations described in Section 5.3:

mod Each stream buffer is allocated with the minimal possible size and modulo is used in buffer indexing. The Arrp compiler avoids obvious modulo redundancies, although it does not attempt to find additional redundancies by modifying buffer sizes.
mask Modulo in buffer index expressions is replaced by bitmasking.
shift Modulo in buffer index expressions is avoided by shifting data within the buffer.

The Arrp compiler automatically generates a C++ implementation for the requested buffer type. Optionally, it also performs loop-invariant code motion on complex array indexing expressions (see Section 5.3). The auto-optimized C++ code is hand-written for each buffer type separately. In hand-optimized C++ code, we manually explore different buffer implementations and choose one that yields the highest throughput in each case.

6.2.5 Machine Code Generation. Machine code is generated from C++ using the Intel compiler version 19.0.1 with options `-O3 -fopenmp -fp-model fast=1` and the GNU compiler version 7.3.0 with options `-O3 -fopenmp -ffast-math`. These options enable automatic vectorization as well as support for explicit vectorization and parallelization using OpenMP. The Intel option `-fp-model fast=1` and GNU option `-ffast-math` trade consistency of floating point operations for speed and maximize the amount of vectorized code. Unlike stricter options, we find that these options yield a fair comparison between the two compilers in the evaluated programs. For auto-optimized C++, we enable the highest degree of automatic loop transformations and parallelization using additional options: with Intel compiler, we add `-parallel`; with GNU compiler, we add `-floop-nest-optimize`, `-floop-parallelize-all`, and `-ftree-parallelize-loops=n` (with `n` the parallelization factor).

6.2.6 Hardware. Evaluation is performed on a machine with a 6-core Intel Xeon E5-1650 v4 CPU with a 32Kb L1 cache, 256Kb L2 cache and 15Mb L3 cache. To increase repeatability, we disable frequency scaling (P states), idle states (C states), and hyperthreading. We keep the Turbo feature enabled, because we find that it increases performance consistently.

6.2.7 Measured Quantities. We measure the following quantities:

Storage Size is the total amount of memory allocated for program data.
Throughput is measured as the number of output data elements per unit of time. We measure elapsed time using the standard C++ facility `std::chrono::steady_clock`. For each algorithm, we measure the execution time of a number of periods, which add up to about 1s to ensure similar precision. Let P be the number of measured periods, d their total duration, and O the number of output elements per period. Throughput is then defined as $O \cdot P/d$.
Logical Latency expresses the amount of input elements consumed before an output element is produced. We define it as follows. Consider first only one-dimensional input and output streams. Assign indices $0, 1, 2, \ldots$ to consecutive input and output elements, and let r be the ratio of input elements consumed to output elements produced in a period of the program. Given an input index n and output index m, we call $n - \lfloor r \cdot m \rfloor$ the *offset* of this pair. The offset of a pair of entire input/output streams is the maximum offset of all the elementwise pairs where the output is produced after the input is consumed. For a particular implementation of an algorithm, the *latency* is the difference between its input-output offset and the minimal possible offset of any implementation. We apply this definition to multi-dimensional streams by modeling them as one-dimensional streams.

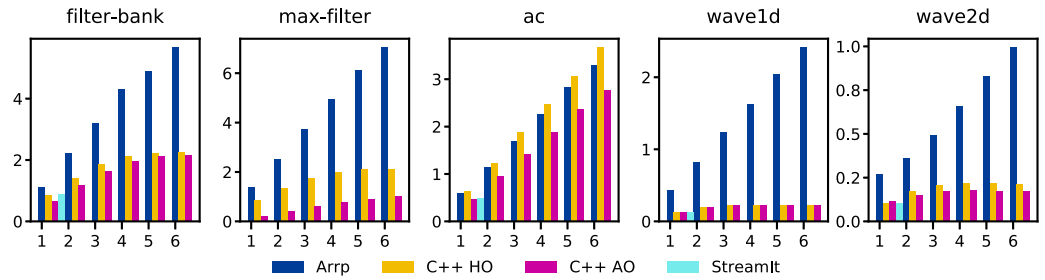


Fig. 6. Throughput (vertical, in output elements/μs for filter-bank, max-filter, ac, and output elements/ms for wave-1d and wave-2d) in relation to number of threads (horizontal). C++ compiler: Intel. Algorithm scale: $N = 2000$. Arrp code uses tile sizes in Table 1. Arrp and auto-optimized C++ use buffer type "mask" for all algorithms, except "shift" for ac. Explicit hoisting and vectorization is enabled for all Arrp sources except for ac. Some StreamIt results are missing due to limitations of the StreamIt compiler.

> For the particular algorithms used in the evaluation, this is simple: the stream domains are hyperrectangular and unbounded only in the first dimension, so we project them to this dimension.

6.3 Results

Figure 6 shows how throughput scales with the number of threads. For Arrp sources and auto-optimized C++ sources, we use the buffer type "mask" as a tradeoff between throughput and storage size. The exception is the "ac" algorithm where the buffer type "shift" is used, because it results in a significantly higher throughput. The effect of different buffer types is reported in more detail below. The StreamIt compiler was unable to compile the max-filter program and was only able to generate single-threaded programs. The figure shows results for the Intel C++ compiler; the GNU compiler gives essentially the same results, except slightly lower values overall and lacking parallelism in "C++ AO" variants. An extended figure is published in the Addendum (Leben and Tzanetakis 2019).

Analysis using Intel VTune Amplifier reveals that the performance of all algorithms except for autocorrelation is bound by the latency of accesses to main memory, shared between CPU cores. This is also supported by the significantly large storage size required by these algorithms compared to autocorrelation, as shown in Table 3 which is discussed in more detail later. Only the Arrp implementation manages to hide this latency; this is due to improved data cache utilization using higher-dimensional tiling. In hand-optimized C++ code, tiling is limited to the single period of the program and yields no improvement in the filter-bank, wave-1d and wave-2d algorithms, and only a small improvement in the max-filter algorithm. The auto-optimized C++ code faces similar limitations. Due to main-memory bus contention, throughput in these implementations scales only sub-linearly with parallelization, whereas Arrp implementations achieve linear scaling. The autocorrelation algorithm, however, is much less memory-bound, and so efficient parallelization is achieved both in auto-optimized code (by Intel compiler) and hand-optimized C++ code (using both Intel and GNU compilers). Here, the Arrp code still enjoys competitive performance.

We explore how buffer optimizations (described in Section 5.3) and their interaction with C++ compiler optimizations affect throughput of Arrp and auto-optimized C++ code. Table 2 shows the results for each combination of different buffer types, GNU or Intel C++ compiler, and in the case of Arrp sources, whether hoisting is applied to buffer index expressions (H) and whether innermost loops are explicitly vectorized using OpenMP (V). We present these parameters together due to

Table 2. Effect of Buffer Type, Hoisting (H) and Explicit Vectorization (V) on Throughput for Arrp and Auto-optimized C++ Implementations and Using Intel or GNU C++ Compiler

Algorithm	Buffer Type	C++ AO GNU	C++ AO Intel	Arrp GNU	Arrp GNU+V	Arrp GNU+H	Arrp GNU+H+V	Arrp Intel	Arrp Intel+V	Arrp Intel+H	Arrp Intel+H+V
filter-bank	mod	**0.36**	2.12	1.23	4.25	3.17	4.47	4.52	4.56	4.75	5.12
	mask	0.49	2.19	2.06	4.55	3.59	4.62	5.10	4.95	5.37	**5.63**
	shift	0.78	2.21	2.05	4.44	3.53	4.58	4.97	4.99	5.44	5.50
max-filter	mod	**0.13**	0.88	6.50	6.50	6.48	6.46	7.15	7.07	7.13	7.05
	mask	0.17	1.03	6.55	6.51	6.48	6.52	7.11	7.09	7.12	7.15
	shift	0.19	1.07	6.48	6.45	6.46	6.51	**7.22**	7.21	6.55	7.15
ac	mod	**0.05**	0.29	0.42	0.42	0.37	0.37	0.29	0.29	0.44	0.30
	mask	0.13	0.74	0.84	0.83	0.68	0.68	0.90	0.90	1.11	1.07
	shift	0.25	2.81	1.65	1.76	0.92	1.75	**3.29**	2.60	2.27	2.32
wave1d	mod	0.13	0.23	2.35	2.34	2.30	2.35	1.34	2.32	1.33	2.40
	mask	0.13	0.23	2.31	2.33	2.32	2.32	1.34	2.33	1.33	**2.41**
	shift	**0.11**	0.16	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
wave2d	mod	0.12	0.17	0.98	0.99	0.98	**0.99**	0.58	0.96	0.58	0.99
	mask	0.12	0.17	0.97	0.99	0.98	0.99	0.57	0.95	0.57	0.99
	shift	**0.10**	0.16	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Using algorithm scale $N = 2,000$, Arrp tile sizes in Table 1 and six threads. Throughput is in output elements/μs for filter-bank, max-filter, ac, and output elements/ms for wave-1d and wave-2d. The lowest and highest value for each algorithm is emphasized.

their interesting and complex interplay. The autocorrelation algorithm exhibits a trend of increased throughput when changing the buffer type from mod to mask and then to shift. The reason is that both Arrp and C++ implementations require a modulo in the innermost loop. Replacing it with bitmasking and data shifting progressively reduces the overhead. This trend is minimized or absent in other algorithms, because buffer indexing dependent on the innermost loop index has a small overhead. The wave-1d and wave-2d algorithms are not automatically vectorized by the Intel compiler, but explicit vectorization using OpenMP is beneficial. In the filter-bank algorithm compiler, manual hoisting of loop-invariant array index expressions has significant benefits with the GNU compiler. In the autocorrelation algorithm with shift buffers though, manual hoisting or vectorization significantly hinders the Intel compiler's ability to optimize.

Table 3 shows the storage size required by different algorithm implementations. In general, we see an increase between the mod, mask and shift buffer types. The shift buffer type makes storage size depend on tile size, and in the Arrp implementation of wave1d and wave2d algorithms it also depends on the parallelization factor (and hence period size, due to the scheduling described above). In the latter two cases, this results in an extreme increase of storage size. This is also the reason for omission of the related results from Table 2—the required amount of memory was not available on our test machine. In conclusion, the shift buffer type is not useful for all algorithms.

Table 4 lists logical input-output latencies for hand-written C++ code and Arrp code—the algorithmic complexity as well as the values. Hand-written C++ implementations enjoy the minimal possible latency, except in the case of autocorrelation, where an intuitive implementation increases

Table 3. Storage Size in Mb for Different Implementations and Buffer Types

Algorithm	C++ AO			C++ HO	Arrp		
	mod	mask	shift	best	mod	mask	shift
filter-bank	**30.5**	30.5	30.6	30.5	64.9	**95.0**	87.9
max-filter	**30.5**	31.3	61.1	31.3	38.3	66.4	**109.4**
ac	**0.23**	0.33	0.23	0.24	0.46	0.51	**1.37**
wave1d	91.6	122.1	183.1	91.6	**61.0**	61.1	**101,562.6**
wave2d	91.6	122.2	183.3	91.6	**61.1**	61.1	**94,821.3**

Using algorithm scale $N = 2{,}000$, Arrp tile sizes in Table 1 and supporting six threads. The lowest and highest value in each row are emphasized.

Table 4. Logical Latency (Complexity and Value)

Algorithm	C++ AO/HO		Arrp	
filter-bank	$O(1)$	0	$O(T)$	127
max-filter	$O(1)$	0	$O(T)$	255
ac	$O(1)$	1	$O(1)$	0
wave1d	$O(1)$	0	$O(PT) \cap O(N^2)$	1,537
wave2d	$O(1)$	0	$O(PT) \cap O(N)$	550

N is algorithm scale, T is tile size in first dimension, P is degree of thread parallelism. Values reported for $N = 2{,}000$, Arrp tile sizes in Table 1, and six threads.

the latency by 1 past the minimum. The latency in Arrp code has a dependence on tile size in the filter-bank and max-filter algorithms. In the wave-1d and wave-2d algorithms, Arrp code depends on the product of tile size and degree of thread parallelism, but it is bounded by the problem size. It is worth noting that in all evaluated cases the actual latency of Arrp implementations is only a fraction of the problem size N.

7 CONCLUSIONS

The proposed polyhedral scheduling technique called *periodic tiling* enables generation of efficient code with statically allocated memory for stream processing programs in the form of a system of affine recurrence equations (SARE). Periodic tiling integrates well with existing polyhedral scheduling techniques, which improves data locality and expose parallelism. Our experimental evaluation shows benefits for high-volume stream processing. By enabling tiling over time, our method improves parallel throughput scaling compared to optimizing a single period of a C++ implementation. On a 6-core Intel Xeon CPU, we observe speedups up to 10× (geometric mean 3.3×) over hand-optimized C++, and up to 10× (geometric mean 4.2×) over hand-written C++ optimized by the Intel C++ compiler. Although these benefits are achievable using simple heuristics to select periodic tiling parameters, automatically optimizing these parameters remains an interesting challenge. It also remains to expand our method to parameterized polyhedral models.

This work also emphasizes the importance of storage optimization for unbounded recurrence equations and proves the utility of existing storage optimization algorithms in combination with periodic tiling. In addition, various buffer implementation techniques well known in the area of stream processing are integrated into the polyhedral code generation process. Each implementation type represents a trade-off between storage size and computational complexity. The best throughput results in our experimental evaluation are achieved at a cost of 2.5× larger storage size. In some algorithms though, the increase in storage size with particular buffer implementations is prohibitive in itself and could also reduce throughput by preventing efficient cache utilization.

This work includes a preliminary evaluation of the effects of the proposed compilation method on input-output latency. We find that tiled execution and parallelization may add the tile size and degree of parallelism as an additional factor to latency in some cases. Still, our experiments exhibit the best throughput while increasing latency only by a fraction of the problem size. Further work is required to determine the real-time latency effects.

To our knowledge, this work represents the first code generation solution for polyhedral models with unbounded domains. While this is particularly useful in compiling recurrence equations, we believe this work may find wider application in the domain of stream processing. Although polyhedral optimization could already be applied to isolated bounded parts of streaming programs even without the techniques presented here, modeling an entire unbounded program provides more opportunities for optimization, for example, tiling over time, merging stream operators, and so on. A variety of source languages may therefore benefit from this work as long as a translation of whole programs into the unbounded model exists. While the focus of this article is to present an effective method for compilation of recurrence equations, the benefit of the proposed techniques for other source languages deserves a more exhaustive evaluation and comparison with existing techniques.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments, which have improved this article.

REFERENCES

Aravind Acharya and Uday Bondhugula. 2015. PLUTO+: Near-complete modeling of affine transformations for parallelism and locality. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. ACM, New York, NY, 54–64. DOI: https://doi.org/10.1145/2688500.2688512

Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. 2012. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Los Alamitos, CA. DOI: https://doi.org/10.1109/SC.2012.107

Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *Compiler Construction*. Springer, Berlin, 283–303. DOI: https://doi.org/10.1007/978-3-642-11970-5_16

Somashekaracharya G. Bhaskaracharya and Uday Bondhugula. 2013. PolyGLoT: A polyhedral loop transformation framework for a graphical dataflow language. In *Compiler Construction*. Springer, Berlin, 123–143. DOI: https://doi.org/10.1007/978-3-642-37051-9_7

Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. 2016. Automatic storage optimization for arrays. *ACM Trans. Program. Lang. Syst.* 38, 3 (2016), 1–23. DOI: https://doi.org/10.1145/2845078

Stefan Bilbao. 2009. *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*. John Wiley & Sons. DOI: https://doi.org/10.1002/9780470749012

Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008a. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction*. Springer, Berlin, 132–146. DOI: https://doi.org/10.1007/978-3-540-78791-4_9

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008b. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 101–113. DOI: https://doi.org/10.1145/1375581.1375595

Pierre Boulet. 2007. *Array-OL Revisited, Multidimensional Intensive Signal Processing Specification*. Research Report RR-6113. INRIA.

Francois Charot, Madeleine Nyamsi, Patrice Quinton, and Charles Wagner. 2004. Modeling and scheduling parallel data flow systems using structured systems of recurrence equations. In *Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04)*. IEEE Computer Society, Washington, DC, 6–16. DOI: https://doi.org/10.1109/ASAP.2004.1342454

Alain Darte, Rob Schreiber, and Gilles Villard. 2003. Lattice-based memory allocation. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'03)*. ACM, New York, NY, 298–308. DOI: https://doi.org/10.1145/951710.951749

Hritam Dutta, Frank Hannig, and Jurgen Teich. 2006. Hierarchical partitioning for piecewise linear algorithms. In *Proceedings of the International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*. IEEE Computer Society, Washington, DC, 153–160. DOI: https://doi.org/10.1109/PARELEC.2006.43

Johan Eker and Jörn W. Janneck. 2003. *CAL Language Report: Specification of the CAL Actor Language*. Technical Report. University of California at Berkeley.

Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *Int. J. Parallel Program.* 20, 1 (Feb. 1991), 23–53. DOI: https://doi.org/10.1007/BF01407931

Paul Feautrier. 1992b. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *Int. J. Parallel Program.* 21, 5 (Oct. 1992), 313–347. DOI: https://doi.org/10.1007/BF01407835

Paul Feautrier. 1992a. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Program.* 21, 6 (Dec. 1992), 389–420. DOI: https://doi.org/10.1007/BF01379404

Paul Feautrier and Christian Lengauer. 2011. Polyhedron model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, Boston, MA, 1581–1592. DOI: https://doi.org/10.1007/978-0-387-09766-4_502

Martin Griebl. 2001. On tiling space-time mapped loop nests. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*. ACM, New York, NY, 322–323. DOI: https://doi.org/10.1145/378580.378740

Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 4 (2012), 1250010. DOI: https://doi.org/10.1142/S0129626412500107

Tobias Grosser, Sven Verdoolaege, and Albert Cohen. 2015. Polyhedral AST generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.* 37, 4, Article 12 (July 2015), 50 pages. DOI: https://doi.org/10.1145/2743016

Frank Hannig. 2009. *Scheduling Techniques for High-Throughput Loop Accelerators*. Dissertation. University of Erlangen-Nuremberg, Germany.

François Irigoin and Rémi Triolet. 1988. Supernode partitioning. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL'88)*. ACM, New York, NY, 319–329. DOI: https://doi.org/10.1145/73560.73588

Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in dataflow programming languages. *ACM Comput. Surv.* 36, 1 (Mar. 2004), 1–34. DOI: https://doi.org/10.1145/1013208.1013209

Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. 1967. The organization of computations for uniform recurrence equations. *J. ACM* 14, 3 (July 1967), 563–590. DOI: https://doi.org/10.1145/321406.321418

Joachim Keinert and Jürgen Teich. 2011. *Buffer Analysis for Complete Application Graphs*. Springer, New York, NY, 151–208. DOI: https://doi.org/10.1007/978-1-4419-7182-1_7

Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2007. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 235–244. DOI: https://doi.org/10.1145/1250734.1250761

Hervé Le Verge, Christophe Mauras, and Patrice Quinton. 1991. The ALPHA language and its use for the design of systolic arrays. *J. VLSI Signal Process. Syst.* 3, 3 (Sep. 1991), 173–182. DOI: https://doi.org/10.1007/BF00925828

Jakob Leben. 2016. Arrp: A functional language with multi-dimensional signals and recurrence equations. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design (FARM'16)*. ACM, New York, NY, 17–28. DOI: https://doi.org/10.1145/2975980.2975983

Jakob Leben. 2019. Polyhedral Compilation for Multi-dimensional Stream Processing: Experimental Framework and Data (Version 1). Zenodo. DOI: https://doi.org/10.5281/zenodo.2650704

Jakob Leben and George Tzanetakis. 2019. Polyhedral Compilation for Multi-dimensional Stream Processing: Addendum (Version 1). Zenodo. DOI: https://doi.org/10.5281/zenodo.2652490

Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sep. 1987), 1235–1245. DOI: https://doi.org/10.1109/PROC.1987.13876

Vincent Lefebvre and Paul Feautrier. 1998. Automatic storage management for parallel programs. *Parallel Comput.* 24, 3–4 (may 1998), 649–671. DOI: https://doi.org/10.1016/S0167-8191(98)00029-5

Christian Lengauer and Martin Griebl. 1995. On the parallelization of loop nests containing while loops. In *Proceedings of the 1st Aizu International Symposium on Parallel Algorithms/Architecture Synthesis (PAS'95)*. IEEE Computer Society, Washington, DC, 10–18. DOI: https://doi.org/10.1109/AISPAS.1995.401360

Praveen K. Murthy and Edward A. Lee. 2002. Multidimensional synchronous dataflow. *IEEE Trans. Signal Process.* 50, 8 (Aug. 2002), 2064–2079. DOI: https://doi.org/10.1109/TSP.2002.800830

Dmitry Nadezhkin, Hristo Nikolov, and Todor Stefanov. 2013. Automated generation of polyhedral process networks from affine nested-loop programs with dynamic loop bounds. *ACM Trans. Embed. Comput. Syst.* 13, 1s (Dec. 2013). DOI: https://doi.org/10.1145/2536747.2536750

Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. 2000. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Program.* 28, 5 (Oct. 2000), 469–498. DOI: https://doi.org/10.1023/A:1007554627716

Sanjay V. Rajopadhye. 1989. Synthesizing systolic arrays with control signals from recurrence equations. *Distrib. Comput.* 3, 2 (June 1989), 88–105. DOI: https://doi.org/10.1007/BF01558666

Sanjay V. Rajopadhye and Richard M. Fujimoto. 1990. Synthesizing systolic arrays from recurrence equations. *Parallel Comput.* 14, 2 (1990), 163–189. DOI: https://doi.org/10.1016/0167-8191(90)90105-I

Yannick Saouter and Patrice Quinton. 1993. Computability of recurrence equations. *Theoret. Comput. Sci.* 116, 2 (1993), 317–337. DOI: https://doi.org/10.1016/0304-3975(93)90326-O

William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002a. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer, Berlin, 179–196. DOI: https://doi.org/10.1007/3-540-45937-5_14

William Thies, Jasper Lin, and Saman Amarasinghe. 2002b. *Phased Computation Graphs in the Polyhedral Model*. Technical Report. MIT Laboratory for Computer Science.

Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. GRAPHITE two years after: First lessons learned from real-world polyhedral compilation. In *Proceedings of the GCC Research Opportunities Workshop (GROW'10)*. INRIA. Retrieved from https://hal.inria.fr/inria-00551516.

Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. 2004. Translating affine nested-loop programs to process networks. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. ACM, New York, NY, 220–229. DOI: https://doi.org/10.1145/1023833.1023864

Sven Verdoolaege. 2013. Polyhedral process networks. In *Handbook of Signal Processing Systems*, Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala (Eds.). Springer New York, New York, NY, 1335–1375. DOI: https://doi.org/10.1007/978-1-4614-6859-2_41

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013), 54:1–54:23. DOI: https://doi.org/10.1145/2400682.2400713

Jie Zhao, Michael Kruse, and Albert Cohen. 2018. A polyhedral compilation framework for loops with dynamic data-dependent bounds. In *Proceedings of the 27th International Conference on Compiler Construction (CC'18)*. ACM, New York, NY, 14–24. DOI: https://doi.org/10.1145/3178372.3179509

Received July 2018; revised April 2019; accepted May 2019