A Programming Language Based on Recurrence Equations and Polyhedral Compilation for Stream Processing

by

Jakob Leben

Bachelor of Arts, from University of Ljubljana, Slovenia, 2010
Master's Degree, from Institute of Sonology, Royal Conservatoire,
The Hague, The Netherlands, 2012

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

University of Victoria

A Programming Language Based on Recurrence Equations and Polyhedral Compilation for Stream Processing

by

Jakob Leben

Bachelor of Arts, from University of Ljubljana, Slovenia, 2010
Master's Degree, from Institute of Sonology, Royal Conservatoire,
The Hague, The Netherlands, 2012

Supervisory Committee

Dr. George Tzanetakis, Department of Computer Science
Supervisor

Dr. Yvonne Coady, Department of Computer Science
Departmental Member

Dr. Amirali Baniasadi, Department of Electrical and Computer Engineering
Outside Member

Abstract

The work presented in this dissertation contributes to the field of programming language design and implementation for stream processing applications. There is a fast-expanding domain of stream processing applications which demand processing high-volume streams quickly and often in real time. Examples include analysis and synthesis of audio, video and other digital media, sensor array signals, real-time physical simulation etc. High performance is crucial in this domain. When choosing between available programming methods, the programmer often chooses one that maximizes performance while sacrificing ease of programming, code comprehension, maintainability and reusability. This work contributes towards improving the state of the art by jointly maximizing these aspects.

High-volume streams are often most naturally represented as multi-dimensional arrays with one infinite dimension representing time. Algorithms working with such streams are typically defined mathematically using recurrence equations. A programming language is presented in this dissertation which enables an almost literal translation of such mathematical definitions to computer programs. The language also supports powerful facilities for abstraction and code reuse such as polymorphic and higher-order functions. Together, these features enable a more natural expression of algorithms and improve code modularity and reusability.

A major contribution of this dissertation is the compilation of the proposed language in the polyhedral framework, specifically targeting general-purpose multi-core processors. This framework provides powerful means of analysis and transformations of computations on multi-dimensional arrays, which enables data-locality optimizations essential for high performance on general-purpose processors with deep memory hierarchies. The benefit of this framework for computations on finite arrays has been extensively explored. However, this dissertation presents essential extensions that enable the application of state-of-the-art optimizations in this framework on infinite arrays representing streams.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor Dr. George Tzanetakis who has helped me throughout this research endeavour in so many ways. I thank him for seeing in me, through my earlier software projects, a curiosity and capacity for exploring more fundamental questions in computer science. George has created a supportive and encouraging space for scientific inquiry that has enabled me to identify and pursue my research goals. His academic guidance and expertise has helped my realize them.

I thank the members of my supervisory committee for their critical evaluation and constructive feedback, which have helped me to focus my work as well as connect it with a wider context.

I am grateful to all the professors at the University of Victoria who have imparted their knowledge to me in the form of lectures and discussions. Dr. Nigel Horspool's excellent class on compiler construction was a major source of motivation, and was instrumental in defining the direction of my work. I have greatly enjoyed Dr. Yvonne Coady's enthusiastic lectures on the beautiful and intricate problems in the world of concurrency. Her teaching has inspired me to contemplate the deeper questions about the nature of computing. My special gratitude goes to Dr. Daniela Damian - she has gone out of her way in offering support to every graduate student seeking guidance on their research path. On several occasions, she has generously volunteered her time, attentive listening and insightful reflections on the most fundamental scientific questions that have helped me clarify the purpose, scope and goal of my research.

I would also like to thank my parents for all their loving support. They ignited and nurtured my thirst for knowledge and work ethic that have provided crucial motivation and guidance in my academic pursuits.

Finally, I would like to express my immense gratitude to my partner Elizabeth who has witnessed and supported me through the larger part of my Ph.D. study. She has been the best companion I could imagine both in the hardest as well as the happiest parts of this journey.

Chapter 1

Introduction

In this dissertation, I present my contributions to the field of programming language design and implementation in support of stream processing applications. I focus on the fast-expanding domain of applications which demand processing high-volume streams quickly and often in real time; this includes feature extraction from audio, video and other digital signals, real-time physical simulation, etc. In support of this application domain, my work addresses the challenge of increasing programmability of stream processing applications (modularity, reusability, simplicity, correctness of code) without sacrificing performance. The foundation is a new functional language where multi-dimensional streams are defined using recurrence equations in combination with polymorphic and higher-order functions. Novel techniques in the polyhedral model are presented to enable efficient compilation and aggressive optimization of the proposed language specifically for general-purpose multi-core processors.

Stream processing problems often have a structure that supports a good amount of code reuse and is highly amenable to abstraction. For example, such problems can be hierarchically decomposed into streaming subproblems. Similar patterns are found with only slight variations at multiple levels of abstraction and multiple stages of processing. Certain implementation details like buffering of streams between operators are so similar across applications that they can be fully automated. A programming system that exploits these opportunities can greatly simplify program development and maintenance, facilitate code reuse, enable reasoning about all aspects of the program in a common setting, and minimize programmer errors. Such programming systems have a long history and a great variety of them is available today.

However, I believe there is room for improvement in support of today's high-volume stream processing applications. Streams such as multi-channel audio, video,

sensor array data and large vectors of features extracted from these sources are most naturally represented as multi-dimensional arrays with one infinite dimension representing time. Algorithms operating on streams like this are usually communicated in technical publications and textbooks in a mathematical form using recurrence equations. Still, there are few programming languages with a multi-dimensional stream representation, and even fewer support defining such streams entirely using the mathematical notation of recurrence equations. Although multi-dimensional streams can be modeled as "flattened" single-dimensional streams, a price may be paid both in terms of programmability and performance. It forces the programmer to manually implement the conversion to and from the multi-dimensional representation or implement algorithms in an unnatural way, reducing the potential for code reuse. While a multi-dimensional representation poses additional challenges for a compiler, it also offers opportunities for optimization that would otherwise be missed (e.g. multi-dimensional tiling for data locality).

In today's practice, the limitations with respect to programmability and performance of high-volume multi-dimensional stream processing are often circumvented using the coarse-grained stream programming paradigm. This paradigm divides program implementation into two distinct tasks: stream operator implementation and composition. Primitive stream operators are implemented using a flexible, high-performance language such as C++ with OpenMP annotations. Individual operators can contain relatively large portions of computation. The notion of streams only appears at a larger scale where the primitive operators are composed into a stream graph in a domain-specific "coordination language". As a result, programmers require fundamentally different reasoning about the behavior of their application at different levels of abstraction and code can not be reused across levels.

This dissertation presents novel solutions to these problems. As a foundation, I propose a programming language design based on recurrence equations in which streams are represented as multi-dimensional arrays (sequences) with one infinite dimension representing time. Since recurrence equations are commonly used in mathematical definitions of sequences and their transformations, such a language promises a short path from mathematical definitions to executable code. Recurrence equations also naturally accommodate multi-dimensional sequences. In the proposed language, this syntax is combined with higher-order functions, polymorphism and type inference to support a high level of modularity and code reuse.

Despite these sophisticated abstraction features, it is shown in this dissertation

that the proposed language can be completely reduced to a system of affine recurrence equations (SARE). The benefit of this reduction is that a SARE can be further translated into efficient executable code with a statically computed schedule and statically allocated memory. A crucial part of this translation is the polyhedral framework [26]. However, existing polyhedral techniques assume finite arrays. The infinite arrays representing streams in the proposed language pose significant obstacles. The major contributions of this dissertation are novel techniques in the polyhedral framework to handle infinite arrays (recurrence equations with unbounded domains). While the polyhedral framework has previously been used for hardware synthesis from unbounded recurrence equations [72], to the best of my knowledge, this dissertation offers the first complete method for generation of software for general-purpose machines. This includes a polyhedral schedule transformation called *periodic tiling*, integration of existing storage allocation techniques in such a way as to ensure finite storage for infinite arrays, and finally extensions of polyhedral code generation techniques for infinite, periodically tiled schedules.

The polyhedral framework offers more than simply facilitating compilation of recurrence equations. An abundance of research has shown its benefits for data locality optimization and automatic parallelization, especially for multi-dimensional array computations [13, 32, 64]. In this dissertation, I demonstrate how such optimizations are accommodated in the proposed compilation method for stream processing. Empirical evaluation shows that they can have a bigger effect when applied to a polyhedral model that captures entire infinite streams, compared to only finite parts of a streaming program as has been previously done. This implies a potential impact of the compilation techniques introduced in this dissertation beyond the particular language proposed here. The compilation techniques are defined in a general way, accepting as input an abstract polyhedral model which could be derived from a variety of other languages, thus extending the reach of polyhedral optimizations to stream processing in general.

1.1 Contributions

The contributions presented in this dissertation are summarized as follows:

- A functional programming language for stream processing named Arrp, based on recurrence equations and featuring higher-order and polymorphic functions

and type inference.

- A method for reduction of Arrp programs to a system of affine recurrence equations and derivation of a polyhedral model.
- A method for translation of polyhedral models of stream processing programs to imperative code with statically allocated memory, including the following:
  - A polyhedral schedule transformation called *periodic tiling* which exposes periodicity in a program with infinite arrays while accommodating state-of-the-art schedule optimizations.
  - A proof that a well-known polyhedral storage optimization called *modular mapping* yields bounded storage for a program with infinite arrays and a periodically tiled schedule.
  - An extension of polyhedral code generation methods to generate imperative code for a non-terminating stream processing program using a periodically tiled schedule.
- A case study comparing Arrp with two other stream processing languages using 4 representative stream processing programs.
- A preliminary experimental evaluation of Arrp on 3 signal processing applications, demonstrating its usability.
- An experimental evaluation of the compilation method on 5 stream processing kernels with large problem sizes, indicating performance benefits in comparison to hand-written C++.

1.2 Organization of Dissertation

The rest of this dissertation is organized as follows:

- Chapter 2 presents the problems addressed in this dissertation in more detail and discusses related work.
- Chapter 3 contains a description of the syntax and semantics of the proposed language for stream processing, and describes a method for its reduction to affine recurrence equations. This includes my previously published work [53].

- Chapter 4 formally defines the problem of code generation from unbounded affine recurrence equations in the polyhedral framework and presents my solution. This includes my previously published work [56].
- Chapter 5 compares Arrp with two other languages for stream processing by studying possible implementations of a number of algorithms in these languages.
- Chapter 6 presents two sets of experiments for empirical evaluation of the research described in the previous chapters. These experiments were part of my earlier publications [53, 56].
- Chapter 7 summarizes the contributions of the dissertation, discusses their significance and relates them to possible future work.
- Appendix A lists all my publications to date.
- Appendix B contains Arrp code examples, including those used in the experiments presented in Chapter 6.

Chapter 2

The Problem and Related Work

2.1 The Problem in More Detail

2.1.1 Desired Language Features

One goal of this dissertation is the design of a programming language where stream processing programs can be expressed in a form as close as possible to the usual notation in mathematical definitions. As an example, consider the following definition of the finite-difference time-domain (FDTD) method to compute the 2D wave equation [10]:

$$\begin{aligned} u[n,i,j] = {} & b_0 u[n-2,i,j] + b_1 u[n-1,i,j] \\ & + b_2\big(u[n-1,i-1,j] + u[n-1,i+1,j] \\ & + u[n-1,i,j-1] + u[n-1,i,j+1]\big), \\ & 0 < i < M-1, \quad 0 < j < N-1 \end{aligned} \tag{2.1}$$

Algorithms like this (more generally called stencil computations) are used for example in a variety of physical simulations. This equation describes the evolution of a 2-dimensional grid representing discrete points in space (indexed by i and j) over discrete points in time (indexed by n). In scientific applications, it is common to limit the simulation duration, e.g. $0 \leq n < T$. However, there is an increasing field of real-time applications for such algorithms; for example, in digital musical instruments used in real-time musical performances. In such applications, the equation above defines the behavior of a non-terminating program, and so n has no upper bound. This makes u a 3-dimensional stream, i.e. a 3-dimensional array with one infinite

dimension. Unfortunately, there are few programming languages with support for defining multi-dimensional infinite arrays with a syntax close to 2.1.

Another goal of this dissertation is to support code reuse when working with streams using the syntax of recurrence equations. To demonstrate this, we will use another stream processing example called *max filter* where each output stream element is the maximum of a finite group of input stream elements:

$$y[n] = \max_{i=0}^{N-1} x[n+i] \tag{2.2}$$

There are multiple opportunities for code reuse in an implementation of this algorithm. These opportunities can be exploited with well-known methods of abstraction like higher-order polymorphic functions, type system features like dependent types, and syntax features like array comprehensions. In the following, we explore their role specifically in the design of a language for stream programming using a syntax close to the above equations, with the max filter as an example.

For code reuse, the language should obviously support functions on streams represented as infinite arrays, so that Eq. 2.2 can be wrapped into a function f such that $y = f(x, N)$.

The operator max in Eq. 2.2 is essentially a function of a finite array - applied to a finite portion of the infinite array x. The language should support the implementation and application of such functions in general, so that $y[n] = \max(w(x, n, N))$, where $w(x, n, N)$ represents some generic and convenient expression for selecting a finite portion of a stream x. To increase reuse, functions on finite arrays should also be polymorphic in array size.

Moreover, the operator max is just one example of the common pattern of *reductions*. A higher-order function $R(f, x)$ which computes a reduction of a finite array x using a binary function f, can be reused to quickly define all kinds of reductions, for example: $\Sigma(x) = R(+, x)$, and $\max(x) = R(\max'', x)$ (where $\max''$ is a pre-defined binary function).

This algorithm is also an instance of the general class of *windowed* algorithms, where each output element depends on a finite portion of an input stream - a *window*. The windows are sometimes overlapping, but sometimes there are elements between windows which are ignored. The spacing between windows is called a *hop*. The

following variant of the max filter uses a parameter H for the hop size:

$$y_h[n] = \max_{i=0}^{N-1} x[Hn + i] \tag{2.3}$$

However, a hop size other than 1 makes this a *multi-rate* algorithm - computing one more element of y requires H more elements of x. Windowed algorithms with a hop size larger than 1 are very common in feature extraction from audio signals, for example. Hence, the desired language should support multi-rate algorithms, and ideally an implementation of such algorithms should support a variable hop size.

Now, assume that we have a stream of M-tuples - represented as a 2-dimensional stream $x'[n, j]$ where n indexes tuples and j tuple elements. Alternatively, this can be seen as a bundle of M one-dimensional streams, one for each j. Suppose that we wish to implement a program that computes Eq. 2.2 for each constituent stream. More precisely, we want to compute the two-dimensional array:

$$y'[n, j] = \max_{i=0}^{N-1} x'[n + i, j], \qquad 0 \leq j < M \tag{2.4}$$

Code can be reused if we can implement a polymorphic function f satisfying both $y = f(x)$ and $y' = f(x')$. This means that the function f should be polymorphic in array shape: accepting both one and two-dimensional arrays. The definition of such a function can be supported by overloading several kinds of expressions. For example, if x is a 2D array, then $x[n]$ means a 1D array $w[j] = x[n, j]$. Also, if w is a 1D array, then $y[n] = w$ means a 2D array $y[n, j] = w[j]$. Finally, built-in operators like max and + can be overloaded to operate on arrays in a pointwise manner.

We can also benefit from polymorphic functions accepting both finite and infinite arrays. For example, one may reasonably expect a function that independently maps each element of an input array to an element of an output array to apply to both kinds of arrays. This may sound obvious in an abstract mathematical context. In practice, stream processing often involves different programming paradigms, sometimes even different languages, at the level of stream operator implementation and composition - code involving streams can not be used on finite sequences and vice versa.

2.1.2 Compilation and Performance Challenges

The goal of the compilation of the desired language is to generate machine code that iteratively computes the values of a chosen stream from the source program. For

example, to compute the 2-dimensional max filter defined in Eq. 2.4 we would like to generate code similar to the C code in Figure 2.1.

```
float x[N][M]; // ... initialize x...
int n = 0;
while(work)
{
    for (int j = 0; j < M; ++j)
    {
        x[n][j] = input();
        float y = x[n][j];
        for (int i = 1; i < N; ++i)
            y = max(y, x[(n+i)%N][j]);
        output(y);
    }
    n = (n + 1) % N;
}
```

Figure 2.1: A C implementation of Eq. 2.4.

Obviously, the language requires a lazy (non-strict) semantics for stream references. If the output stream is defined by reference to other streams, we should not attempt to compute those streams entirely before computing the output stream. One challenge therefore is to interleave the computation of streams.

For performance reasons, we should also take care to not generate unnecessary intermediate arrays. For example, if the term max in Eq. 2.4 is represented as a function on a finite array extracted from the stream x, creating this array in the machine code would result in a large number of unnecessary copies of elements from x.

There is another significant performance concern. On modern general-purpose multi-core processors with deep memory hierarchies, it is crucial to successfully utilize the memory caches. This translates to optimizing *data locality*: instructions that use data close in memory should be executed close in time. The effect on performance can be dramatic. In addition to speeding up execution on each individual processor core, cache optimizations also enable more parallelism. This is so because multiple cores must wait for each other when accessing shared memory due to a miss in their private cache.

It turns out that the code in Figure 2.1 is particularly bad in this regard: each iteration of the innermost loop skips an entire row of the array `x` (assuming row-major order). Interchanging the two `for` loops as demonstrated in Figure 2.2 can

```
float x[N][M]; // ... initialize x...
float y[M];
int n = 0;
while(work)
{
    for (int j = 0; j < M; ++j)
    {
        x[n][j] = input();
        y[j] = x[n][j];
    }

    for (int i = 1; i < N; ++i)
        for (int j = 0; j < M; ++j)
            y[j] = max(y[j], x[(n+i)%N][j]);

    for (int j = 0; j < M; ++j)
        output(y[j]);

    n = (n + 1) % N;
}
```

Figure 2.2: A better C implementation of Eq. 2.4.

dramatically improve performance when `N` and `M` are large.

Figure 2.2 is still not the best we can do. The best data locality is achieved with an optimization called *tiling*. Rather than scanning multi-dimensional arrays in a lexicographical order, the idea is to partition arrays into relatively small multi-dimensional tiles (rectangles, cubes, other shapes...) and computing tile after tile. Data in a cache can thus be reused in multiple directions within a tile before being evicted by accessing more remote data in other tiles. However, writing tiled code by hand is extremely laborious and error-prone: efficient tiling is often achieved by tiles with complex shapes and it involves writing a large number of deeply nested loops with complicated expressions for bounds.

Therefore, automated methods for data locality optimizations have been developed. Arguably, the most successful is the polyhedral framework [26] which is gradually being adopted in production compilers like the GNU C compiler [79] and the LLVM framework [31]. A typical polyhedral optimization process starts by deriving a polyhedral model from static affine nested loop sections of a program [23, 81]. Then, the schedule of the program is transformed - for example using a popular scheduling algorithm introduced in the Pluto optimizer [13] which enables good tiling for data locality while exposing parallelism. Finally, the polyhedral model with a trans-

formed schedule is converted back to imperative code [70, 5, 33]. The output code is parallelized using OpenMP [13] or by generating kernels for GPUs [83].

There are obstacles though when applying the state-of-the-art polyhedral techniques on potentially infinite programs. For example, they can generate loop nests with inner unbounded loops. This means that such techniques could be applied to the finite body of the `while` loop in Fig. 2.1, but not the entire program. However, experimental measurements reveal that this misses a significant optimization opportunity. The best performance is achieved only when Eq. 2.4 is tiled over time, which means considering multiple iterations of the `while` loop. The measurements supporting this claim are presented in Chapter 6, specifically in Figure 6.1. In this dissertation, I address the obstacles towards applying polyhedral optimizations on (potentially infinite) stream processing programs.

The power of the polyhedral model however stems from certain limitations it imposes on programs: for example, array index expressions must be affine expressions. Fortunately, many stream processing problems like those presented above satisfy these constraints. Such constraints are therefore adopted by the language and compiler techniques presented in this dissertation.

2.2 Related Languages

A language similar to Arrp has been recently proposed [75] (published after the conception of Arrp in 2016). This language does not focus specifically on stream processing and has fewer limitations. For example, contrary to Arrp, it supports arrays with multiple infinite dimensions and it imposes no restrictions on array index expressions. The meaning and utility of multiple infinite dimensions however is not obvious; one infinite dimension to represent time is usually enough. This and other features also make the language less amenable to optimization, as discussed in more detail in Section 2.3. Another language supporting unbounded recurrence equations is ALPHA [52, 84, 19, 17]. It was designed specifically for systolic array synthesis using the polyhedral model - rather than software generation like Arrp. PAULA [35] is another language with recurrence equations designed for the same purpose as ALPHA, although there is no report of its application using unbounded recurrence equations. Both ALPHA and PAULA lack higher-order and polymorphic functions. Many other languages with finite arrays support a syntax and semantics close to recurrence equations. This includes C, Haskell, and Python, to name just a few.

Recurrence equations are well complemented by pointwise array operations: the latter can simplify some stream definitions, as well as make stream functions polymorphic in stream shape, as described in section 2.1.1. The style of array programming using pointwise operators, without referring to individual array elements, is called point-free style. This style is most frequently used in scientific computing for operations on finite arrays representing vectors and matrices. For example, the + operator is applied to two vectors directly, implying the pointwise sum of their elements. The language APL [42] is known as having promoted and popularized this style. Modern examples of scientific languages with this style include MATLAB [1], Octave [2], R [3] and Julia [4].

There is a large group of stream programming languages which exclusively support the point-free style: the programmer defines complex streams using a few primitive stream constructors and stream operators. Examples include the early textual dataflow languages VAL and Lucid (see [43] for an overview), the language ALPHA for the design of systolic arrays [52, 17], the synchronous reactive languages Lustre [16] and Signal [34], the signal processing languages Faust [67], Kronos [66] and Sig [78], a signal processing language embedded in Haskell [3] and many more. Besides pointwise arithmetic operators, stream processing in this style usually involves a few unique operators. Most notable is the operator which prefixes a stream with a finite number of elements (sometimes called *delay*). This allows recursive stream definitions, e.g. $x = \delta(0, x + 2)$, where x denotes a stream, $\delta(0, s)$ prefixes a stream s with a single zero, and $x + 2$ adds 2 to all elements of x; this equation is satisfied by the stream $(0, 2, 4, 6, ...)$. This programming style is most suitable for single-dimensional streams; implementation of multi-dimensional algorithms like Eq. 2.1 in this style can be rather inconvenient and is often not supported.

An extreme form of the point-free style is one where even streams are not explicitly represented - only stream operators are. If we consider stream operators as functions mapping streams to streams, this corresponds to the general functional programming style where functions are combined without directly mentioning their arguments - for example $f \circ g$ means a function $x \mapsto f(g(x))$. Consequently, this style is naturally available with stream processing libraries for general purpose functional languages like Haskell [3]. This style is also the basis for arrowized functional reactive programming

[1] `https://www.mathworks.com/products/matlab.html`
[2] `https://www.gnu.org/software/octave/`
[3] `https://www.r-project.org/`
[4] `https://julialang.org`

[40]. Another example is process composition in the UNIX shell, where the expression `a | b` directs the output of process a to the input of process b. The previously mentioned language Faust for signal processing is another example where this style is used extensively. For example, the Faust expression `_ <: (_, _') : -` means a function $x \mapsto y$ where $y[i] = x[i] - x[i-1]$. While this style supports extremely concise programs, programming complex programs solely in this style can be impractical and make programs incomprehensible.

Visual dataflow languages represent another very popular paradigm. Simulink [5], LabVIEW [6], Ptolemy [68], and similar graphical environments are often used in the design of signal processing systems. While visual programming offers an extremely intuitive expression of simple relations between stream operators, it may be less practical for expressing complex behaviors using a large number of operators. Similarly to the point-free textual style described above, the visual style is especially inconvenient for complex multi-dimensional algorithms. For example, Array-OL [27] is a visual language with multi-dimensional streams. Besides the visual composition of nodes into a graph, it requires a large number of textual annotations, which makes it rather hard to work with and comprehend complex programs.

To various degrees, the aforementioned stream programming styles can be complemented with coding styles and languages that are not specific to stream processing. In a common paradigm, the overall stream graph - the set of operators and their communication patterns - is defined in a high-level language, called a *coordination language*, while the details of each operator's behavior are filled in using a different programming style. For example, a built-in operator in a coordination language may implement the general pattern of applying the same function repeatedly on consecutive finite chunks (windows) of an input stream to compute consecutive chunks of an output stream; the applied function however can be implemented in a general-purpose language without the notion of streams. This corresponds for example to the Expression operator in the visual environment Ptolemy, the Expression Language used to customize the behavior of many built-in operators in the IBM Streams Processing Language (SPL) [38], stream operators *map* and *reduce* in Apache Flink [7] which accept functions as parameters, etc. StreamIt [76] is an imperative textual language with a distinct programming style for stream operator implementation and

[5] `https://www.mathworks.com/products/simulink.html`
[6] `https://www.ni.com/en-ca/shop/labview.html`
[7] `https://flink.apache.org/`

composition. This distinction is also found in many stream processing libraries for general-purpose languages. Operator implementation is sometimes quite disjoint from their composition. The CAL Actor Language [22] has the specific purpose of implementation of stream operators, to be composed in a different language. Ptolemy and Apache Flink support actors implemented as separate Java classes. Simulink, IBM SPL and LabVIEW support actors implemented in C++.

An extreme case of stream programming with a minimal notion of streams is the implementation of stream operators as independent processes (threads), communicating using streams of messages. This includes for example simply relying on the UNIX operating system facilities for multi-threading and communication using sockets and pipes. A little more structured solutions include Kahn's Process Networks [45], Hoare's Communicating Sequential Processes [39], the Message Passing Interface (MPI) specification [8], 'goroutines' and channels in the language Go [9], and numerous other messaging libraries for general-purpose languages.

Aside from the variety of approaches to programming stream processing systems presented above, streaming programs can be classified as fine-grained or coarse-grained - depending on how prominent the notion of the stream graph is. In fine-grained programs, each stream operator performs a small task and most of the program behavior is described by the stream graph. In coarse-grained programs, each stream operator performs a complex task and more of the program behavior is described in the implementation of operators. This dissertation does not particularly promote fine-grained or coarse-grained approaches. Rather, it is motivated by the problems stemming from a strong boundary between stream operator implementation and composition which makes the distinction between fine-grained and coarse-grained approaches more acute. The programmer carries the burden of deciding what aspects of program behavior to place on each side of the boundary. This decision is often dictated by what kind of behaviors can be expressed on each side of the boundary as well as by performance implications of the alternatives. This can be an obstacle to natural program modularization, which is particularly problematic because code can not be reused across the boundary.

Based on these observations, this dissertation proposes a programming language named Arrp and a compilation method which support the implementation of streaming programs in a more natural and unified manner across different levels of abstrac-

[8] `https://www.mpi-forum.org/`
[9] `https://golang.org/`

tion. In particular, the ability to define multi-dimensional streams using recurrence equations in Arrp is especially beneficial for high-volume stream processing applications at the focus of this work. Today, such applications are often implemented in a coarse-grained manner with complex primitive operators implemented in C++, for performance reasons and due to a lack of expressivity of stream programming languages.

2.3 Related Compilation Techniques

A large number of programming languages exists, but large groups of them have a lot in common. Underlying the concrete syntax of a language are usually concepts and patterns shared with many others. Together, these concepts form an abstract model of a program also called a *model of computation* (MoC). The fact that a few models of computation are manifested in many different languages makes the development of compilation and optimization techniques more economical - multiple languages can benefit from a technique developed within an abstract model. A standardized representation of a model of computation is often called an *intermediate representation* (IR) because it serves as an intermediate form for a program in the process of compilation - between the source form manipulated by humans and the target form manipulated by or embodied in hardware. One very successful and general intermediate representation is for example the LLVM IR [51] which is at the center of the LLVM compilation framework serving a large number of source languages and targets.

Compilation (translation from a source to a target form) and optimization (transformation from one form to a better form with equivalent meaning) obviously depend on the knowledge about the program and its behavior, since they must preserve the intended behavior. A model of computation is crucial in defining the boundaries of this knowledge. The usual tendency is that a more general model which supports a wider variety of behaviors provides less certainty about the future behavior of a program, while a more restricted model provides more such guarantees. Therefore, certain specialized domains like stream processing that can tolerate more restricted models can also benefit from them. The benefit manifests both in a higher productivity of programmers and a better performance of programs. The programmer benefits because a more restricted model may include assumptions about commonly intended behaviors of programs and so the programmer does need to specify these behaviors explicitly. One example is the buffering of streams communicated between stream op-

erators. The program performance can be improved because a more restricted model limits the possible behaviors of a program and therefore allows more aggressive optimizations with certainty that the behavior will be preserved. For example, if it can be proven that two stream operators produce equal streams, they can be replaced with a single one.

In this section, we look at various models of computation involved in stream programming and associated compilation and optimization techniques. Throughout this overview, focus is placed on models and techniques most relevant to the problems addressed in this dissertation. In particular, we focus on the compilation of functional languages featuring infinite multi-dimensional arrays defined using recurrence equations. The approach to compilation proposed in this dissertation is to completely reduce a program to a set of recurrence equations (via reduction of function applications). Recurrence equations can then be efficiently manipulated in the polyhedral model to generate imperative code. At the same time, the polyhedral model enables powerful data-locality optimizations and parallelization for efficient execution on general-purpose multi-core processors.

2.3.1 Recurrence Equations and the Polyhedral Model

We turn our attention first to the models of recurrence equations. Equations like those used in section 2.1.1 are often found in scientific publications and textbooks. Naturally, the question arises: can we translate such equations directly to executable code? In order to approach this problem, a rigorous model of such equations has been defined - called simply a *system of recurrence equations* (SRE). The model consists of a set of equations in the following general form:

$$v_0[\vec{i}] = e(v_1[m_1(\vec{i})], v_2[m_2(\vec{i})], ...v_n[m_n(\vec{i})]) \tag{2.5}$$

Here, v_j are variables denoting multi-dimensional sequences of atomic values, indexed by tuples of integers. An equation like this defines the value of v_0 at index $\vec{i}$. An equation is associated with a domain - a set of indices $\vec{i}$ to which it applies. The right-hand side of the equation is an expression e which strictly depends on the arguments shown above (values of other sequences or v_0 itself) and has a constant computational time. The sequence on the right hand side of the expression are indexed using mappings m_n of the index $\vec{i}$ of the value being defined. The reader may find that the model does not include common expressions for summation $\sum_{i=a}^{b}$ and similar. Nevertheless, they

can be modeled by adding another dimension to a sequence domain, representing the variable i.

Significant analytical power for SREs is gained from restricting the shape of index domains and index mappings m_n. For example, the SRE model was first proposed by Karp, Miller and Winograd [48], but restricted to Systems of Uniform Recurrence Equations (SURE) where the index mappings m_n are constant translations. Later, it was extended to Systems of Affine Recurrence Equations (SARE) [71, 72]. By restricting the index domains to convex polyhedra and index mappings to affine functions, we can apply affine transformations and linear optimization techniques to construct a schedule for computations of individual sequence values and distribute the computations across parallel processing units. The goal of the earliest affine scheduling techniques was synthesis of systolic array hardware from recurrence equations. This work has been used for example in hardware synthesis from the language ALPHA [52, 17]. These techniques support equations with unbounded domains (infinite sequences, streams), although they have not been used for software generation for general-purpose hardware.

The polyhedral model [26] is a generalization of SARE. This model consists of a set of multi-dimensional arrays (corresponding to sequences in SARE), and a set of statements reading and writing array values (corresponding to equations in SARE). However, multiple statements can write into the same array location. This supports the modeling of imperative languages with multiple assignments into the same memory location. As a consequence the meaning of a model is only fully defined with the addition of a schedule which assigns an order to array accesses. The polyhedral model can describe static affine nested loop programs (SANLP) [23] - for example a set of nested loops in the C language, enclosing assignment statements which write and read array values. Similarly to equations in SARE, a statement has an index domain, also called an *iteration domain*, describing for example the set of loop indices for which the statement executes. Just like in SARE, iteration domains must be describable as polyhedra - for example loop bounds must be affine functions of constant program parameters and enclosing loop indices. Similarly, array indices used in statements must be affine functions of program parameters and loop indices.

Since the motivation for the polyhedral model was optimization and parallelization of SANLP, the techniques based on this model often assume terminating programs (finite statement domains), and are not directly applicable to streaming programs. For example, the goal of the scheduling techniques due to Feautrier [25, 24] is to minimize

the total duration of a program, which obviously does not apply to infinite programs. The same objective is used in hardware synthesis from the language PAULA [35].

Nevertheless, the large amount of research on optimizations of SANLP in the polyhedral framework has produced powerful techniques which are already successfully used on finite parts of streaming programs. Most notably, a popular algorithm used in the Pluto optimizer for C and C++ [12, 13] provides essential optimizations for software execution on general-purpose multi-core machines. Similar algorithms are in use today in several compilers: GRAPHITE in the GCC compiler [79], Polly in the LLVM framework [31], and the R-Stream compiler [63].

In summary, the existing techniques in the polyhedral framework support hardware synthesis from streaming programs on one hand, and on the other hand optimization of terminating programs for general-purpose hardware. However, the work presented in this dissertation leverages the polyhedral model to generate executable code from streaming programs. For this purpose, novel polyhedral techniques are developed which address the limitations of the existing techniques to generate code from unbounded polyhedral models. Moreover, a premise of this dissertation is that polyhedral optimizations of streaming programs can be more successful when performed on a complete unbounded model of a streaming program, rather than its finite parts. The reason is that the latter reduces the available information that can be useful in optimization.

The work presented in this dissertation can benefit a variety of languages, not just the language Arrp presented here. For example, the language $\lambda_{\alpha}^{\infty}$ [75] is very similar to Arrp and supports infinite arrays. However, it currently only has an interpreter and a method of translation to Single-assignment C (SaC) [74] which is limited to finite arrays. It has been suggested [75] that a restriction of that language to finite arrays could be compiled to efficient code using the polyhedral framework, although it is an open question whether infinite arrays can be treated with the same method. This dissertation provides an affirmative answer precisely to this question.

2.3.2 Dataflow Models

A very different group of models of computation, albeit more prevalent in the domain of stream processing, are the so-called dataflow models of computation. Generally, these models directly represent the graph of stream operators, in this context also called a *dataflow graph*. The term *dataflow* is actually borrowed from a general

compiler technique called *data flow analysis*. In particular, one kind of data flow analysis traces definitions and uses of variables to construct a *data dependency graph* (also called dataflow graph) where nodes represent primitive operations and edges their data dependencies; one can say that data "flows" between operations across edges. This analysis is ubiquitous in compilers for all kinds of languages and is not necessarily related to stream processing. In the analysis of a terminating imperative program for example, a node may represent a computation executing a single time, and an edge may transmit a single value during the entire execution of the program. However, nodes representing computations within a loop of an imperative program for example execute many times, each time transmitting values over their incident edges. It is easy to extrapolate from there to a non-terminating program where the data flowing across edges form infinite streams.

A dataflow (data dependency) graph is useful in parallelizing a program, because it clearly indicates independent sets of computations which can execute in parallel. This is the reason why parallelization has been an important motivator for research related to dataflow models. For example, the development of a whole group of so-called *dataflow* languages, including VAL and Lucid mentioned previously, was motivated by the desire to exploit the massive, fine-grained parallelism promised by a new kind of computer architecture developed at the same time - the *dataflow architecture* [43]. The goal of these languages was precisely to support detailed data dependency analysis resulting in a fine-grained dataflow graph - a form directly executed by the dataflow architecture. Since the goal was parallelization of programs in general, some of these languages do not have a concept of streams.

However, another driving force for the development and wide adoption of dataflow models is the fact that these models closely correspond to the intuition about the behavior and structure of streaming programs, and they can be derived in a straightforward way from visual programming languages mentioned in Section 2.2 [58]. Hence, a group of dataflow models has been developed which is particularly suited to streaming programs. An excellent overview of these models and related compilation techniques is provided in the Handbook of Signal Processing Systems [9]. In the rest of this section, we relate them to the work presented in this dissertation. While the previous section focused on languages as manifestations of these models, this section focuses on the benefit of these models for program analysis and optimization.

A more abstract group of dataflow models are the process network models. They include Kahn's Process Networks (KPN) and Hoare's Communicating Sequential Pro-

cesses - underlying the concrete languages proposed by Kahn [45] and Hoare [39]. In these models, stream operators are represented as non-terminating sequential programs communicating over FIFO queues. A number of useful properties have been proven about these models, for example that the KPN model is deterministic (independent of the timing of individual operators), that certain transformations of KPN preserve meaning, etc. However, process networks provide little insight into the internal behavior of stream operators. This precludes more aggressive static program transformations involving merging, interleaving and reordering the computation of different operators, similar to what is enabled by the polyhedral model.

More details about the behavior of a program are captured in actor models - named after the work by Hewitt [37] and Agha [2]. The semantics of stream operators in many programming systems mentioned in Section 2.2 can be described in an actor model, for example Lucid, Faust, StreamIt, IBM SPL, Apache Flink, CAL and a subset of Ptolemy. In contrast with process networks, the behavior of an actor is modeled as a sequence of finite actions in response to incoming stream elements, and hence an operator is also named an *actor*. The execution of an action is called a *firing*, and each action consumes and produces a pre-determined amount of tokens (elements) in input and output channels (queues), and in some models also updates the actor state. Different models impose different restrictions on how many distinct actions an actor can perform and when it can fire. The least restricted actor models are called *dynamic dataflow models*, because they allow firing conditions which involve dynamic program aspects like the value of input tokens and the actor's state. Some models even support non-deterministic execution: for example an actor with two inputs which fires as soon as a token is available on one or the other input obviously depends on the timing of other processes which produce its input. However, models with more restrictions support more static program analysis an optimization.

Since the focus of this dissertation is high-performance stream processing, our attention is directed to the more restricted actor models. The model that has received the most attention is the Synchronous Dataflow (SDF) model [60]. It has been extensively used in digital signal processing applications. In this model, each actor always executes the same action, consuming and producing the same amount of tokens independently of input token values and the actor state. The amount of tokens consumed and produced on each channel is called the *pop rate* and *push rate*, respectively. While this model is still expressive enough for many stream processing programs, it gives the compiler a great power of static analysis and optimization

[59]. For example, a complete schedule of actor firings can be statically determined and bounded-size buffers for actor communication can be statically allocated. The compiler can also distribute computation across parallel processors with more knowledge about the resulting amount of communication and synchronization between the processors. Variations of this model which still do not involve any dynamic information include the more restricted Homogeneous Synchronous Dataflow (HSDF) model [60] where all pop and push rates are 1, the Computation Graphs model [47] with an additional *peek rate* stating the amount of input tokens used in an action but not necessarily consumed, the Cyclo-static Dataflow model [11] where each actor executes a predefined cyclical sequence of actions, the Multi-dimensional Synchronous Dataflow (MDSDF) model [65] supporting multi-dimensional streams, and others.

A lot of research has been done in exploiting the parallelism exposed by dataflow models and their amenability to static analysis, for a variety of target architectures. Of particular interest in this dissertation is code generation for general-purpose multicore processors. However, research suggests that too fine-grained dataflow models can harm performance on such targets [14, 36]. One solution is to merge (fuse) actors into larger actors (superactors) [14, 28]. Careful fusion which minimizes communication between superactors will benefit their parallel execution. Careful fusion may also enable more aggressive optimization of an individual superactor's firing using a general-purpose compiler [14, 28, 8]. Another solution is to rely on coarse-grained actors defined by the programmer. For example, a dataflow scheduling method has been proposed [85] which takes into account internal parallelism of coarse-grained actors (e.g. implemented in C and parallelized using OpenMP). However, as previously mentioned, a coarse-grained dataflow specification with distinct programming paradigms on the level of actor implementation and composition harms modularity and code reuse and limits the benefits of a domain-specific language for programmer productivity. The polyhedral compilation method for stream processing presented in this dissertation addresses these concerns: it supports fine-grained stream programming while relying on the power of the polyhedral framework for efficient grouping and interleaving of stream computations based on their data dependencies. While most of the research on dataflow optimization is limited to single-dimensional stream models, the polyhedral model is particularly suitable for multi-dimensional streams.

There have also been efforts to apply polyhedral techniques directly on dataflow models. An approach in the StreamIt model [20] considers a two-dimensional space where one dimension represents different actors and the other each actor's sequence

of firings. It then searches for good affine transformations of this space followed by tiling to minimize cache misses involved in communication between actors and across actor firings (in the form of carried actor state). The polyhedral model has also been used in the MDSDF model [49] to simplify determination of buffer sizes while using a multi-dimensional polyhedral schedule. However, only scheduling functions involving scaling and shifting are considered in that work, instead of the full range of affine functions supported in state-of-the-art polyhedral scheduling. The polyhedral model has been used to optimize a subset of the LabVIEW dataflow language [6], although the subset only includes finite arrays and streams are not considered. In contrast with the above approaches, this dissertation proposes a generic representation and optimization of complete stream processing programs directly in the polyhedral model. A solution is designed with few assumptions about the origin of this representation. It is shown how it can be derived from the language Arrp in particular (and similar languages based on recurrence equations in general), but it could also be derived from a dataflow model.

The discussion above suggests that the polyhedral compilation method presented in this dissertation offers new optimization opportunities for stream processing in general, not just for a language like Arrp. However, it also has a significant limitation compared to known optimizations in dataflow models. Namely, the polyhedral model has a rather limited support for dynamic behaviors - in this regard, it is equivalent to the SDF model. For this reason, it seems that a combination of dataflow and polyhedral compilation techniques could be particularly symbiotic. For example, the polyhedral method could serve to optimize finer details while the coarser level could be handled in a dynamic dataflow model supporting dynamic reconfiguration. This could be facilitated for a language like Arrp by deriving dataflow actors from recurrence equations. A similar task is accomplished in the derivation of the Polyhedral Process Network model (PPN) from SANLP [82].

2.4 Conclusions

Some stream processing algorithms are most naturally expressed by representing streams as multi-dimensional sequences and defining them using recurrence equations. This chapter has made a case that a programming language with a syntax as close as possible to that form is desirable. Other known stream programming styles may support the implementation of such algorithms, although with more difficulties

for the programmer. Such algorithms also offer a lot of opportunities for code reuse, which can be best exploited with a multi-dimensional stream representation in combination with polymorphic functions and pointwise operators. In Chapter 3 we present a new language named Arrp with a design based on these observations.

High-volume, multi-dimensional streaming applications also pose significant challenges in achieving optimal performance. In order to maximize performance, programmers often implement large amounts of program behavior in faster general-purpose languages, rather than more convenient domain-specific languages. Hence, efficient execution is crucial in order for a new language like Arrp to be adopted. We are particularly concerned with the execution on general-purpose multi-core processors with deep memory hierarchies. On such hardware, data locality optimizations are essential in order to utilize the available computing power.

A lot of research into optimization has been done within the dataflow models of computation, although mostly models with a single-dimensional notion of streams. Moreover, maximizing performance on our hardware of interest often involves a coarse-grained dataflow model with a hard boundary between actor implementation and composition. The polyhedral model seems most suitable for the algorithms and hardware of interest. It supports multi-dimensional arrays and it offers detailed static analysis and aggressive transformations. Together these features promise efficient execution of streaming programs with a homogeneous implementation from the fine to the coarse level in a language like Arrp. However, there are previously unsolved challenges in the application of state-of-the-art polyhedral optimizations on languages like Arrp, where streams are represented using recurrence equations with unbounded domains. These challenges are addressed in Chapter 4.

This dissertation focuses on streaming programs without much dynamically changing behavior which can be modeled in the polyhedral framework. However, many real-world applications require dynamic behaviors, and so an interesting direction for future work is the combination of the techniques proposed here with dynamic dataflow models.

Chapter 3

The Arrp Language

3.1 Introduction

This chapter presents a new language for stream processing named Arrp. Its goal is to support a natural expression of streaming algorithms which are usually represented mathematically as multi-dimensional sequences and defined using recurrence equations. Moreover, the design of Arrp is guided by the desire to improve modularity and code reuse in stream programming, which is supported with functional features like higher-order polymorphic functions and pointwise semantics of primitive operators.

This chapter is structured as follows:

- In section 3.2, we present the syntax and semantics of Arrp. We demonstrate its use with a variety of examples, including multi-dimensional and multi-rate signal processing algorithms, and show how it supports abstraction and code reuse in these domains.
- In section 3.3, we present the reduction of Arrp to a system of affine recurrence equations. This enables further compilation and optimization in the polyhedral model, which is described in the following chapter.

3.2 Language Description

This section describes the syntax and semantics of Arrp informally through examples. A formal grammar for Arrp is available on the Arrp website. [1]

3.2.1 The Functional Layer

At the highest level, Arrp is a typical functional language with the following features:

- bindings of names to expressions and functions
- function applications (including partial)
- lambda abstractions (anonymous functions)
- higher-order functions (with other functions as parameters)

A program is a sequence of global name bindings. For example `n = e` binds the name `n` to the expression `e`. The scope of a global name is the entire program (global bindings may refer to each other). Arrp supports name bindings local to an expression using the forms `let n = e1 in e2` and `e2 where n = e1`, so that the name `n` is bound to the expression `e1` and is in scope of both `e1` and `e2`. In addition the form `n = e` can be used anywhere as an expression with the same value as `e`, so that the name `n` can be used recursively in `e`.

Lambda abstractions have the usual form `\x,y -> e` where `x` and `y` are function parameters and `e` is the body of the function. The syntax `f(x,y) = e` to define a named function is an alternative to `f = \x,y -> e`. The expression `f(x,y)` applies a function `f` to the arguments `x` and `y`. All functions are generic: they are instantiated at each application whereby types are inferred according to the arguments.

Here is an example:

```
sum_of_squares(x) = square(x) + square(x) where square(y) = y*y
```

Recursive functions are not supported, although some very common uses of functional recursion can be substituted with recursive arrays, as described in section 3.2.4. The reason for this restriction is that function applications are reduced at compile time, and the restriction is a simple way to ensure that the reduction terminates. Reduction of functions at compile time is required to allow translation of an entire program into a System of Affine Recurrence Equations, as explained in section 3.3.

[1] `http://arrp-lang.info`

3.2.2 Stream and Array Definition

At the core of the design of Arrp is the support for multi-dimensional streams. The foundation of this design is the view of streams as multi-dimensional arrays with one infinite dimension that represents time, which allows uniform treatment of infinite and finite dimensions.

An array is regarded as a function from indices to values of some other type. The domain of an array are integer points within a hyperrectangle - a Cartesian product of integer intervals. The lower bound in each dimension is 0. One dimension may have no upper bound (or the bound is infinity), thus representing time in a stream. For example, the following equation describes the domain of a two-dimensional array with the first dimension of infinite size and the second of size n:

$$([0, \infty) \cap \mathbb{Z}) \times ([0, n) \cap \mathbb{Z}) = \{ \langle i, j \rangle \mid i, j \in \mathbb{Z} \ \wedge \ 0 \leq i \ \wedge \ 0 \leq j < n \} \tag{3.1}$$

Since the lower bound of any dimension is always 0, we may also describe the domain simply with a tuple denoting its size: $\langle \infty, n \rangle$.

There are programming languages, such as ALPHA [52], which permit a broader range of array shapes: general polyhedra (integer points in a multi-dimensional space bounded by hyperplanes). In comparison, the restrictions of Arrp allow much simpler syntax and semantics as well as straightforward lifting of primitive operations to arrays, as explained in section 3.2.6.

An *array definition* in Arrp is an expression similar to a Haskell list comprehension:

```
y(x) = [~,5: t,i -> x[t] * i]
```

It begins with the domain specification, for example `~,5` meaning an array with the size $\langle \infty, 5 \rangle$. The body of the definition is similar to the body of a Haskell lambda abstraction. For example `t,i -> x[t] * i` means that each element of `y` at index $\langle t, i \rangle$ is equal to the element of `x` at index t multiplied by `i`.

Note that the array `x` in the above example is indexed using the index variable `t` which has no upper bound. Both `y` and `x` must therefore be of infinite size in the first dimension - in other words, streams. The example can be modified so that it is valid regardless of whether `x` is a finite or an infinite array. This is achieved using the expression `#x` denoting the size of `x` in the first dimension to limit the size of `y`:

```
y(x) = [#x,5: t,i -> x[t] * i]
```

The Arrp syntax for array definitions is close to the usual mathematical definitions of stream processing algorithms. Such definitions typically involve equations relating individual stream elements denoted using the index (subscript) operator. It is therefore easy to translate such definitions to Arrp. The flexibility of indexing the dimension of time also has important consequences for modularity and code reuse, as will be demonstrated in the following subsections.

3.2.3 Array Bounds Checking and Size Inference

One significant restriction in Arrp is that array sizes must be known at compile time. In addition, array index expressions are, for the most part, expected to be (quasi)-affine expressions (with exceptions described in Section 3.2.8). The main purpose of these restrictions is to enable optimization in the polyhedral model, but they have other benefits. For example, it can be statically checked that indexing is within array bounds, since interval analysis can easily be performed on affine indexing expressions and the minimum and maximum value can be checked against the statically known array size.

Additionally, interval analysis of indexing expressions can be used to infer the maximum possible size of an enclosing array definition, restricted by the size of the indexed array. Consider for example that an array `x` of size $\langle \infty, 10 \rangle$ is used in the expression `x[i,j] + x[i,j+1]`. We can easily see that this expression is only valid when $0 \leq i$ and $0 \leq j < 9$. Therefore, an array `y` with the maximum possible size using this expression can be defined without explicitly specifying the size:

```
x = [~,10: ...];   y = [i,j -> x[i,j] + x[i,j+1]]
```

A complete array size inference is not always possible. Consider this example:

```
y(x) = [i,j -> x[5*i+j]]
```

If the size of `x` is infinite, both `i` and `j` would have no upper bound, but we only allow a single such dimension. On the other hand, if the size of `x` is bounded, then the maximum size of `y` is ambiguous: any size $\langle m, n \rangle$ that satisfies $m-1+n-1 = \#x-1$ will do.

In these cases we require the user to specify a finite size for some dimensions. Sizes for other dimensions which the user would like inferred can be indicated with a placeholder symbol `_`. For example, `y` in the following example gathers each consecutive overlapping range of 5 elements of `x` into its second dimension:

```
y(x) = [_,5: i,j -> x[5*i+j]]
```

Using nested array definitions, as described in section 3.2.5, the size specification for the inferred dimension can be omitted altogether:

```
y(x) = [i -> [5:j -> x[5*i+j]]]
```

As a special case, the implicit upper bound of index variables which are not used in any index expression is infinity. An array of all natural numbers can thus be expressed simply as `[n -> n]`.

3.2.4 Array Recursion, Patterns and Guards

Recursive arrays are defined either by a recursive reference to the name to which they are bound [2], or using the keyword `this` which refers to the enclosing array definition. For example, here is a function that computes the integral of its input stream, in both forms:

```
integral(x) = [0 -> x[0];
               t -> this[t-1] + x[t]]

integral(x) = y = [0 -> x[0];
                   t -> y[t-1] + x[t]]
```

To terminate the recursion, `integral` is defined as a *piecewise function* with a special value at index 0. This is achieved using pattern matching and guards on index variables. The syntax is similar to patterns and guards in Haskell. For example, the pattern `3,j,4 -> e` defines a piece with expression `e` restricted to all indices $\langle i, j, k \rangle$ where $i = 3$ and $j = 4$. To avoid overlapping pieces, each pattern also implicitly excludes the domain of all the preceding patterns.

Since Arrp does not support recursive functions yet, some of their frequent uses can be substituted with recursive arrays. For example, one can define the sum of an array as the last element of the cumulative sum (equivalent to the integral define above). Note that this requires the array x to be finite:

```
sum(x) = csum[#x-1] where
  csum = [ 0 -> x[0];
           t -> csum[t-1] + x[t] ]
```

[2] In the current implementation of the Arrp compiler, type inference is not available for recursive array definitions using the bound name. A type annotation must be provided for such names. However, type inference in this case is theoretically possible.

The following is an example of patterns in a two-dimensional array:

```
x = [10,10:
  0,j -> 0;
  i,0 -> 0;
  i,j -> x[i-1,j-1] + 1
]
```

More complex shapes of array pieces can be achieved using guards. Guards further split the domain of a pattern into subdomains where indices satisfy certain equations. Again, to avoid overlap, each guard implicitly excludes the domains of the preceding guards in the same pattern. The final guard requires no explicit restriction on indices and matches all indices excluded by the preceding guards. For example, the following function multiplies each even element of `x` with `l` and each odd element with `r`:

```
y(l,r,x) = [n | n % 2 == 0 -> x[n] * l
              | x[n] * r ];
```

The pattern `n` (which matches every index) is split by two guards, one for indices where $n \bmod 2 = 0$ (even), and one for all other indices (odd).

It is required that the union of the domains of patterns and guards covers the entire domain of the array, or else some of its values would be undefined. By restricting guard expressions to quasi-affine expressions, this can be efficiently checked using integer linear algebra with libraries like ISL (Integer Set Library) [80].

Furthermore, we impose the restriction that individual array elements must not be mutually dependent. The following is an example of mutual dependence: the element at index n depends on the element at index $n + 1$, but that element also depends on the one at index $(n + 1) - 1 = n$:

```
[10: 0 -> 0; 9 -> 0;
     n -> this[n-1] + this[n+1]]
```

This restriction implies that array elements can be computed iteratively from known values of other elements (in contrast to solving a system of equations, for example).

3.2.5 Array Currying

By regarding a multi-dimensional array as a function with multiple integer arguments (or a tuple of integers), we can also consider array currying. An array can be seen as a function with a single argument which returns another array that is a function with a single argument which returns yet another array, and so on... Conversely, an

array containing other arrays may be seen as uncurried into a single multi-dimensional array.

The following is an example of array uncurrying: `sine` returns an array of size $\langle\infty\rangle$ (a sine wave signal). `harmonics` combines multiple sine waves (each at a different frequency) into a 2-dimensional array of size $\langle n, \infty\rangle$:

```
sine(freq) = [t -> sin(t*2*pi*freq)]
harmonics(freq, n) = [n: i -> sine(freq*(1+i))]
```

By analogy to partial function application - enabled by currying - we can consider partial array indexing: the result of indexing an array using less indices than it has dimensions is another array. This paradigm is popular in scientific computing, and is present for example in the languages MATLAB [3] and Octave [4], and the Numpy library for Python [5].

For example, a function like `integral` defined in Section 3.2.4 operates on the first dimension of a stream, but we can apply it to each harmonic in a collection using partial indexing:

```
hs = harmonics(440/sr,5);
hsi = [n -> integral(hs[n])];
```

Such an array definition, where the expression for elements is also of array type, is called *array nesting*. When arrays are nested, we must ensure that the resulting array maintains the shape of a hyperrectangle, since this property is important for compositionality (see section 3.2.6). Syntactically, it is possible to break this constraint, by defining the size of a nested array in dependence of the enclosing array index:

```
[i -> [i: j -> e]]
```

However, the restriction that the size of any array expression must be constant (see section 3.2.3) is intended to reject exactly such a case, so it is considered ill-typed. Furthermore, when an array is defined in multiple pieces using patterns or guards, the expression of each piece could be an array of different size. Such an array definition is therefore also ill-typed.

[3] `http://www.mathworks.com/products/matlab/`
[4] `https://www.gnu.org/software/octave/`
[5] `http://www.numpy.org/`

3.2.6 Pointwise Operations and Broadcasting

Primitive operations (e.g. addition, multiplication, as well as built-in functions like `sin`, `log`, etc..) operate pointwise on arrays, including infinite arrays. For example, assuming that `sine(f)` produces a sine wave stream with a frequency `f`, two sine waves can be mixed together like this:

```
sine(f1) + sine(f2);
```

Since array domains in Arrp are limited to hyperrectangles with all lower bounds equal to 0 (see section 3.2.2), the notion of a binary pointwise operation on equally-sized arrays is straightforward:

$$f(a, b)[i] = f(a[i], b[i]) \tag{3.2}$$

Still, arrays in a pointwise operation need not have the exact same size. There is an intuitive notion of a pointwise operation on arrays with different sizes, called "broadcasting". It is used in systems for scientific computing such as Octave and Numpy, and we find it useful in Arrp as well. Two arrays are compatible if their sizes are equal in each dimension, or the size of one is 1, or one does not have a corresponding dimension. In each dimension, the smaller array is virtually expanded to the size of the larger by replicating values.

More precisely, two arrays with different sizes $\langle a_0, a_1 ... a_m \rangle$ and $\langle b_0, b_1 ... b_n \rangle$ are compatible only if:

$$a_i = b_i \ \vee \ a_i = 1 \ \vee \ b_i = 1, \quad \forall i \in (0.. \min\{m, n\})$$

When pairing two arrays, the size of the result is $\langle c_0, c_1 ... c_l \rangle$, where $l = \max\{m, n\}$, and:

$$c_i = \begin{cases} \max\{a_i, b_i\} & \text{if } i \leq \min\{m, n\}, \\ a_i & \text{if } i > n, \\ b_i & \text{if } i > m. \end{cases}$$

The result at index $\vec{i}$ uses (or reuses) the value of an operand with size $\vec{s}$ at index $\beta(\vec{i}, \vec{s})$, defined as follows:

$$\beta(\langle i_0, i_1 ... i_m \rangle, \langle s_0, s_1 ... s_n \rangle) = \langle j_0, j_1 ... j_n \rangle \text{ where } j_k = \min\{i_k, s_k - 1\} \tag{3.3}$$

In combination with polymorphic functions, broadcasting significantly promotes code reuse. For example, consider the function `mix` below that sums two values `a` an `b` in different proportions, determined by the value `balance`. This function can be used to mix two primitive values or two streams (of any number of dimensions). Likewise, `balance` can be a primitive value - for a constant proportion, or a stream - for a proportion varying across the mixed stream elements:

```
mix(a,b,balance) = a * balance + b * (1-balance);
```

As another example of the value of broadcasting, recall the definition of the `sum` function:

```
sum(x) = csum[#x-1] where
    csum = [ 0 -> x[0]; t -> csum[t-1] + x[t] ]
```

Due to broadcasting in the application of `+` and partial array indexing, this function can be applied to an array of any number of dimensions, as long as the first dimension is finite. The result will have all the dimensions of the argument but the first. For example, instead of returning a multi-dimensional array of harmonics like the function `harmonics` in Section 3.2.5, we could sum them into a richer signal as in the following example. Note that the size of `hs` is $\langle n, \infty \rangle$, but the size of the result is just $\langle \infty \rangle$:

```
harmonics(freq, n) = sum(hs) where
    hs = [n: i -> sine(freq*(1+i))]
```

Broadcasting applies to array subdomains too. Expressions in different patterns and guards need not have the exact same size. The compatibility of sizes and the size of the resulting array follow the rules of broadcasting. When the resulting array is indexed, the equation 3.3 applies.

Furthermore, we extend the concept of broadcasting to Arrp's indexing operation, so that an expression like `x[i,j]` is actually valid for a one-dimensional array `x`, or even just a primitive value. This further increases code reuse. For example, consider the following definition of a periodic array:

```
phase(freq) = [
  0 -> 0;
  t -> let next = this[t-1] + freq[t-1] in
    if next >= 1 then next - 1 else next
]
```

This function can be used to generate a signal with a constant or a varying frequency. If the `freq` argument is a primitive value, it will be reused for all indices in the expression `freq[t-1]`.

3.2.7 Multi-rate Signal Processing

Since the time dimension of a signal can be indexed, multi-rate signal processing is achieved with simple arithmetic on indices:

```
downsample(x, factor) = [n -> x[n * factor]];

repeat(x, factor) = [n -> x[n / factor]];

upsample(x, factor) =
    [ n | n % factor == 0 -> x[n / factor]
        | 0 ]
```

In digital signal analysis, there are many algorithms which operate on consecutive, equally sized, possibly overlapping groups of signal elements (windows): for example short-time discrete Fourier transform, short-time autocorrelation, etc. Instead of implementing windowed iteration of the input in every such algorithm, we would rather abstract it into a function that transforms the input into a sequence of windows. This sequence will have an additional dimension and a reduced rate. A function operating on each window can then easily be mapped over this sequence.

In general, if the distance between windows (hop) is h, then the relation between the rate of the windowed stream r_w and the input rate r_i is $r_w = r_i/h$. This can easily be abstracted into a function in Arrp:

```
windows(size,hop,x) = [i -> [size:j -> x[hop*i + j]]]
```

Given a function `dft` that implements the discrete Fourier transform of a finite signal, we can easily implement `stft` - the short-time Fourier transform operating on portions of an infinite signal:

```
map(x,f) = [i -> f(x[i])]
stft(win,hop,x) = map(windows(win,hop,x), dft);
```

Many programming systems are restricted to one-dimensional streams (StreamIt [76], Faust [67], Pure Data [69], Max/MSP[6], SuperCollider[62]) or at most two-dimensional streams (multi-rate Faust [44], Marsyas [15]), and their approaches to windowed stream processing do not generalize well to multi-dimensional streams. In contrast, the function `windows`, as implemented in Arrp above, easily applies to a multi-dimensional stream `x`. For example, if the size of `x` is $\langle \infty, n \rangle$, then the size of `windows(s,h,x)` is $\langle \infty, s, n \rangle$, each window having the size $\langle s, n \rangle$.

[6] `https://cycling74.com/products/max`

3.2.8 Non-affine Index Expressions

While affine array index expressions enable detailed static analysis as described in Section 3.2.3, index expressions are not limited to such expressions. More complex expressions, including data-dependent expressions are supported. However, with such expressions, the power of static analysis may be limited.

For example, if the programmer requests the size of an array to be automatically inferred, but the inference depends on an array indexing expression with unknown bounds, then the compiler can simply request the user to specify the array size explicitly.

When the bounds of an index expression are not known, but the indexed array dimension is finite, the compiler can assume without much difficulty that any element in that dimension may be accessed during the execution of the program. This is sufficient for example to implement a variable-frequency wavetable oscillator. The following is a crude sketch assuming that `table` is a finite-size array and `freq` is a stream of integer frequency values:

```
oscillator(table, freq) =
    [t -> table[i] where i = phase(freq, #table)[t]];
phase(freq, max) =
    [0 -> 0;
     t -> (this[t-1] + freq[t]) % max;];
```

If the indexed array dimension is infinite and the index is unbounded, an implementation would require random access to an infinite amount of elements at all times which is obviously not feasible. In such a case, a compiler for Arrp should issue an error. However, the user can assist the compiler in determining the bounds of an indexing expression. In particular, the `min` and `max` functions can be used to impose the bounds explicitly. This enables the implementation of a signal delay operator with a variable delay value. The following is a simplified example, assuming that `x` is a signal to be delayed, and `d` is a stream of delay values, limited to the range $(0, 10)$:

```
delay(x,d) = [n -> x[n + max(0, min(10, d[n]))]];
```

A more useful example using this feature is the fractional delay implementation in Appendix B.3.2.

3.2.9 Interfacing With the World

A stream processing program would be of little use if it could not exchange information with the outside world. In Arrp, program input and output is specified simply by designating named values of an elementary type or array type as inputs or outputs. The exact way that these values are exchanged with the outside world is left to an implementation of the language.

The output of a program is the value bound to the name `main`. Inputs are declared without a bound value, but with an expected type. In the following example, an input named `x` is declared with the type of array of 64 bit floating point numbers with a size $\langle \infty, 5 \rangle$. The output `main` multiplies each element of `x` by 2:

```
input x :: [~,5]real64;
main = x*2;
```

3.3 Reduction to Affine Recurrence Equations

This section describes the reduction of Arrp to a system of affine recurrence equations (SARE). In this form, a program can be efficiently captured and analyzed in the *polyhedral model*, which allows generation of efficient executable code, as described in Chapter 4.

A multi-dimensional recurrence equation has the following general form:

$$\forall \vec{i} \in D : s(\vec{i}) = e(\vec{i}, s_1(m_1(\vec{i})), s_2(m_2(\vec{i})), ...s_n(m_n(\vec{i}))) \tag{3.4}$$

where s is a multi-dimensional sequence (array), and every s_i is another such sequence or s itself. A d-dimensional sequence is a function $I \to V$ where an element of the domain $I \subset \mathbb{Z}^d$ is called an *index*, and V is some arbitrary value type. A recurrence equation defines the values of a sequence for a subset of indices $D \subseteq I$. D is called the domain of the equation. The expression e is assumed to have a time complexity of $O(1)$, and strictly depend on the arguments shown above. Often in literature, e is assumed to only depend on the values of other sequences, although for convenience when modeling Arrp, we also include a dependence on the index $\vec{i}$ itself. Every m_i maps an index $\vec{i}$ of s to some index of s_i.

A system of *affine* recurrence equations imposes additional restrictions to this general form. A domain D of an equation must be a $\mathbb{Z}$-polyhedron - a set of integer

points satisfying a set of affine constraints. For example:

$$D = \{ \langle i, j \rangle \mid i, j \in \mathbb{Z} \ \wedge \ i, j \geq 0 \ \wedge \ i + j < 10 \} \tag{3.5}$$

Furthermore, the index mappings m_i must be affine functions. Contemporary tools for integer set manipulations, such as the Integer Set Library (ISL) [80], also support quasi-affine index mappings and domain constraints, meaning that division and modulo of an index variable with a constant divisor is allowed.

Most Arrp programs can be reduced to a SARE, since array sizes in Arrp are statically known, piecewise array definitions use quasi-affine index constraints, and array indexing expressions are typically quasi-affine. However, some uses of non-affine array indexing are also supported, as described in Section 3.2.8. The reduction of Arrp presented in the rest of this section is agnostic of the precise nature of array index expressions. It results in a pure SARE as long as all index expressions are affine, and otherwise it differs from a SARE only in the nature of these expressions. It is worth noting that even non-affine index expressions can be approximated using affine bounds, as described in Section 3.2.8. This allows the representation of complex and data-dependent index expressions in the polyhedral model at least in an approximate form while significantly increasing the expressivity of the language.

3.3.1 Reduction of Function Applications

A program is first reduced under the usual β-reduction rules of the lambda calculus. The reduction begins at the name *main* representing the program output, reducing all of its function applications. The same is done for the named expressions referenced in *main*, and their own references, and so on. Since functional recursion is not allowed, this process will necessarily terminate. *main* must reduce to a non-function type (a primitive or array type). Function definitions are ultimately removed from the program; only named expressions of primitive and array types are preserved.

For example:

```
add(a,b) = a + b;
map(f,x) = [i -> f(x[i])]
iterate(f,v) = [0 -> v; n -> f(this[n-1])]
x = iterate(add(1),0);
main =  map(add(2),x);
```

...reduces to:

```
x = [0 -> 0; n -> 1 + this[n-1]];
main = [i -> 2 + x[i]];
```

In order not to repeat computation, a function argument is turned into a local name if it is referenced more than once, and it is not a simple reference itself:

```
square(x) = x*x;
f(x) = square(x + 2);
```

...reduces to:

```
f(x) = let a = x + 2 in x * x;
```

3.3.2 Arrays, Patterns, Guards

Each named expression in Arrp is represented as a sequence in a SARE defined with one or more equations. If the expression is a primitive value, the sequence has a single element. Here is an example of Arrp code, followed by its representation as a SARE:

```
c = 3;
a = [~,5: i,j -> (i+j)*c]
```

$$c(0) = 3 \tag{3.6}$$

$$a(i,j) = (i+j) \cdot c(0), \quad 0 \leq i \ \wedge \ 0 \leq j < 5 \tag{3.7}$$

A few remarks about the notation are in order. While Eq. 3.4 defines a sequence as a function on integer tuples, for example $s(\langle i,j \rangle)$, we consider it equivalent to a function with tuple elements as separate parameters $s(i,j)$, to simplify notation. We choose a more convenient way to denote the domain of a recurrence equation, instead of the exact form shown in Eq. 3.4. Specifically, if an index is fixed at a single value, the value is used directly in the equation instead, as in Eq. 3.6 above. Otherwise, the affine constraints defining the domain follow the equation, as in Eq. 3.7 above.

Arrays with patterns and guards become sequences defined with multiple equations. The constraints on indices imposed by patterns and guards are reflected in the equation domains:

```
a = [~,10:
  x,0 -> 0;
  x,y | x+y < 10 -> 1
      | 2
]
```

$$a(x,0) = 0, \quad 0 \leq x \tag{3.8}$$

$$a(x,y) = 1, \quad 0 \leq x \ \wedge \ 0 \leq y < 10 \ \wedge \ \neg(y = 0) \ \wedge \ x + y < 10 \tag{3.9}$$

$$a(x,y) = 2, \quad 0 \leq x \ \wedge \ 0 \leq y < 10 \ \wedge \ \neg(y = 0) \ \wedge \ \neg(x + y < 10) \tag{3.10}$$

3.3.3 Nested Arrays

An array definition nested inside another is merged into a single array definition. Its domain is the product of the domains of the two arrays. The guards and patterns of the two arrays are merged. Any recursive applications of the nested array are extended with additional arguments:

```
f(x) = [~: 0 -> 0; n -> this[n-1] + x];
a = [5: i -> f(i)]
```

... combines to:

```
a = [5,~: i,0 -> 0; i,n -> this[i, n-1] + i]]
```

$$a(i,0) = 0, \qquad 0 \leq i < 5 \tag{3.11}$$

$$a(i,n) = a(i,n-1) + i, \qquad 0 \leq i < 5 \ \wedge \ 0 < n \tag{3.12}$$

3.3.4 Array References

An array reference in Arrp may be only partially applied or not applied at all. For example, a is unapplied in the expression of b:

```
a = [5: ...]; b = [10: i -> a]
```

In such case, the reference is first converted into a full application in the process known as η-abstraction in the lambda calculus:

```
a = [5: ...]; b = [10: i -> [5: j -> a[j]]
```

The nested arrays are then combined as described in section 3.3.3.

3.3.5 Local Names

A local name, such as `c` in the following example, becomes a sequence on its own, in a process known as *lambda lifting* in the lambda calculus. The domain of the resulting sequence must include all index variables that appear in the lifted expression, and the restrictions on those variables must be preserved or else the sequence could be

ill-defined. In the following example, the definition of `c` uses the index variable `i` in a context where `i` is restricted to $0 \leq i < 5$ which becomes the domain of the sequence c. Once a local name is lifted, its references are treated as in section 3.3.4:

```
a = [5: ...];
b = [10,5: i,j
  | i < 5 -> c + j where c = a[i] + 1
  | i + j ];
```

$$a(i) = ..., \qquad 0 \leq i < 5 \tag{3.13}$$

$$b(i,j) = c(i) + j, \qquad 0 \leq i < 5 \ \wedge \ 0 \leq j < 5 \tag{3.14}$$

$$b(i,j) = i + j, \qquad 5 \leq i < 10 \ \wedge \ 0 \leq j < 5 \tag{3.15}$$

$$c(i) = a(i) + 1, \qquad 0 \leq i < 5 \tag{3.16}$$

If a local name contains any free recursive references to an enclosing array, the resulting sequence keeps the original referent. The two sequences thus become mutually referential.

```
a = [~: 0 -> 0;
        n -> b % 5 where b = this[n-1] + 1 ]
```

$$a(0) = 0 \tag{3.17}$$

$$a(n) = b(n) \bmod 5, \qquad 1 \leq n \tag{3.18}$$

$$b(n) = a(n-1) + 1, \qquad 1 \leq n \tag{3.19}$$

3.3.6 Pointwise Operations and Broadcasting

If the operands to a primitive operation are non-recursive arrays, then we can reduce the operation to a single array with the combined expressions of the operand arrays, by straightforward application of the equations in section 3.2.6 on broadcasting:

```
a = [~,1,5: i,j,k -> e1(i,j,k)]
  + [~,10: i,j -> e2(i,j)]
```

...reduces to:

```
a = [~,10,5: i,j,k -> e1(i,0,k) + e2(i,j)]
```

A recursive array operand can not be merged in this way. In the following example `a1` denotes pointwise multiplication of 2 with each element of the recursive array

operand. However, a simple reduction as described above would result in the array `a2` which represents a different sequence:

```
a1 = 2 * [0 -> 0; n ->      this[n-1] + 1 ];
a2 =     [0 -> 0; n -> 2 * (this[n-1] + 1)];
```

The array `a1` represents the sequence $0, 2, 4, 8, ...$, whereas the array `a2` represents the sequence $0, 2, 6, 14, ...$ This can be remedied by lambda lifting recursive operand arrays (see section 3.3.5). The array `a1` is thus safely reduced to:

```
a1 = [n -> 2 * b[n]];
b = [0 -> 0; n -> this[n-1] + 1]
```

If multiple operand arrays have patterns or guards which translate to sequences defined with multiple equations, the merged sequence performing the pointwise operation will have an equation for each combination of the equations of the individual arrays. In general, if we have n operand arrays, each translating to m equations, the resulting sequence will have n^m equations. This combinatorial explosion can also be remedied by lambda lifting of the operand arrays whereby the operands reduce to arrays with a single equation.

3.4 Conclusions

The new language Arrp presented in this chapter effectively combines a syntax based on recurrence equations with the functional paradigm. The recurrence equation syntax allows almost literal translation of many stream processing algorithms from a mathematical definition to a computer program. It also naturally supports the expression of multi-dimensional and multi-rate algorithms. Moreover, recurrence equations are a good foundation for abstraction and code reuse enabled by polymorphic and higher order functions.

A number of improvements to the language are envisioned. While some common uses of recursive functions can currently be substituted with recursive arrays, structural recursion would require recursive functions. These are currently not supported to ensure termination of function reductions in the compiler. Support for non-inlined functions would reduce the amount of executable code and allow separate compilation of components of a program. Another interesting avenue is the integration of record-type data structures and their representation in the polyhedral model.

While the current support for data-dependent array indexing significantly improves expressivity, many stream processing applications would benefit from more dynamic behaviors. This includes for example dynamically resizing arrays or even dynamically instantiating program components. A complete Arrp program can already be integrated as an actor in a larger dataflow system supporting dynamic behavior. However, making Arrp even more widely applicable would be useful, because reducing the number of languages involved in a project can improve code reusability, code quality and ultimately programmer productivity.

The reduction of Arrp to a system of affine recurrence equations described in this chapter enables its efficient translation to executable code using the polyhedral framework, which is described in the next chapter. If non-affine and data-dependent array index expressions are used in the program, they will remain in the resulting system of equations. Nevertheless, it was shown in this chapter how the range of such expressions can be safely approximated using affine bounds, which allows their successful handling in the polyhedral model.

Chapter 4

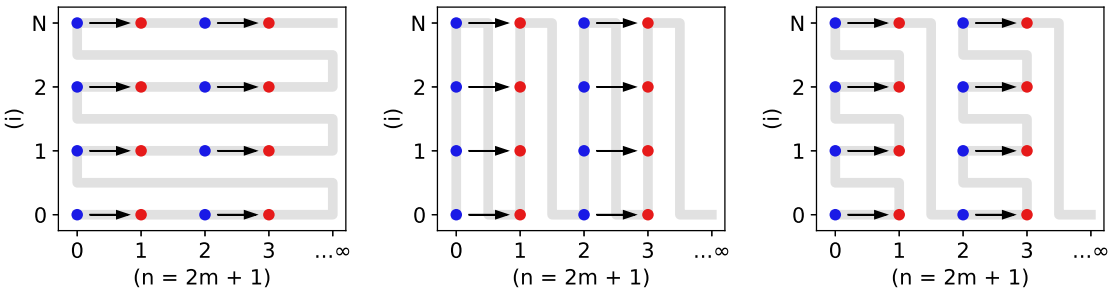
Polyhedral Compilation

4.1 Introduction

Section 2.1.2 has introduced the challenges involved in the compilation of streaming languages like Arrp based on multi-dimensional recurrence equations. The polyhedral model is uniquely suitable to address these challenges due to its power of analysis and transformation of computations over multi-dimensional arrays. This work specifically targets code generation for general-purpose processors. The primary role of the polyhedral model in the compilation process is to facilitate generation of imperative code which can then be more easily translated to machine code for such hardware. In addition, the polyhedral model enables static memory allocation and offers powerful optimizations that improve data locality and enable efficient parallelization.

This chapter presents a complete method within the polyhedral model for translation of streaming programs from multi-dimensional recurrence equations to imperative code while accommodating the existing polyhedral optimizations. This method relies heavily on prior work in the polyhedral model. However, there are unique challenges in code generation from unbounded recurrence equations representing infinite streams which have not been addressed previously. These challenges are addressed by the novel polyhedral techniques introduced in this chapter.

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 4.1a depicts a schedule for this program computed



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

Figure 4.1: Examples of polyhedral schedules.

by the popular scheduling algorithm due to [12] 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 dependences. 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 4.1c is a periodically tiled variant of the schedule in Fig. 4.1a - 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 in Figure 4.1b, existing techniques such as [33] create infeasible loops to scan schedule points, like in Figure 4.2a. Note that the iterator `c1` has no upper bound and would overflow at some point. The *periodic tiling* technique presented in this chapter 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 4.2b 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 con-

```
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);
        }
    }
}
```

(a) Code for the entire schedule in Fig. 4.1b.

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

(b) Code for a period of the schedule in Fig. 4.1c.

Figure 4.2: Code for schedules in Figure 4.1.

strained 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* ([18]) and in particular algorithms due to [61] and [7].

This chapter is structured as follows:

- Section 4.2 covers the background on polyhedral compilation.
- Section 4.3 formally defines the problem addressed in this chapter.
- Section 4.4 presents the periodic schedule tiling technique. It also describes how it can be combined with existing polyhedral schedule optimizations for efficient tiling and parallelization.
- Section 4.5 proves that modular mappings achieve finite storage in combination with periodically tiled schedules. It also presents our extensions of existing code generation techniques and integration of well-known buffer optimizations into our framework.

4.2 Background

4.2.1 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 $\langle v_1, ..., v_d \rangle \in \mathbb{R}^d$ that satisfy a finite number of affine inequalities in the variables v_i:

Definition 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 chapter, we refer to such a set simply as a *polyhedron* unless explicitly stated otherwise.

Definition 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 1 (Minkowski-Weyl Theorem)**.** *For every convex polyhedron $P \subset \mathbb{R}^d$, there is a finite number of vectors $\vec{v_1}, ..., \vec{v_n}$ and $\vec{r_1}, ..., \vec{r_m}$ such that $P = conv(\vec{v_1}, ..., \vec{v_n}) + cone(\vec{r_1}, ..., \vec{r_m})$.*

In the above, $conv(\vec{v_1}, ..., \vec{v_n}) = \{ \sum_{i=1}^{n} \gamma_i \vec{v_i} \mid \gamma_i \geq 0 \wedge \sum_{i=1}^{n} \gamma_i = 1 \}$ is the bounded set of convex combinations of vectors v_i (a polytope), $cone(\vec{r_1}, ..., \vec{r_m}) = \{ \sum_{i=1}^{m} \lambda_i \vec{r_i} \mid \lambda_i \geq 0 \}$ 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 3 (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 1. *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 $\langle v_1, ...v_n \rangle \mapsto \langle v_l, ...v_m \rangle$ 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 chapter, 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 1 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, etc. We also write $\langle i_1, ...i_n \,|\, j_1, ...j_m \rangle$ 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 $\mathrm{ran}(Q)$.

4.2.2 Polyhedral Model

In the polyhedral model, a program is represented as a set of statements $s_1, s_2, ..., s_n$ that read and write values in multi-dimensional arrays $a_1, a_2, ..., a_m$. 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 $\left\{ \langle \vec{i}, q(\vec{i}) \rangle \mid \vec{i} \in D_s \right\}$.

In this work, we focus on the compilation of affine recurrence equations (SARE) where the polyhedral model plays an essential role. The latter are frequently used as a mathematical description of stream processing algorithms (including in this chapter). As described in Section 3.3, the language Arrp presented in this dissertation can be reduced to a SARE, and so can other languages like ALPHA [52] and PAULA [35]. Fortunately, a polyhedral model can be easily derived from a SARE. Specifically, in such a polyhedral model, streams are represented as arrays with unbounded domains, and each array element is written only by a single instance of a single statement. Therefore, these properties are assumed in the rest of this chapter.

However, this work is not limited to the compilation of languages based on recurrence equations. A suitable polyhedral model can be derived from a variety of languages and other models of computation. It can be derived for example from the synchronous dataflow model and its extensions [77] including the multi-dimensional extensions [49]. It can also be derived from imperative languages, specifically static-affine nested loop programs (SANLP) using the same approach as in the derivation of Polyhedral Process Networks [82].

Since SARE have a special place in this dissertation, we recall their definition and describe how a polyhedral model is derived. A system of recurrence equations consists of a set of variables V where each variable a is a function defined by a set of equations on disjoint domains $D_a^1, ..., D_a^n$ with the following general form:

$$\forall \vec{i} \in D_a^j : a(\vec{i}) = f^j(b_1(r_1(\vec{i})), ..., b_m(r_m(\vec{i}))) \tag{4.1}$$

where $b_1, ..., b_m$ are variables in V. A system of *affine* recurrence equations is one where each equation domain D_a^j is a polyhedron and the index mappings $r_1, ..., r_m$ are affine functions. The equation domains and index mappings can also be described using quasi-affine functions (involving a division or the remainder of a division of a variable with a constant divisor), since such sets can be represented as projections of polyhedra, as described in Section 4.2.1.

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: $\{ \langle \vec{i}, r_i(\vec{i}) \rangle \mid \vec{i} \in D_a^j \}$. It is worth noting that some programs which are not strictly conforming to the SARE definition can be represented in the polyhedral model. In particular, if index mapping expressions r_i are not affine functions or if they depend on additional information like values of other arrays, it may still be possible to approximate the range of such expressions using affine bounds (i.e. as a polyhedron). Examples of such cases in Arrp are given in Section 3.2.8. Such index mappings are represented in the polyhedral model as follows. Let R_i denote a polyhedron approximating the range of r_i. Then a corresponding array read relation is $\{ \langle \vec{i}, \vec{j} \rangle \mid \vec{i} \in D_a^j \ \wedge \ \vec{j} \in R_i \}$.

4.2.3 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. [72]). 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 ([24], [25]). A popular approach used in the Pluto optimizer [13] finds an abstract multi-dimensional affine mapping in a single optimization framework; any dimension which does not carry dependences can be interpreted as a distribution in space and others as a distribution in time. We use a generic definition of a schedule which accommodates all these approaches.

Definition 4 (Polyhedral Schedule)**.** A polyhedral schedule Φ is a set of multi-dimensional functions $\phi_s : \mathbb{Z}^d \rightarrow \mathbb{Z}^n$. 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:

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

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 5 (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 dependences 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 6 (Dependence relation)**.** Given a write relation W and a read relation R, there is a dependence relation:

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

Definition 7 (Valid schedule)**.** A schedule is *valid* when it *satisfies* all dependences 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 8 (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 [41], 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 [21, 35]. In contrast, tiling after space-time mapping has also been proposed [30]. The popular Pluto algorithm [13, 1] 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 [50].

This dissertation 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 4. 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 [31], GCC/Graphite [79], PPCG [83].

4.2.4 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 which 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 dependences 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:

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

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 [18]. The so-called *successive modulo technique* (SM) introduced by [61] finds a suitable $\vec{e}$ when M is identity or otherwise given. We paraphrase its definition as given in [7]:

Definition 9 (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 which have a distance larger than 0 in that dimension, since they are guaranteed to be satisfied. Repeat this for each following dimension.

More recently, [7] 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 [70, 33], 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 10 (Polyhedral AST)**.** Polyhedral AST is an imperative AST that executes all statement instances of a polyhedral model with a given polyhedral schedule and storage optimization, generated using algorithms such as [70, 33].

We will introduce an extension of these techniques to generate an AST without unbounded loop induction variables even from polyhedral models of stream processing with unbounded array and statement domains.

4.3 Problem Statement

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

Definition 11 (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:

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

Definition 12 (Productive Schedule)**.** A schedule is *productive* if each point in its range has a finite number of lexicographically preceding points.

Definition 13 (Rate-consistent Schedule)**.** A schedule is *rate-consistent* if all unbounded access schedules for the same array have one and the same infinite direction.

The problem addressed in this chapter can be formally stated as follows:

Definition 14 (Polyhedral Compilation of a Stream Processing Program)**.** Given a polyhedral model of a stream processing program, and a schedule, 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.

Definition 15 (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 rest of this section discusses and justifies the restrictions on the admissible input. 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 examples in Figure 4.2: 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 13 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 $\langle 2i, i \rangle$ (not equivalent). 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 $\langle 2i, t \rangle$ (not equivalent). 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. [1]

A rate-consistent schedule together with a connected dependence graph implies that the ranges of all schedule statements have one and the same infinite direction. If the dependence graph is not connected, the program might as well be separated into separate programs, or artificial dependences may be introduced to "synchronize" independent parts of the program. The final restriction in Def. 15 reflects a reasonable

[1] A proof and extended discussion regarding Pluto are published in an Addendum [57] to the first publication of this content.

expectation that dependencies do not predominantly point opposite to the infinite direction of the schedule.

4.4 Periodic Schedule Tiling

We present a transformation of an admissible schedule as defined in Def. 15 which ensures that the resulting schedule is productive, thus satisfying the requirement 1 of our problem. 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 schedule into equally shaped finite tiles called *periods* with desirable properties. First, we define periodic tiling of any set in general in Section 4.4.1. Then, in Section 4.4.2, we define periodic schedule tiling in particular, prove some of its properties and provide an algorithm to find one. Finally, in Section 4.4.3 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 chapter that the infinite direction of a schedule is positive in all dimensions, to simplify notation.

4.4.1 Periodic Tiling

We define one-dimensional tiling similarly to [41] 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:

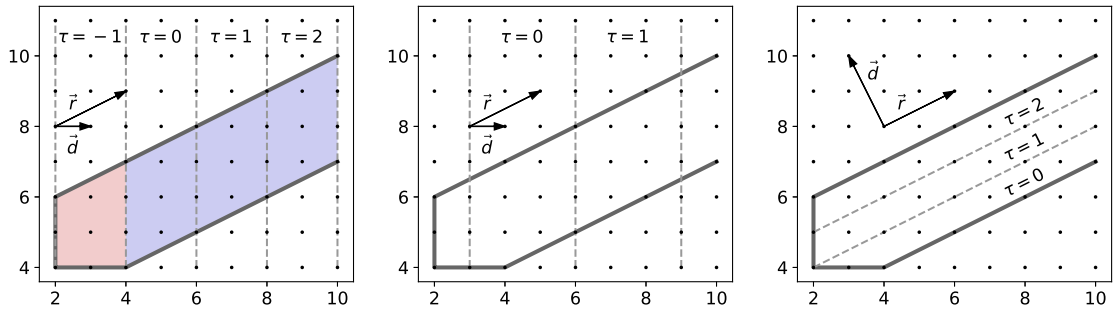
Definition 16 (Tiling)**.** A *tiling* of a set $S \subset \mathbb{Z}^d$ is a tuple $\langle \vec{d}, \mu, \sigma \rangle$. $\vec{d} \in \mathbb{Z}^d$ is a *tiling direction*, $\mu \in \mathbb{Z}$ is a *tiling offset*, and $\sigma \in \mathbb{Z}$ is a *tile size*. The function $\text{tile}(\vec{i}) = \lfloor (\vec{d} \cdot \vec{i} - \mu)/\sigma \rfloor$ 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 17 (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:

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

Equivalently, consider a projection $P' = J(P)$ and call its vertices $V' = \{\, J(V) \mid \vec{v} \in V \,\}$. A *periodic tiling* of P' is a tiling satisfying the conditions above if V is replaced



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

Figure 4.3: A periodic tiling 4.3a and two tilings which are not periodic: 4.3b and 4.3c.

by V' and $\vec{r}$ with a ray of P'. All non-empty tiles for $\tau < 0$ are called *prologue tiles*, and all non-empty tiles for $\tau \geq 0$ are called *periodic tiles*. For generality, we also define a periodic tiling of a bounded polyhedron or its projection as any tiling which satisfies only the condition 2 above.

Corollary 2. *Following from Def. 3 and Def. 17: 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 4.3 depicts three different tilings of a polyhedron. 4.3a 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. 4.3b is not a periodic tiling: μ is too small and σ is not an integer multiple of $\vec{d} \cdot \vec{r}$. 4.3c 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 which we deem evident, although their mathematical proof is beyond the scope of this work. As shown later in this chapter, these properties turn out to be useful in solving the problem at the focus of this work:

Lemma 1. *A periodic tiling has the following properties:*

- *There is a finite number of prologue tiles, and (in an unbounded polyhedron) 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.*

4.4.2 Periodic Schedule Tiling

Definition 18 (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:

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

A periodic tiling might turn a valid schedule into an invalid schedule with respect to data dependences (Def. 7). Due to [41], we have the following sufficient condition for validity:

Lemma 2. *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 2 (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 Def. 17 and Lemma 2. 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 $\{ \langle x_1, ... \rangle \mid x_1 \geq 0 \}$ since only in that case all dependences 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 4.4 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 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 Fig. 4.4b. This schedule undergoes periodic tiling, resulting in the schedule in Fig. 4.4c. To help illustrate the procedure, vertices of some of the schedules and rays of some of the access schedules are given in Fig. 4.4d. Fig. 4.5 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.

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 work, Section 4.4.3 discusses how the tiling direction, offset and size relate to various performance objectives.

The algorithm includes a few auxiliary procedures. The procedure SmallestRay returns the smallest ray of an integer set with a single infinite direction. The procedure IsBounded returns whether an integer set is bounded. The procedure Vertices returns

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 $0 \le i < N$.

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

(b) Φ = A schedule computed by ISL.

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

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

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

Figure 4.4: Example of periodically tiled schedule

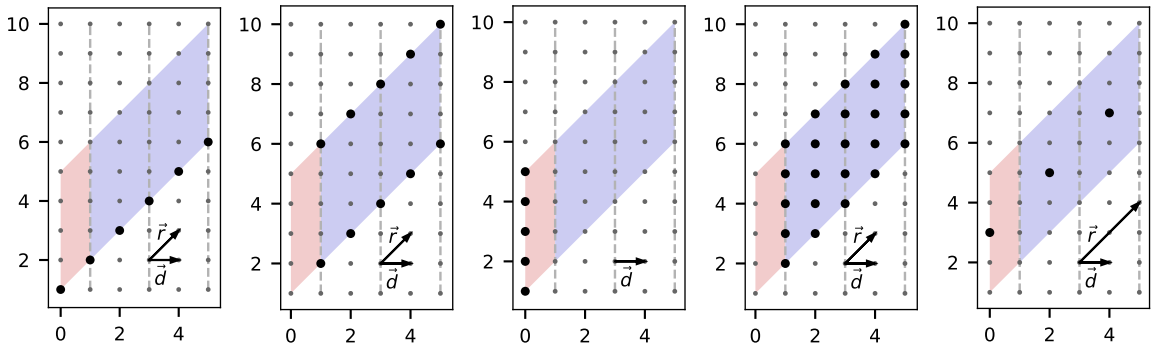


Figure 4.5: First two dimensions of schedule Φ introduced in Figure 4.4, with $N = 7$. Dashed lines indicate tile boundaries in periodically tiled schedule Φ'. The prologue tile is red and periodic tiles are blue. Each subfigure highlights schedule for a particular statement using bold dots; from left to right: ϕ_{s_1}, $(\phi_{s_2} \cup \phi_{s_3})$, ϕ_{s_4}, ϕ_{s_5}, ϕ_{s_6}.

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 of our problem:

Theorem 3. *A periodically tiled schedule is productive.*

Proof. According to the definition of periodically tiled schedule (18) and the schedule execution order (see Section 4.2.3), 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 1. 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 which support the proposed storage allocation (Section 4.5.1) and code generation techniques (Section 4.5.2).

Lemma 3. *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 1. 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.*

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 L^1-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
```

Proof. We know from Lemma 1 that each pair of corresponding schedule points has a distance $\vec{r}$. So, the above is true if there exists $\vec{o}$ such that $\langle \vec{a}, \vec{t} \rangle \in \phi_A \implies \langle \vec{a}, \vec{t} \rangle + \langle \vec{o}, \vec{r} \rangle \in \phi_A$. This is indeed true, since there exists $\langle \vec{o}, \vec{r} \rangle$ which is a ray of ϕ_A (from Def. 18) and it follows from Def. 3. □

Lemma 4. *All access relations of an array have a common distance $\vec{o}$ described in Lemma 3.*

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 (Def. 13). So, $\langle \vec{o}_i, \vec{r} \rangle = k \langle \vec{o}_j, \vec{r} \rangle$, which is only true if $k = 1$ and $\vec{o}_i = \vec{o}_j$. □

4.4.3 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 which improves data locality and parallelism while also considering the effect on input-output latency

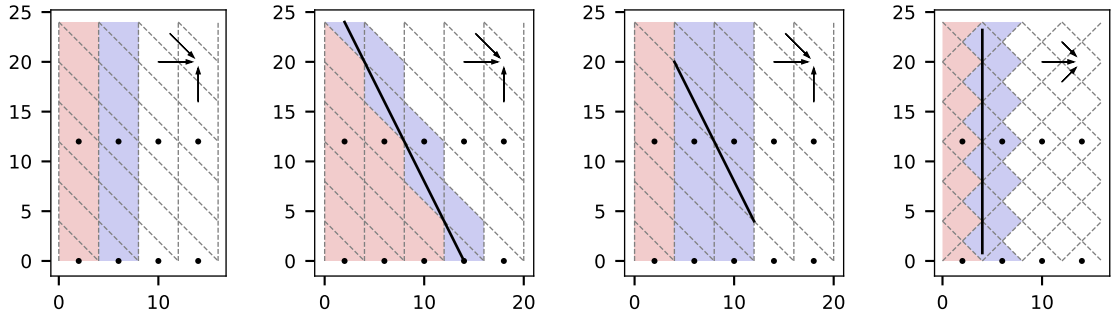


Figure 4.6: Combination of periodic tiling and performance tiling for program in Fig. 4.4 with $N = 24$. Horizontal axis represents $n = 2m$ and vertical i. Prologue tiles in red, the first periodic tile in blue. Bottom row of dots marks sub-tiles where input occurs, and top row marks sub-tiles where output occurs. Arrows depict dependences between tiles. The bar connects a set of sub-tiles within a period that can execute in parallel.

- an important aspect of stream processing systems. It is sufficient for our goal 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 work.

We use the schedule in Figure 4.4b 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 4.6a (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 Fig. 4.6a. 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 dependences. This is a known problem, and [13] provides 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 Fig. 4.6b. Within each period, we have two schedule hyperplanes in the direction which carries all dependences, and so all subtiles on such

a hyperplane can be executed in parallel (an example is marked with the thick black line).

There are alternative periodic tilings though. Note that the schedule in Fig. 4.6b increases input-output latency in comparison to Fig. 4.6a - 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 Fig. 4.6c for $\sigma = 2$. An even better solution is possible with an altogether different schedule based on the "diamond tiling" presented in [4]. 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 Fig. 4.6d. The direction $\langle 1, 1, 0, 0, 0 \rangle$ now carries all the inter-tile dependences. 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.

4.5 Storage Allocation and Code Generation

4.5.1 Finite Storage Using Modular Mapping

Storage optimization is critical for streaming applications: since arrays are infinite, a trivial storage allocation that maps each array element into a unique memory location would require an infinite amount of memory. We prove that the scheduling technique presented in this chapter always admits finite storage using the well-known intra-array storage optimization techniques. Specifically:

Theorem 4. *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 Def. 9 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). 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. □

4.5.2 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 19 (Periodic Polyhedral AST)**.** An AST that consists of an infinite sequence $(q, p, p, p, ...)$, 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.

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

Theorem 5. *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 which preserves the order relation (Lemma 1) 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 3, 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:

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

□

Corollary 3. *In a Periodic Polyhedral AST, there are no unbounded loop iterators.*

Proof. This follows from the definition of Periodic Polyhedral AST. □

4.5.3 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 4.5.2, 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, 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, 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, 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.

4.6 Conclusions

To the best of my knowledge, this work represents the first complete code generation solution for polyhedral models with unbounded domains. Specifically, the compilation method presented here supports polyhedral models derived from stream processing programs in the form of a system of affine recurrence equations (SARE) - which includes the Arrp language. The result of the compilation is imperative code with statically allocated memory.

The central part of the proposed compilation method is a novel polyhedral scheduling technique called *periodic tiling*. This techniques integrates well with existing polyhedral scheduling techniques which provide data locality optimizations and expose parallelism. Significant performance benefits are achievable using simple heuristics to select periodic tiling parameters presented in this chapter, as shown by experiments described in Chapter 6. However, automatically optimizing periodic tiling parameters

remains an interesting challenge. It would also be beneficial to extend periodic tiling to parametric polyhedral models (models with symbolic parameters for array bounds etc.). That would support some forms of dynamic reconfiguration of streaming programs. While it should not take much effort to accommodate parametric models in the theory presented in this dissertation, such models might play a more significant role in optimizing periodic tiling parameters.

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. Chapter 6 describes an experimental evaluation of these techniques.

While this work is particularly useful in compiling languages like Arrp, we believe it 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 infinite program provides more opportunities for optimization, for example tiling over time, merging stream operators, etc. A variety of source languages may therefore benefit from this work as long as a translation of whole programs into the unbounded polyhedral model exists. The performance benefit of the proposed techniques for other source languages deserves a more exhaustive evaluation and comparison with existing techniques.

This compilation method also supports generation of code that fits well into the actor paradigm. This is enabled in particular by the extraction of prologue and period parts of the schedule. For example, in the StreamIt dataflow model [76, 46], the prologue would correspond to the *prework* function executed once at initialization, while the period would correspond to the *work* function representing the action repeated at every firing. This allows integration of programs compiled and optimized using this method into a larger stream processing system. In particular, this method could be used to optimize static parts of a program modeled on a larger scale in a dynamic dataflow model.

Chapter 5

Case Studies

This chapter compares Arrp to two other textual language for stream processing in order to better illustrate its design considerations and highlight its unique place among related languages. As a representative of the point-free style of stream programming, we consider the Faust language [67] which is extensively used for sound synthesis and processing. As a representative of the actor paradigm, we consider the StreamIt language [76] which has been the context of much research in stream programming.

The potentials and limitations of these languages are explored using a number of concrete program examples. We focus in particular on how natural and straightforward it is to translate various algorithms from a mathematical definition to an implementation in these languages. We also investigate to what extent code can be reused between algorithms and when expanding certain algorithms into multiple dimensions. Moreover, we evaluate how well multiple dimensions and high-volume streams are handled by these languages and their compilers.

Faust has a simple notion of streams as infinite sequences of real numbers. In the current public release, it does not support multi-rate programs (it uses the Homogeneous Synchronous Dataflow [9] model), although research has been done into its extension to multi-rate programs [44]. Faust is a functional language and it primarily supports the extreme point-free style of stream operator composition without mentioning streams. Every expression denotes a stream operator with potentially multiple stream inputs and outputs. Even a constant number is considered a stream operator with no inputs and a single constant output stream. Nevertheless, when a named operator with a single output stream is used in a larger expression, it can usually be interpreted as if denoting its output stream without changing the meaning. This supports a more explicit point-free programming style. For example, in a

function definition like `f(x) = ...`, the argument `x` could represent an operator with multiple output streams. However, the definition `f(x) = x*2` in particular requires `x` to have a single output and it is a correct interpretation if we say that `x` denotes a stream which is pointwise multiplied with a constant stream of 2. In fact, this is just an alternative notation for `f(x) = (x,2):*`. In the latter, the expression `a,b` denotes a parallel composition - an operator whose list of outputs contains the outputs of the operator `a`, followed by the outputs of the operator `b`. The expression `a:b` denotes a sequential composition connecting the outputs of `a` to the inputs of `b`. It is sometimes useful to think of a parallel composition of stream operators as a list of their output streams. As the examples below will show, common functions on lists such as `map` and `fold` can be implemented for such parallel compositions in Faust, and provide useful abstractions. Other composition operators are split `<:`, merge `:>` and recursion `~` which facilitate one to many, many to one, and feedback connections of stream operator outputs and inputs, respectively.

StreamIt also has a single-dimensional notion of streams. Its computational model has been termed Phased Computation Graphs [77] and is a combination of the Computation Graphs [47] and Cyclo-static Dataflow [9] models - both extensions of the Synchronous Dataflow model. StreamIt supports implementation and composition of actors in an imperative style. The syntax of an actor definition is similar to that of a class in Java and C++. The core of an actor definition is a `work` function that defines the actor's firing. A limitation in StreamIt is that each actor has a single input stream and a single output stream. However, this limitation supports simple and intuitive actor composition using a few predefined structural patterns: pipeline, splitjoin and feedback. A pipeline composes a list of actors sequentially (feeding the output of one as the input to the next). A splitjoin splits or duplicates a stream across inputs of multiple actors and merges their outputs. A feedback loop merges the output of an actor back with its input.

5.1 Biquad Filter

As the first example, we look at a well-known DSP algorithm - the biquad filter in direct form I. The following is its usual mathematical definition [1], where x denotes the input stream, y the output stream, and a_1, a_2, b_0, b_1, b_2 are some pre-determined

coefficients:

$$\begin{aligned} y[n] = b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] \\ - a_1 y[n-1] - a_2 y[n-2] \end{aligned} \tag{5.1}$$

In the Faust implementation in Figure 5.1, recursive use of y in Eq. 5.1 is implemented using the recursive composition `A~B` which feeds the output of A into B, and the output of B back into A. An additional input of A and its output become the input and output of the expression. In our example, A is a partially applied abstraction X, and B is a partially applied abstraction Y, each corresponding to a part of Eq. 5.1. In the definition of X and Y, `s@n` denotes a delay of a stream s by way of prefixing n zeros. Note that the output of X which becomes the parameter `y` in the definition of Y is already delayed by 1 element by the recursive operator `~`.

```
biquad(a1,a2,b0,b1,b2) =
  +(X(b0,b1,b2)) ~ Y(a1,a2)
  with {
    X(b0,b1,b2,x) =   b0*x + b1*x@1 + b2*x@2 ;
    Y(a1,a2,y)    =        - a1*y   - a2*y@1 ;
  };
```

Figure 5.1: Biquad filter in Faust

In the StreamIt implementation in Fig. 5.2, the `peek` function is used to access past elements of the input stream. The convenience of `peek` is that these elements do not need to be explicitly stored as actor state, although this still introduces some syntactical overhead compared to Eq. 5.1. `push` adds an output element to the output stream, and `pop` removes an element from the input stream. Unfortunately, there is no expression like `peek` to access past output elements, so they must be stored explicitly as `y1` and `y2`. These variables are explicitly initialized to 0.

The Arrp implementation in Fig. 5.3 closely resembles Eq. 5.1 - it differs only in wrapping the equation into a function and the explicit definition of $y[0] = y[1] = 0$ so that the equation is valid for the entire domain of y starting with index 0.

[1] `https://ccrma.stanford.edu/~jos/filters/Direct_Form_I.html`

```
float->float filter Biquad(float a1, float a2, float b0, float b1, float b2)
{
    float y1 = 0;
    float y2 = 0;
    work peek 3 pop 1 push 1 {
        float x  = peek(0);
        float x1 = peek(1);
        float x2 = peek(2);
        float y = b0 * x  + b1 * x1 + b2 * x2
                          - a1 * y1 - a2 * y2;
        push(y);
        pop();
        y2 = y1; y1 = y;
    }
}
```

Figure 5.2: Biquad filter in StreamIt

```
biquad(a1,a2,b0,b1,b2,x) = y where
    y = [0 -> 0;
         1 -> 0;
         n -> b0 * x[n] + b1 * x[n-1] + b2 * x[n-2]
                        - a1 * y[n-1] - a2 * y[n-2]];
```

Figure 5.3: Biquad filter in Arrp

5.2 FIR Filter

Next, we look at another well-known DSP algorithm - the finite impulse response (FIR) filter. In the following usual mathematical definition [2], x represents the input stream, y the output stream, and each b_i a coefficient:

$$y[n] = \sum_{i=0}^{N-1} b_i x[n-i] \tag{5.2}$$

While recursive infinite impulse response (IIR) filters like the biquad in Eq. 5.1 can be implemented as a composition of two FIR filters, the implementation of the FIR filter is often distinct because it depends on a much larger amount of past input elements and uses a much larger number of coefficients. The coefficients are often supplied as an array, and the definition of FIR is polymorphic with respect to the size of this array.

[2] https://ccrma.stanford.edu/~jos/filters/FIR_Digital_Filters.html

In Faust, there is no notion of a finite array with an indexing operation returning a single value. The closest is a table (`waveform`) which can be read with a built-in operator `rdtable` producing a stream of values. However, as noted above, parallel compositions `(a,b,c,...)` can be considered as lists. It is a reasonable assumption in Faust that the list of coefficients can be defined as such a parallel composition. Based on this assumption, the `fir` function in Figure 5.4 is a typical recursive implementation where pattern matching is used to decompose the argument representing the coefficients into the first coefficient (`b`) and the rest of them (`bs`). The expression `x'` is equivalent to `x@1` shown in the previous example; it delays the stream `x` by an additional element at each recursion. Although atypical for Faust, it is possible to come closer to Eq. 5.2, as demonstrated by `fir2` in the same figure. The latter relies on the built-in function `sum` which sums multiple instances of `x@i` for all `i` from 0 to $N-1$. In addition, it uses `select` to access one of the coefficients `bs` by index, and `outputs` to count the number of coefficients `bs`. While `outputs` is a built-in, `select` can be implemented by recursive decomposition of `bs` similarly to `fir`.

```
fir((b,bs),x) = (b*x) + fir(bs, x');
fir(b,x) = (b*x);

fir2(bs,x) = sum(i, N, select(bs,i) * x@i)  with { N = outputs(bs); };
```

Figure 5.4: FIR filter in Faust

In the StreamIt implementation in Fig. 5.5, an array of coefficients can be conveniently passed as a parameter of the `Fir` actor. The summation in Eq. 5.2 is implemented iteratively using a `for` loop, as in other imperative languages.

StreamIt also supports a more fine-grained implementation similar to Faust's, shown in Figure 5.6. Here, the actor `Tap` implements a single 'tap' - a multiplication in Eq. 5.2. Multiple instances are composed using the `splitjoin` structure which duplicates the input stream to each tap and merges their output elements in a roundrobin fashion. Unfortunately, there does not exist a more convenient way to sum the outputs of taps but to implement another actor `Sum` which sums each consecutive group of N elements from the interleaved tap outputs. Consequently, there is no support for creating a generic abstraction of summation that could replace the summation in both the coarse-grained and the fine-grained implementation.

The theoretical benefit of the fine-grained StreamIt and Faust implementations is a lot of opportunity for parallelism in the dataflow model when using a large number N

```
float->float filter Fir(int N, float[N] b)
{
    work pop 1 peek N push 1 {

        float y = 0;
        for (int i = 0; i < N; ++i)
        {
            float x_i = peek(i);
            y += b[i] * x_i;
        }

        push(y);
        pop();
    }
}
```

Figure 5.5: FIR filter in StreamIt

of filter taps. However, the StreamIt and Faust compilers apparently assume a rather limited number of stream operators. In the process of compiling these examples, both compilers generate C++ code with a size complexity in $O(N)$. In other words, they repeat the similar code N times where a loop could be used instead. When N is large this can cause long compilation times and hurt performance. For example, a quick experiment with $N = 1000$ and using 4 threads on a 2-core Intel processor with hyperthreading shows that the fine-grained StreamIt implementation takes 6 times longer to compile (50 seconds), but executes at only 60% of the speed compared to the coarse-grained implementation.

In the Arrp implementation in Fig. 5.7, parts of the definition of `fir` are clearly related to Eq. 5.2. The definition uses `sum`, which is just a special case of the higher-order function `fold` - an implementation of reductions commonly found in functional programming. In the definition of `fir`, `#b` represents the size of the coefficient array `b`. The first part of the definition of `y` defines the first N output elements to be 0, similarly to the Arrp implementation of the biquad filter. The rest corresponds closely to Eq. 5.2, thanks to the inline definition of an array of size N which is an argument to `sum`.

5.3 Max Filter

In this section, we look at a simple max filter algorithm in Eq. 5.3: each element of the output stream y is the maximum of N elements (a *window*) of the input stream

```
float->float filter Sum(int N)
{
    work pop N push 1 {
        float y = 0;
        for (int i = 0; i < N; ++i)
            y += pop();
        push(y);
    }
}

float->float filter Tap(int d, float b)
{
    work pop 1 peek d+1 push 1 { push(b * peek(d)); pop(); }
}

float->float pipeline FIR(int N, float[N] b)
{
    add float->float splitjoin {
        split duplicate;
        for (int i = 0; i < N; ++i)
            add Tap(i, b[i]);
        join roundrobin;
    }
    add Sum(N);
}
```

Figure 5.6: Fine-grained FIR filter in StreamIt

x. The spacing between the windows (the *hop size*) is determined by the parameter H. In addition, we consider a variant of this algorithm in Eq. 5.4, where the same computation is applied independently to M streams, represented together as a 2-dimensional stream. Due to the similarity between the two equations, we would like to maximize code reuse between their implementations. Similar solutions could be used to implement a bank of M FIR filters processing a one-dimensional input and producing a 2-dimensional output stream.

$$y[n] = \max_{i=0}^{N-1} x[Hn + i] \tag{5.3}$$

$$y[n, j] = \max_{i=0}^{N-1} x[Hn + i, j], \quad 0 \leq j < M \tag{5.4}$$

The Faust implementation is shown in Fig. 5.8. Faust does not support multi-rate programs, so we can only implement the max filter with a hop size of 1 - hence the hop size parameter is absent. `max_filter` is a very concise implementation of Eq. 5.3 which relies on two generic abstractions: `par` and `fold`. The built-in `par` creates

```
fir(b,x) = y where {
    N = #b;
    y = [n | n < N -> 0
           | sum([N: i -> b[i] * x[n-i]]) ];
};

sum = fold(\a,b -> a+b);

fold(f,x) = r[#x-1] where
    r = [#x: 0 -> x[0];
             i -> f(r[i-1], x[i]) ];
```

Figure 5.7: FIR filter in Arrp

a parallel composition of instances of `x@i` (delayed versions of the stream `x`) for all `i` from 0 to $N-1$. The function `max_` reduces the elements of this parallel composition using the binary operator `max`, with the help of the higher-order function `fold`. If we consider a parallel composition as a list, `fold` has the same semantics as the usual `fold` on lists.

```
max_filter(N,x) = max_(par(i, N, x@i));

max_filter_md(N,x) = map(max_filter(N), x);

max_(x) = fold(max, x);

fold(f,(x,xs)) = f(x, fold(f,xs));
fold(f,x) = x;

map(f,(x,xs)) = f(x) , map(f,xs);
map(f,x) = f(x);
```

Figure 5.8: Max filter in Faust

Faust does not have multi-dimensional streams, so `max_filter_md` implements Eq. 5.4 as an operator with M input streams. Just like the coefficients in the FIR function in Fig. 5.4, `max_filter_md` assumes its inputs are defined as a parallel composition. With this assumption, it uses another well-known higher-order function `map`, which applies `max_filter` pointwise to the list of input streams. Fortunately, the recursive definition of `map` supports a composition `x` of any size, so M does not need to appear explicitly. It is worth nothing that `max_filter` and `max_filter_md` have different semantics, despite the same number of arguments; the former can not be applied to multiple streams, and the latter requires explicit handling of multiple

streams. Moreover, a parallel composition given as an argument to `max_filter_md` only models 2 dimensions (the composed streams and their elements) - there is no built-in facility in the language to model higher-dimensional structures.

The StreamIt implementation of Eq. 5.3 (`MaxFilter` in Fig. 5.9) looks quite similar to the FIR implementation (Fig. 5.5). However, it uses a parameter `H` for the `pop` rate representing the hop size. To implement Eq. 5.4 though, we need to explicitly handle the extra dimension. Since StreamIt does not support multi-dimensional streams, they are typically modeled as flattened into single-dimensional streams. In the case of Eq. 5.4, y is modeled as a sequence:

$$y[0,0], y[0,1], y[0,2], ..., y[1,0], y[1,1], y[1,2]...$$

The `splitjoin` structure already used in the fine-grained FIR filter is also useful here. It distributes the input stream elements among M instances of `MaxFilter` in a roundrobin fashion and recombines their output elements in the same way. The drawback of this is that the 2-dimensional structure is not encoded in the stream itself, and M must be an explicit parameter. This means that the parameter must be threaded through all the code using multi-dimensional streams. This is not only tedious but also error-prone, with errors only manifesting at runtime. `MaxFilter2d` also suffers from the same issue as the fine-grained FIR filter: it instantiates M actors, which causes the compiler to generate code with a size complexity $O(M)$. The compilation time for rather modest parameters $M = N = 100$ and $H = 1$ is 6 minutes an a half. A more fine-grained implementation like the fine-grained FIR filter would be dramatically worse: it would instantiate $M \times N$ actors.

An even more coarse-grained implementation is also possible in StreamIt, shown in Fig. 5.10. It requires a manual implementation of the `splitjoin` functionality which further obfuscates the code and reduces opportunities for reuse. We gain a faster compilation time of 11 seconds for $M = N = 100$ and $H = 1$. However, when increasing either M or N to 1000 the StreamIt compiler crashes while running out of memory, even though the program instantiates only a single actor. It seems that the large pop, peek, and push rates are to blame.

The Arrp implementation in Figure 5.11 is again very close to Eq. 5.3 using a similar syntax as the FIR filter example. The `max_` function is another application of `fold` (defined in Fig. 5.7). A unique property of the Arrp implementation is that the same function `max_filter` implements both Eq. 5.3 and Eq. 5.4. It applies without

```
float->float filter MaxFilter(int N, int H)
{
    work pop H peek N push 1 {

        float y = peek(0);
        for (int i = 1; i < N; ++i)
        {
            float x = peek(i);
            y = max(y,x);
        }
        push(y);
        for (int i = 0; i < H; ++i)
            pop();
    }
}

float->float splitjoin MaxFilter2d(int M, int N, int H)
{
    split roundrobin;
    for (int j = 0; j < M; ++j)
        add MaxFilter(N,H);
    join roundrobin;
}
```

Figure 5.9: Max filter in StreamIt

modification to a collection of streams modeled as a 2-dimensional stream or in fact any number of dimensions. Multiple streams do not need to be handled explicitly as in Faust and StreamIt - they are implicitly handled by the pointwise semantics of the primitive operations.

5.4 2d Wave Equation

We turn our attention to a streaming algorithm that is inherently multi-dimensional: the simulation of the vibration of a 2 dimensional surface. There are various numerical methods to simulate this phenomenon, one of which is a finite-difference time-domain (FDTD) scheme. This is used for example in the simulation of musical instruments [10] to simulate a vibrating membrane of a drum. In this application, one usually also simulates the complex interaction between the membrane and a source of excitation (e.g. a drum mallet). The sound produced by the vibration can be derived by sampling the displacement of the surface at one or more points. A real-time simulation can therefore be implemented as a streaming system with the source of excitation as

```
float->float filter MaxFilter2d(int M, int N, int H)
{
    work pop M*H peek M*N push M
    {
        for (int j = 0; j < M; ++j)
        {
            float y = 0;
            for (int i = 0; i < N; ++i)
            {
                float x = peek(i*M + j);
                y = max(y,x);
            }
            push(y);
        }
        for (int i = 0; i < M*H; ++i)
            pop();
    }
}
```

Figure 5.10: Coarse-grained version of 2d max filter in StreamIt

```
max_filter(N,H,x) = [n -> max_([N: i -> x[H*n + i]]) ];

max_ = fold(\a,b -> max(a,b));
```

Figure 5.11: Max filter in Arrp

an input and the surface displacement as output. Equation 5.5 describes a simplified form of such a system using a FDTD scheme. The surface is modeled as a regular 2-dimensional grid and a displacement value is associated with each point in the grid. The changing of the surface over time adds a third, infinite dimension. The surface is therefore modeled by a 3-dimensional stream u. The input stream x represents the excitation and is simply added to the displacement values in this simplified example. The output stream y samples a single point at the middle of the surface.

$$
\begin{aligned}
y[n] &= u[n, M/2, N/2] \\
u[n,i,j] &= b_0 u[n-2,i,j] \\
&\quad + b_1 u[n-1,i,j] \\
&\quad + b_2\big(u[n-1,i-1,j] + u[n-1,i,j-1] \\
&\quad + u[n-1,i+1,j] + u[n-1,i,j+1]\big), \\
&\quad + x[n], \\
&\quad\quad 0 < i < M-1, \quad 0 < j < N-1
\end{aligned}
\tag{5.5}
$$

This is an example which really tests the limits of expressivity in a language like Faust. Theoretically, it would be possible to model the multi-dimensional stream u in Eq. 5.5 in Faust as a bundle of $M \times N$ streams. An operator computing the recursively defined u could have its $M \times N$ outputs fed back as inputs. The main difficulty is in expressing the dependence of each stream on multiple neighbouring streams in 2 dimensions. There is no built-in composition operator implementing this dependence pattern. Attempting to implement it using the available facilities, one quickly finds the task quite impractical. A more feasible task and one more likely undertaken by a programmer is to implement this algorithm in a different language, and use it in Faust via the external function call facilities.

A StreamIt implementation of Eq. 5.5 is shown in Figure 5.12. As explained above, a multi-dimensional stream like u could be modeled as flattened into a one-dimensional stream, and its recursive definition could be implemented using a feedback connection. One issue is that actors in StreamIt have a single input stream, and we need one input for the excitation stream x. The latter could theoretically be interleaved with u in the input stream and de-interleaved inside the actor. However, a single actor firing would requires $M \times N$ elements from u and 1 element from x, and StreamIt does not provide an option for such an imbalanced interleaving. An alternative, used in Fig. 5.12, is to not model u as a stream but as part of the private state of the actor. Unfortunately, this requires a manual implementation of a circular buffer or similar for u, adding bloat and potential sources of errors. An alternative would be a more fine-grained implementation with one actor for each element of the grid. This would require $M \times N$ actors, which can be problematic for the StreamIt compiler when the grid is large, as already demonstrated with the previous examples.

The Arrp implementation in Figure 5.13 is rather straightforward, thanks to the support for multi-dimensional streams defined using recurrence equations.

```
float->float stateful filter
Wave2d(int M, int N, float b0, float b1, float b2)
{
    float[3][M][N] u;
    int n = 2;
    // ... initialization omitted ...
    work pop 1 push 1
    {
        int n1 = (n+3-1)%3;
        int n2 = (n+3-2)%3;
        float x = pop();
        for (int i = 1; i < M-1; ++i)
        {
            for (int j = 1; j < N-1; ++j)
            {
                u[n][i][j] =
                      b0 * u[n2][i][j]
                    + b1 * u[n1][i][j]
                    + b2 * ( u[n1][i+1][j] + u[n1][i][j+1]
                           + u[n1][i-1][j] + u[n1][i][j-1] )
                    + x
                    ;
            }
        }
        float y = u[n][M/2][N/2];
        push(y);
        n = (n+1)%3;
    }
}
```

Figure 5.12: 2d wave equation in StreamIt

5.5 Conclusions

This comparison of StreamIt, Faust, and Arrp provides useful insights into the potentials and limitations of different stream programming paradigms and models. StreamIt's imperative style with a code structure similar to languages like Java and C is generally much more verbose than the functional languages Arrp and Faust, as is commonly observed in comparisons of imperative and functional languages. The abstraction facilities in Arrp and Faust such as higher-order polymorphic functions have proven useful in stream programming. For example, the higher-order functions `map` and `fold` commonly used in functional languages also serve the streaming domain, facilitating code reuse.

Arrp's multi-dimensional stream model is unique among these languages. In combination with recurrence equations, it offers the most direct translation from math-

```
wave2d(M,N,b0,b1,b2,x) = y where
{
    y = [~: n -> u[n, M//2, N//2]];
    u = [~, M, N: n,i,j
        | n < 2 or i == 0 or j == 0 or i == M-1 or j == M-1 -> 0
        |   b0 * u[n-2, i, j]
          + b1 * u[n-1, i, j]
          + b2 * ( u[n-1, i+1, j] + u[n-1, i, j+1]
                 + u[n-1, i-1, j] + u[n-1, i, j-1] )
          + x[n];
    ];
};
```

Figure 5.13: 2d wave equation in Arrp

ematical equations to code. The max filter example also demonstrates a potential for code reuse only available in Arrp, due to its combination of multi-dimensional streams, functions polymorphic in array shape and pointwise semantics of primitive operations. While multi-dimensional streams can be modeled to some extent in StreamIt and Faust, the languages are not best suited for this. Finite and infinite dimensions have to be treated in different ways, adding syntactical overhead, limiting code reuse and increasing potential for errors.

There is also potential for improvement in the StreamIt and Faust compilers, particularly with the programs presented in this study. Multi-dimensional streaming problems quickly test the limits of these compilers as the stream volume increases. This manifests in long compilation times and high memory usage in the compiler as well as large generated code size - all of these quantities are proportional to the problem size. The poor structure of the generated code can also hurt performance.

To sum up, the main advantages of Arrp highlighted by this study are the strong resemblance to certain mathematical definitions and the support for high-volume multi-dimensional streams. There may well be other cases where the unique features of StreamIt and Faust prove an advantage. For example, StreamIt's disciplined actor composition using the built-in pipeline, splitjoin and feedback structures can make the structure of complex programs more obvious. In some cases, Faust's composition of stream operators with the help of recursive functions could be more intuitive than the usual mathematical definition of algorithms. A more exhaustive comparison of the programmability and performance of Arrp and other languages would certainly be valuable.

Nevertheless, these studies make it clear that Arrp occupies a distinct place in

the landscape of stream programming languages. Certain kinds of programs which are difficult, unintuitive or impossible to implement in other languages can be easily implemented in Arrp. Some programs which present significant obstacles to compilers of other languages are gracefully handled by the Arrp compiler. Some programs, which would typically be composed of parts implemented in different languages can be implemented entirely in Arrp, which enables more code reuse.

Additional examples of more complex programs implemented in Arrp are provided in Appendix B.

Chapter 6

Experimental Evaluation

Two sets of experiments are presented in this chapter:

- Section 6.1 describes the experiments and results previously published at the Workshop on Functional Art, Music, Modeling and Design [53]. These experiments focus on the overall usability of the language Arrp presented in chapter 3 and compilation techniques presented in chapter 4.
- Section 6.2 describes the experiments and results previously published in the ACM Transactions on Architecture and Code Optimization [56]. These experiments focus on the efficiency of the polyhedral compilation techniques presented in chapter 4 when combined with existing polyhedral optimizations for data locality and parallelization.

6.1 Preliminary Evaluation of Arrp for DSP Applications

This section presents a preliminary experimental evaluation of the language Arrp, presented in Chapter 3 and its compiler based on the techniques presented in Chapter 4. It is intended to confirm that the language allows implementation of realistic applications and to demonstrate the potential for efficient execution of the language. One of the main challenges is a fair selection of a baseline to compare to. Since the goal of Arrp is to simplify programming without sacrificing performance, we choose to compare it against hand-written C++. The latter is frequently used when high-performance is desired, but it is expected to be more verbose. Moreover, the

Arrp compiler generates C++ code, which simplifies comparison. The hypothesis is therefore that Arrp programs are more concise (avoiding syntactical redundancies in C++) while performing similarly or better than C++ programs. This evaluation is limited to single-threaded program execution while relying on automatic generation of vectorized code in the Intel C++ compiler.

6.1.1 Applications

We evaluate Arrp using the following applications:

- **Synth:** A 10-voice synthesizer, each voice consisting of a triangle wave generator filtered through two biquad filters in sequence.
- **EQ:** A 10-channel equalizer, each channel using a 64th order FIR filter.
- **AC:** Short-time autocorrelation. Every 512 input samples, a vector of 512 values is generated. Each n-th value in this vector is the result of correlating two 512 sample portions of the input at an offset of n samples.

In the EQ and AC applications, we use a computationally simple sawtooth wave signal as input.

These applications in the audio domain were selected because they all exhibit data dependencies over the infinite dimension of time, which is a unique characteristic of stream processing applications. All of these applications benefit from the multi-dimensional stream representation in Arrp which simplifies their implementation.

The hand-written C++ code uses a paradigm common in C++ libraries and C++ implementations of higher-level languages. Each self-sufficient and potentially reusable stream operator is implemented as a class: triangle wave generator, sawtooth wave generator, biquad filter, FIR filter, and autocorrelation processor. Moreover, a class is used as an abstraction of communication channels between actors.

In both languages, all floating point computation is done using the double-precision floating point data type. To improve performance, the size of each individually addressed buffer is rounded to the next power of two, so that we can use bit masking instead of modulo to wrap indices. All C++ code (auto-generated and hand-written) is pure C++ with no explicit vectorization or parallelization; we rely on the C++ compiler for automatic vectorization.

All evaluated code, including the C++ code automatically generated from Arrp, is available online [54]. The Arrp code is also available in the Appendix B.1.

6.1.2 System

The evaluation is performed on a system with an Intel Core i5 M560 CPU with a 2.67GHz clock, 2 cores, and 32KB L1 cache. The system runs the Linux kernel version 3.13.0. The evaluated code uses only a single core, and all data fits into the L1 cache. All the C++ code is compiled using the Intel C++ compiler version 16.0.2. The compiler's options "-O3 -fp-model precise" are used so that only value-preserving optimizations are performed. Inlining of all functions is forced using the option "-inline-forceinline".

6.1.3 Metrics

- **Cycles:** The number of CPU cycles spent in user-space while generating a number of output samples (Synth: 10000 samples, EQ: 1000 samples, AC: 20 sample). The measurement is performed using the PAPI library. [1]. The reported value is the median of measurements for 1000 repetitions.
- **Speed:** The reciprocal of "Cycles".
- **Buffers:** The total amount of memory required for buffering data between iterations of actors, obtained by manual code inspection. This does not include the memory allocated on stack at each program iteration.
- **Lines of Code:** The number of "essential" lines of code in Arrp and in hand-written C++; in both languages, we only count non-empty lines, and we skip lines that contain only delimiters such as parentheses and brackets.

6.1.4 Results

The summary of results is provided in table 6.1. The table shows the ratio of values for Arrp and values for hand-written C++.

Arrp enjoys a considerable speedup in the Synth and AC applications. Further information is revealed by detailed profiling of executable code. The code for Synth generated from Arrp has significantly simplified buffer accesses: much fewer cycles are spent in index wrapping. In the AC application, the C++ compiler has much more

[1] `http://icl.cs.utk.edu/papi`

Application	Cycles	Speed	Buffers	Lines of Code
Synth	0.37	2.70	1.58	0.24
EQ	1.09	0.92	1.03	0.14
AC	0.61	1.64	1.00	0.23
Mean	**0.69**	**1.75**	**1.20**	**0.20**

Table 6.1: Results of evaluation: Arrp / C++

opportunity for vectorization for code generated from Arrp. Both of these improvements are due to the detailed data dependency analysis and optimized scheduling in the polyhedral model.

On the other hand, Arrp requires significantly more buffers in the Synth example. This is also attributed to the way code is scheduled. This could be improved by adjusting the generic polyhedral scheduling algorithm that we currently use, to balance speed and memory requirements.

Finally, it is evident that an implementation of the same applications requires much less source code in Arrp.

6.2 Data Locality Optimizations and Parallelization

The polyhedral compilation techniques presented in Chapter 4 have been integrated into a compiler for the language Arrp presented in Chapter 3. The compilation framework 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.

The framework is evaluated using stream processing kernels exhibiting a variety of mathematical operations, data layouts and data dependence patterns. The input to the framework is an Arrp implementation of these algorithms. The output is C++ with statically allocated memory and OpenMP annotations for vectorization and

parallelization. The latter is further translated into executable code using a general-purpose C++ compiler. This is compared 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, no parallelism was achieved in StreamIt programs and only 4 out of 5 programs were successfully compiled. In many cases, the compiler did not terminate within 1 hour or it ran out of memory.

All the data generated in these experiments as well as instructions and source code required to replicate the experiments are available online [55]. The Arrp code is also available in the Appendix B.2.

6.2.1 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, definitions of outputs at domain bounds are omitted.

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 $0 \leq l < W$ 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.[2] $b_0, b_1, ...$ denote predefined coefficients.	$u[n,i] = b_0 u[n-2,i] + b_1 u[n-1,i]$ $+ b_2(u[n-1,i-1] + u[n-1,i+1])$ $+ x[n], \quad 0 \leq i < N^2$ $y[n] = u[n, N/2]$
wave2d: FDTD scheme for 2-dimensional wave equation.[2] $b_0, b_1, ...$ denote predefined coefficients.	$u[n,i,j] = b_0 u[n-2,i,j] + b_1 u[n-1,i,j]$ $+ b_2\big(u[n-1,i-1,j] + u[n-1,i+1,j]$ $+ u[n-1,i,j-1] + u[n-1,i,j+1]\big)$ $+ x[n], \quad 0 \leq i < N,\ 0 \leq j < N$ $y[n] = u[n, N/2, N/2]$

6.2.2 Algorithm Implementation and Evaluation

Source Code

The Arrp, StreamIt and hand-written C++ source code is available online [55]. 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 4.5.2: 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 etc. 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 into the

[2] Used for example in physical modeling of musical instruments [10].

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

Table 6.3: Scheduling parameters for evaluated Arrp programs

outside world, as described in Section 4.3. In C++ code, these statements appear as calls to `input` and `output` functions which transfer an element or array of elements (see Figure 4.1 as an example). Hand-written C++ and C++ generated by the StreamIt compiler follow the same pattern.

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. A large number of manually selected tile sizes were evaluated, in the range of 4 to 4096 units in each schedule dimension as well as leaving some dimensions untiled (tile size equals ∞). Results are reported for the tilings yielding the highest throughput. The autocorrelation algorithm is a special case where a tile size is chosen based on the parameter N so as to align the tile boundaries with the input window boundaries. Table 6.3 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 4.5.2). For most algorithms in this evaluation, the chosen direction for periodic tiling is simply the standard basis vector for the first dimension of the schedule and the smallest periodic tiling size and offset for this direction are used. In wave1d and wave2d though, data dependences preclude parallel execution of sub-tiles using the default inter-tile schedule. This is alleviated with the approach depicted in Figure 4.6c and explained in Section 4.4.3.

In hand-optimized C++, tiling and other schedule modifications were explored and yielded only limited benefits, due to the limitation of transformations to a single iteration of the period. See Section 6.2.3 for details. For auto-optimized C++,

the highest degree of schedule transformations is enabled using C++ compiler options. See Section 6.2.2 for details. When compiling StreamIt, no options related to scheduling were found to improve performance.

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 dependences 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, the loop for to the third scheduling direction l is parallelized instead - it also carries no dependences but has a much larger number of iterations.

In hand-optimized C++ code, OpenMP pragmas are inserted for explicit parallelization and vectorization. For auto-optimized C++, automatic parallelization and vectorization is enabled using C++ compiler options. See Section 6.2.2 for details. The StreamIt compiler was unable to compile the programs with parallelization enabled.

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 4.5.3:

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 4.5.3). The auto-optimized C++ code is hand-written for each buffer type separately. In hand-optimized C++ code, different buffer implementations are manually explored and the one is chosen that yields the highest throughput in each case.

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, these options yield a fair comparison between the two compilers in the evaluated programs. For auto-optimized C++, the highest degree of automatic loop transformations and parallelization is enabled using additional options: `-parallel` for the Intel compiler; `-floop-nest-optimize`, `-floop-parallelize-all`, and `-ftree-parallelize-loops=n` for the GNU compiler (with `n` the parallelization factor).

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, some CPU features are disabled, namely frequency scaling (P states), idle states (C states) and hyperthreading. The Turbo feature is enabled, because it was found to increase performance consistently.

Measured Quantities

The following quantities are measured:

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.
Elapsed time is measured using the standard C++ facility `std::chrono::steady_clock`.

For each algorithm, a specific number of periods is executed such that their total duration adds up to about 1 second, 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. In this evaluation, it is defined as follows. Consider first only one-dimensional input and output streams. Assign indices $0, 1, 2, ...$ 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. This definition is applied to multi-dimensional streams by modeling them as one-dimensional streams. 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.

6.2.3 Results

Figure 6.1 shows how throughput scales with the number of threads. We use the scale parameter $N = 2000$ for all algorithms. Arrp code uses tile sizes in Table 6.3. Explicit hoisting and vectorization is enabled for all Arrp sources except for ac. For Arrp sources and auto-optimized C++ sources, the buffer type 'mask' is used 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. Unfortunately, StreamIt results for some configurations are missing, because the StreamIt compiler was unable to compile the max-filter program and was only able to generate single-threaded programs.

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 6.5.

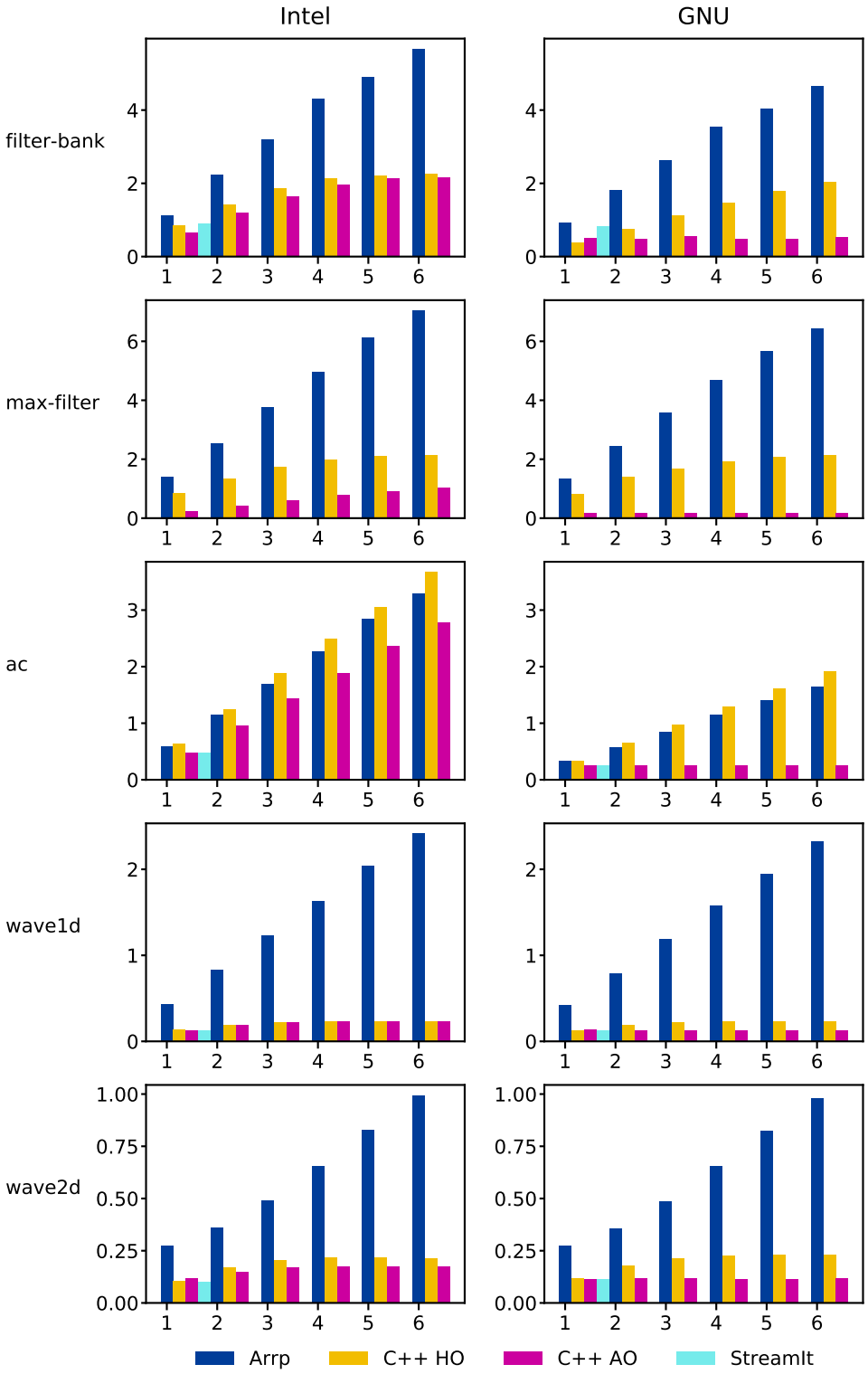


Figure 6.1: Throughput (vertical, in output elements/μs for filter-bank, max-filter, ac, and output elements/ms for wave-1d and wave-2d) in relation to number of threads (horizontal) using Intel C++ compiler (left) and GNU (right).

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.

Table 6.4 shows how buffer optimizations (described in Section 4.5.3) and their interaction with C++ compiler optimizations affect throughput of Arrp and auto-optimized C++ code. These results use a common algorithm scale $N = 2000$, Arrp tile sizes in Table 6.3 and 6 threads. The table contains 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). These parameters are presented together due to 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 6.5 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

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

Table 6.4: Effect of buffer type, hoisting (H) and explicit vectorization (V) on throughput for Arrp and auto-optimized C++ implementations. Units are 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.

results in an extreme increase of storage size. This is also the reason for omission of the related results from Table 6.4 - 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 6.6 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 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.

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	**101562.6**
wave2d	91.6	122.2	183.3	91.6	**61.1**	61.1	**94821.3**

Table 6.5: Storage size in Mb for different implementations and buffer types. Using algorithm scale $N = 2000$, Arrp tile sizes in Table 6.3 and supporting 6 threads. The lowest and highest value in each row are emphasized.

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	$O(PT) \cap O(N^2)$	1537
wave2d	$O(1)$	0	$O(PT) \cap O(N)$	550

Table 6.6: Logical latency (complexity and value). N is algorithm scale, T is tile size in first dimension, P is degree of thread parallelism. Values reported for $N = 2000$, Arrp tile sizes in Table 6.3, and 6 threads.

6.3 Conclusions

In the experiments presented in Section 6.1, Arrp code and the C++ code generated by the Arrp compiler was compared to conventionally hand-written C++ code. While Arrp reduces the amount of source code by a factor of 5x in average, it enjoys an average speedup by a factor of 1.75x. On the other hand, it uses in average 1.2x the amount of memory.

The experiments presented in Section 6.2 evaluated the role of the compilation techniques presented in this dissertation in maximizing the performance of stream processing kernels on very large problem sizes. As the experiments confirm, data locality optimizations are crucial in maximizing performance of such programs and successfully utilizing the available parallelism. Much prior work has been done on automating these optimizations in the polyhedral model. However, the novel techniques presented in this dissertation play an important role in applying the polyhedral optimizations to polyhedral models with unbounded domains that describe streaming

programs. Even more, the experimental results suggest that the polyhedral optimizations can be even more effective when applied to complete, unbounded models of streaming programs, compared to optimizing a single period of a C++ implementation. In particular, on a 6-core Intel Xeon CPU, we see speedups up to 10x (geometric mean 3.3x) over hand-optimized C++, and up to 10x (geometric mean 4.2x) over hand-written C++ optimized by the Intel C++ compiler. Therefore, the techniques presented in this dissertation could prove useful in optimizing any stream programming language that can be translated into the unbounded polyhedral model - of which Arrp is just an example.

The central part in the compilation method evaluated here is a novel polyhedral schedule transformation called periodic tiling. It is worth noting that the benefits reported above are achieved using simple heuristics to select periodic tiling parameters, also involving some manual intervention. There is room for improvement using an automated optimization of these parameters, which remains an interesting challenge for the future.

Section 6.2 also describes experiments evaluating various well-known buffer implementation techniques in the context of polyhedral code generation. Each implementation type represents a trade-off between storage size and computational complexity. The best throughput results in the experimental evaluation are achieved at a cost of 2.5x 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.

Section 6.2 includes a preliminary evaluation of the effects of the proposed compilation method on input-output latency. It was found that tiled execution and parallelization may add the tile size and degree of parallelism as additional factors to latency in some cases. Still, the best throughput was observed while increasing latency only by a fraction of the problem size. Further work is required to determine real-time latency effects.

Chapter 7

Conclusions

7.1 Summary of the Dissertation

In this dissertation, I have presented a new programming language for stream processing named Arrp, as well as novel compilation techniques in the polyhedral framework which facilitate compilation and optimization of this language and potentially others.

Like many other languages and programming systems for stream processing, Arrp exploits the particular nature of streaming programs to improve the productivity of programmers. The design of Arrp is motivated by the particular problems encountered in the implementation of programs processing high-volume multi-dimensional streams. Specifically, it is motivated by the observation that various considerations often force programmers to take a coarse-grained stream programming approach where large amounts of program behavior are implemented in a general-purpose language like C++ while only a coarse program structure is defined in a language specifically designed for streaming applications. Consequently, the benefits of the domain-specific language with regard to programmer productivity are minimized. The major reasons for resorting to a coarse-grained approach are the lack of expressivity of the domain-specific language and performance concerns. These issues are especially exacerbated in multi-dimensional stream processing applications with high performance demands.

In addressing the issue of expressivity, the design of Arrp is guided by another significant observation. Namely, many multi-dimensional stream processing algorithms are most naturally expressed in the form of recurrence equations commonly used in mathematical definitions of streaming algorithms, and no other existing style of stream programming supports nearly as intuitive and convenient expression. There-

fore, Arrp's design is centered around the expression of multi-dimensional streams using recurrence equations. In addition, Arrp provides powerful abstractions like higher-order functions, polymorphic in the shape and size of multi-dimensional streams, as well as the convenient pointwise semantics of primitive operators. Altogether, these properties provide new forms of expressivity and potential for modularity and code reuse not offered by existing languages. This has been demonstrated through a series of case studies comparing Arrp with two other representative languages for stream processing, as well as by implementing larger applications as part of the experimental evaluation of Arrp.

The challenge in addressing the issue of performance is devising a compilation method for Arrp that is competitive with hand-written code in a language like C++. This is crucial for the adoption of new languages, since programmers tend to quickly abandon a more convenient language for a faster one. In this dissertation, I focus specifically on efficient execution on general-purpose multi-core machines - they are widely available and the number of cores per processor is steadily rising, providing ample parallelism from which streaming applications can greatly benefit. To address this challenge, I have turned to the polyhedral framework for two reasons: it has previously been used in the compilation of recurrence equations, and it also represents the state-of-the-art in automated optimizations for the aforementioned hardware. However, significant obstacles were identified in generating executable code from an unbounded polyhedral representing a streaming program. Novel polyhedral techniques presented in this dissertation overcome these obstacles and accommodate the state-of-the-art polyhedral optimizations, which has been confirmed by empirical evaluation. Moreover, the evaluation also shows that these optimizations can be even more effective when applied on complete, unbounded models of streaming programs. Since the latter is enabled by the work presented in this dissertation, this work may have a broader impact than just the compilation of Arrp. The novel polyhedral techniques and related proofs are stated in a general way, with minimal assumptions about the underlying polyhedral model, so they can be applied to polyhedral models derived from a wide variety of programming languages.

7.2 Future Work

While Arrp improves expressivity for a broad range of streaming programs, it has certain limitations. Most notably, it has a rather limited support for data-dependent

behaviors and no support for time-dependent behaviors. These limitations are in fact what makes it amenable to detailed static analysis and aggressive optimizations in the polyhedral model. In this regard, it is essentially as expressive as the Synchronous Dataflow (SDF) model. The latter has been applied extensively in the stream processing domain, which implies equal applicability of Arrp. Nevertheless, many real-world problems require dynamic behaviors, and it is a worthwhile challenge to integrate Arrp's language features and compilation techniques into a system which supports such behaviors. This is already possible in a way since the code generated by the Arrp compiler in fact constitutes a SDF actor, and could be integrated into a larger dataflow graph with a dynamic dataflow model. This is an immediate opportunity which deserves more extensive exploration. However, a more tight integration of dynamic behaviors could further improve programmability and performance. Extending the proposed polyhedral compilation techniques to parametric polyhedral models is a crucial step in enabling dynamic stream sizes and dynamically bounded recurrence equations.

The polyhedral compilation techniques presented in this dissertation accommodate state-of-the-art polyhedral optimizations which improve data locality and expose parallelism. The experimental evaluation has shown that they can be exploited even with simple heuristics and manual tweaking of certain parameters. A worthwhile pursuit is to automatically select optimal parameters - in particular the tile sizes when tiling for data locality, and the periodic tiling direction and size. Automatic tile size selection is a challenging problem with an active search for solutions [73]. Tiling is not a linear transformation and so linear optimization techniques do not apply directly. Moreover, good choices often depend on many aspects of the target hardware architecture. Fortunately, the compilation method presented here is independent of the tile size selection algorithm and can easily accommodate various existing and future solutions. The choice of a periodic tiling direction and size has a significant impact specifically in streaming programs, because it can affect the input-output latency. There is often a trade-off between throughput and latency, so a future work on optimal periodic tiling parameter selection will have to carefully explore their relation.

This dissertation has focused on compilation for general-purpose multi-core processors. Based on current trends in hardware development, this work will remain relevant for a good amount of time. The number of processors and processor cores in systems keeps rising. The language Arrp and the polyhedral compilation method

for stream processing are in a good position to utilize the increasing amount parallelism in such systems. At the same time, power consumption is an increasingly important factor. While power consumption has not been measured in this work, we can expect that the polyhedral data locality optimizations enabled by the proposed compilation method reduce power consumption in addition to increasing throughput, because they improve cache utilization. However, heterogeneous architectures, reconfigurable hardware and application specific hardware synthesis are all receiving a lot of attention in the struggle to increase the computing performance per watt of power. Compilation for these types of hardware requires specific methods which have not been explored in the compilation of Arrp yet. However, these methods usually favor functional languages from which a detailed dataflow graph (or polyhedral model) can be derived. Examples range from the first work on the derivation of systolic arrays from recurrence equations [52] and the early dataflow languages [43] to Single-assignment C (SaC) used today [29]. Because Arrp has similar properties and can be translated into the polyhedral model using the method presented in this dissertation, its execution on these types of hardware may not be far away.

An interesting extension of the optimization techniques explored in this dissertation would be optimization under specific constraints on throughput, latency, memory usage, power consumption, etc. Such constraints often arise in real-time applications and embedded systems with restricted resources. In the presence of such constraints, multi-objective optimization may be guided to balance between objectives so as to satisfy all the constraints. Another relevant approach is approximate computing whereby an algorithm is adapted to compute an approximate result to the given problem, instead of a more accurate one, in order to satisfy performance constraints.

Appendix A

Publications

Parts of the work presented in this dissertation have been previosuly published in the following:

- Jakob Leben. Arrp: A functional language with multi-dimensional signals and recurrence equations. In Proceedings of the 4th International Workshop on Func- tional Art, Music, Modelling, and Design, FARM 2016, pages 17–28, New York, NY, USA, 2016. ACM. doi: 10.1145/2975980.2975983.

- Jakob Leben and George Tzanetakis. Polyhedral compilation for multi-dimensional stream processing. ACM Transactions on Architecture and Code Optimization, 2019. doi: 10.1145/3330999. IN PRESS.

Prior to the work presented in this dissertation I have developed a language for specification of stream processing graphs using primitive stream operators provided by the Marsyas C++ library for audio analysis and synthesis. This resulted in the following publication:

- Jakob Leben and George Tzanetakis. Declarative Composition and Reactive Control in Marsyas. In Proceedings of the International Computer Music Conference, 2014.

Earlier, I have contributed to the development of the SuperCollider language for sound synthesis and computer music composition. This language is used extensively by computer music composers, performers and at educational institutions. Related to this work is the following publication:

- Jakob Leben and Tim Blechmann. SuperCollider IDE: A Dedicated Integrated Development Environment for SuperCollider. In Proceedings of the Linux Audio Conference, 2013.

Appendix B

Code Examples

B.1 Experiments 1

This Arrp code was used in the experiments described in Section 6.1. One of the goals was to evaluate the overall usability of Arrp, including all its features supporting code modularity and reuse. Hence, this code is written in a style intended to maximize the utility of these features in order to increase programmer productivity. Several modules with reusable functions are defined first, followed by the programs evaluated in the experiments. Almost all of the functions in the modules are ultimately used in the programs, some multiple times.

B.1.1 Array function module

```
module array;

slice(pos,len,a) = [len:i -> a[pos + i]];

scan(f,a) =
  [#a: 0 -> a[0];
       i -> f(a[i], this[i-1]); ];

fold(f,a) = s[#a-1] where s = scan(f,a);

iterate(f,init) =
  [~:
    0 -> init;
    i -> f(this[i-1]);
```

```
  ];

map(f,a) = [#a: i -> f(a[i])]
```

B.1.2 General math module

```
module math;
import array;

pi = 3.14159265359;

add(x,y) = x + y;

sum(a) = array.fold(add,a);
```

B.1.3 DSP module

```
module signal;
import array;
import math;

phase(freq, init) =
  [~: 0 -> init;
      t ->
        if this[t-1] + freq[t] >= 1
        then this[t-1] + freq[t] - 1
        else this[t-1] + freq[t]
  ];

sine(freq, init_ph) =
  let ph = phase(freq, init_ph) in
    sin(ph * 2 * math.pi);

triangle(freq, delay) =
  let ph = phase(freq, delay) in
    1 - abs(ph*2 - 1);

biquad(a,b,x) =
  let y = [~: n
      | n < 2 -> 0
      |   x[n]      * b[0] + x[n-1]    * b[1] + x[n-2] * b[2]
        - this[n-1] * a[0] - this[n-2] * a[1]]
  in [~: n -> y[n+2]];
```

B.1.4 Synth Program

```
module synth;

import signal;
import math;

a = [0.2; 0.1];
b = [0.3; 0.2; 0.1];

n_voices = 10;

voices = [n_voices: i ->
  signal.biquad(a,b,
    signal.biquad(a,b,
      signal.triangle(1.0/(i+5), 0)))
  * (i+1)
];

main = math.sum(voices);
```

B.1.5 EQ Program

```
module eq;

import math;

convolve(h,x) =
  [#x: i -> math.sum([#h: k -> h[k] * x[i+#h-1-k]])];

eq(hs,x) =
  let chs = [#hs: i -> convolve(hs[i],x)]
    in math.sum(chs);

src = [~: 0 -> 0; i -> (this[i-1] + 1) % 5];

hs = [10: i -> [64: n -> 1.0/(n+1)]];

main = eq(hs,src);
```

B.1.6 AC Program

```
module ac;
```

```
import array;
import math;
import signal;

corr(a,b) = math.sum(a*b);

acorr(t,w,d,a) =
  corr(array.slice(t,w,a), array.slice(t+d,w,a));

acorr_run(hop,wnd,dmax,a) =
  [~,dmax: t,d -> acorr(t*hop,wnd,d,a)];

src = [~: 0 -> 0; i -> (this[i-1] + 1) % 100];

win = 512;

main = acorr_run(win,win,win,real64(src));
```

B.2 Experiments 2

This code was used in the experiments described in Section 6.2. The purpose of this code was to evaluate the compilation and optimization of streaming programs in the form of recurrence equations. For this reason, this code is written so as to minimize the overhead of reduction to recurrence equations and to enable a direct control over the resulting system of equations. Consequently, no functions or other abstraction mechanisms are used.

B.2.1 filter-bank

```
W = @PROBLEM_SIZE@;
F = @PROBLEM_SIZE@;

input main_in :: [~]real64;
input coefs :: [F, W]real64;

f :: [~,F,W+1]real64;
f = [~,F,W+1:
    t,i,0 -> 0;
    t,i,k -> f[t,i,k-1] + main_in[t+k-1] * coefs[i,k-1];
```

```
];

main = [~,F: t,i -> f[t,i,W]];
```

B.2.2 max-filter

```
N=@PROBLEM_SIZE@;
W=@PROBLEM_SIZE@;

input main_in :: [~,N]real64;

f :: [~,N,W+1]real64;
f =
[~,N,W+1:
    t,i,0 -> 0;
    t,i,j -> max(f[t, i, j-1], main_in[t+j-1, i]);
];

main = [~,N: t,i -> f[t,i,W]];
```

B.2.3 autocorrelation

```
win_size = @PROBLEM_SIZE@*5;
hop_size = win_size//4;

input main_in :: [~]real64;

ac :: [~,win_size,win_size+1]real64;
ac =
[~,win_size,win_size+1:
    t,d,0 -> 0;
    t,d,i -> ac[t,d,i-1] +
        main_in[t*hop_size + i-1] *
        main_in[t*hop_size + i-1 + d]
];

main = [~,win_size: t,d -> ac[t,d,win_size]];
```

B.2.4 wave1d

```
N = @PROBLEM_SIZE@ * @PROBLEM_SIZE@;
s0 = 0.1;
s1 = 0.1;
```

```
input main_in :: [~]real64;

u :: [~,N+1]real64;
u =
[~,N+1:
    t,n
    | t < 2 and n == 0 -> 0
    | t < 2 and n == N -> 0
    | t < 2  -> n
    | n == 0 or n == N -> 0
    |
              -u[t-2,n]
        + s0 * u[t-1,n]
        + s1 * (u[t-1,n-1] + u[t-1,n+1])
        + main_in[t-2]
];

main = [~: t -> u[t, N//2]];
```

B.2.5 wave2d

```
Nx = @PROBLEM_SIZE@;
Ny = @PROBLEM_SIZE@;

s0 = 0.010203;
s1 = 0.49743;
t0 = -0.99993;

input main_in :: [~]real64;

u :: [~, Nx+1, Ny+1]real64;
u =
[~, Nx+1, Ny+1:

    t,x,y
    | x == 0 or y == 0 or x == Nx or y == Nx ->
        0
    | t < 2 ->
        x + y
    |
        s1 * (
```

```
              u[t-1, x+1, y]
            + u[t-1, x-1, y]
            + u[t-1, x, y+1]
            + u[t-1, x, y-1]
        )
        + s0 * u[t-1, x, y]
        + t0 * u[t-2, x, y]
        + main_in[t-2]
    ;
];

main = [~: t -> u[t, Nx//2, Ny//2]];
```

B.3 Other Examples

The examples in this section are more complex programs demonstrating versatility of Arrp.

B.3.1 IIR filter

The function `iir` is an implementation of a generic IIR filter of any order determined by the number of given coefficients `as` and `bs`.

```
map(f,a) = [i -> f(a[i])];

scan(f,a) = [
  0 -> a[0];
  i -> f(a[i], this[i-1])
];

fold(f,a) = scan(f,a)[#a-1];

sum = fold(\a,b -> a+b);

reverse(x) = [i -> x[#x-1-i]];

delay(v,d,x) = [i | i < d -> v | x[i-d]];

windows(size,hop,x) = [i -> [size:j -> x[hop*i + j]]];

convolve(h,x) =
  map(\w -> sum(h * reverse(w)), windows(#h,1,x));
```

```
iir(as,bs,x) =
      y = convolve(bs,delay(0,#bs-1,x)) -
          convolve(as,delay(0,#as,y));
```

B.3.2 Fractional Delay

The function `fdelay3` in this example is an implementation of a variable fractional delay with 3rd order Lagrange interpolation[1]. This implementation is closely based on the implementation of `fdelay3` in the standard Faust library named `delays`. Such delays are typically used in larger physical modeling systems. In this example, 8 channels of input audio named `main_in` are processed through the delay, and the delay amount is controlled by a real-valued input stream `control_one_in`. This control signal is modeled with a lower sampling rate than the audio (1/512 times the audio rate), as usual in real-time controlled audio systems. However, the control signal is upsampled and smoothed in the program, which is also common in such systems.

```
CONTROL_RATE = 512;

input main_in :: [~,8]real64;
input control_one_in :: [~]real64;

frac(x) = x - floor(x);

## Variable delay, with an integer delay d clamped to range [0,D]
vdelay(D,d,x) = [~: n -> x[n + max(0, min(D, D-d[n]))]];

## Fractional delay with 3rd order Lagrange Interpolation
fdelay3(D, d, x) =
    vdelay(D, di+0, x) * h_3_0(df) +
    vdelay(D, di+1, x) * h_3_1(df) +
    vdelay(D, di+2, x) * h_3_2(df) +
    vdelay(D, di+3, x) * h_3_3(df)
where
{
    o = (3-1.00001)/2;
    dmo = d - o;
```

[1] https://ccrma.stanford.edu/~jos/Interpolation/Time_Varying_Lagrange_Interpolation.html

```
    di = int(dmo);
    df = o + frac(dmo);
    h_3_0(d) = (d - 1) * (d - 2) * (d - 3) * (-1/6);
    h_3_1(d) = d * (d - 2) * (d - 3) * (1/2);
    h_3_2(d) = d * (d - 1) * (d - 3) * (-1/2);
    h_3_3(d) = d * (d - 1) * (d - 2) * (1/6);
};

delay(d,x) = [~: n | n < d -> 0 | x[n-d]];

smooth(a, x) = y where {
    y :: [~]real64;
    y = (1-a) * x + a * delay(1,y);
};

upsample(r, x) = [~: n -> x[n//r]];

smooth_control(a) = smooth(a) . upsample(CONTROL_RATE);
control = upsample(CONTROL_RATE);

delay_in = smooth_control(0.995, control_one_in);

main = fdelay3(128, delay_in, main_in);
```

B.3.3 Freeverb

This is an implementation of the Freeverb algorithm[2], a type of Schroeder reverberator frequently used in audio processing software.

```
## NOTE: Assuming main_in is audio sampled at 44100 Hz
scaleroom = 0.28;
offsetroom = 0.7;
COMB_FEEDBACK = real32(0.5 * scaleroom + offsetroom);

scaledamp = 0.4;
COMB_DAMPING = real32(0.5 * scaledamp);

ALLPASS_FEEDBACK = real32(0.5);

input main_in :: [~]real32;
```

[2] https://ccrma.stanford.edu/~jos/pasp/Freeverb.html

```
delay(N,x) = [~: n
    | n < N -> real32(0)
    | x[n-N]
];

lbcf(D,x) = y where {
    y :: [~]real32;
    y = delay(D, w * COMB_FEEDBACK + x);
    w :: [~]real32;
    w = delay(1,y) * (real32(1) - COMB_DAMPING) + delay(1,w) * COMB_DAMPING;
};

allpass_comb(D, x) = y where {
    y = -ALLPASS_FEEDBACK * w1 + w2;
    w2 :: [~]real32;
    w1 :: [~]real32;
    w2 = delay(D, w1);
    w1 = w2 * ALLPASS_FEEDBACK + x;
};

mono_freeverb(x) = y0 where {
    y0 = allpass_comb(225, y1);
    y1 = allpass_comb(341, y2);
    y2 = allpass_comb(441, y3);
    y3 = allpass_comb(556, y4);
    y4 =
        lbcf(1116,x) +
        lbcf(1188,x) +
        lbcf(1277,x) +
        lbcf(1356,x) +
        lbcf(1422,x) +
        lbcf(1491,x) +
        lbcf(1557,x) +
        lbcf(1617,x);
};

main = mono_freeverb(main_in);
```

Bibliography

[1] Aravind Acharya and Uday Bondhugula. 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 2015, pages 54–64, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. doi:10.1145/2688500.2688512.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[3] Markus Aronsson, Emil Axelsson, and Mary Sheeran. Stream processing for embedded domain specific languages. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*, pages 8:1–8:12, New York, NY, USA, 2014. ACM. doi:10.1145/2746325.2746334.

[4] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. doi:10.1109/SC.2012.107.

[5] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society. doi:10.1109/PACT.2004.11.

[6] Somashekaracharya G. Bhaskaracharya and Uday Bondhugula. PolyGLoT: A polyhedral loop transformation framework for a graphical dataflow language. In *Proceedings of the 22nd International Conference on Compiler Construction*,

CC'13, pages 123–143, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37050-2. doi:10.1007/978-3-642-37051-9_7.

[7] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Automatic storage optimization for arrays. *ACM Transactions on Programming Languages and Systems*, 38(3):1–23, 2016. ISSN 01640925. doi:10.1145/2845078.

[8] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Optimized software synthesis for synchronous dataflow. In *Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 250–262, July 1997. doi:10.1109/ASAP.1997.606831.

[9] Shuvra S. Bhattacharyya. *Handbook of signal processing systems*. Springer Science & Business, 2013. doi:10.1007/978-1-4419-6345-1.

[10] Stefan Bilbao. *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*. John Wiley & Sons, 2009. ISBN 9780470749012. doi:10.1002/9780470749012.

[11] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, Feb 1996. ISSN 1053-587X. doi:10.1109/78.485935.

[12] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-78791-4_9.

[13] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi:10.1145/1375581.1375595.

[14] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silvén. Actor merging for dataflow process networks. *IEEE Transactions on Signal Processing*, 63(10):2496–2508, May 2015. ISSN 1053-587X. doi:10.1109/TSP.2015.2411229.

[15] Stuart Bray and George Tzanetakis. Implicit patching for dataflow-based audio analysis and synthesis. In *Proceedings of the International Computer Music Conference (ICMC)*, pages 13–16, 2005.

[16] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188. ACM, 1987. doi:10.1145/41625.41641.

[17] Francois Charot, Madeleine Nyamsi, Patrice Quinton, and Charles Wagner. 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, pages 6–16, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2226-2. doi:10.1109/ASAP.2004.1342454.

[18] Alain Darte, Rob Schreiber, and Gilles Villard. Lattice-based memory allocation. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 298–308, New York, NY, USA, 2003. ACM. ISBN 1-58113-676-5. doi:10.1145/951710.951749.

[19] F. de Dinechin, P. Quinton, and T. Risset. Structuration of the ALPHA language. In *Programming Models for Massively Parallel Computers*, pages 18–24, Oct 1995. doi:10.1109/PMMPC.1995.504337.

[20] Lukasz Domagala, Duco van Amstel, and Fabrice Rastello. Generalized cache tiling for dataflow programs. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016, pages 52–61, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4316-9. doi:10.1145/2907950.2907960.

[21] Hritam Dutta, Frank Hannig, and Jurgen Teich. Hierarchical partitioning for piecewise linear algorithms. In *Proceedings of the International Symposium on Parallel Computing in Electrical Engineering*, PARELEC '06, pages 153–160,

Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2554-7. doi:10.1109/PARELEC.2006.43.

[22] Johan Eker and Jörn W. Janneck. CAL language report: Specification of the CAL actor language. Technical report, University of California at Berkeley, 2003.

[23] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, feb 1991. ISSN 1573-7640. doi:10.1007/BF01407931.

[24] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21 (6):389–420, dec 1992. ISSN 1573-7640. doi:10.1007/BF01379404.

[25] Paul Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, oct 1992. ISSN 1573-7640. doi:10.1007/BF01407835.

[26] Paul Feautrier and Christian Lengauer. Polyhedron model. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi:10.1007/978-0-387-09766-4_502.

[27] Calin Glitia, Philippe Dumont, and Pierre Boulet. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 21(2):105–131, June 2010. ISSN 0923-6082. doi:10.1007/s11045-009-0085-4.

[28] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 151–162, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. doi:10.1145/1168857.1168877.

[29] Clemens Grelck. Single Assignment C (SAC) high productivity meets high performance. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 207–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-32096-5. doi:10.1007/978-3-642-32096-5_5.

[30] Martin Griebl. On tiling space-time mapped loop nests. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 322–323, New York, NY, USA, 2001. ACM. ISBN 1-58113-409-6. doi:10.1145/378580.378740.

[31] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012. doi:10.1142/S0129626412500107.

[32] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi:10.1145/2581122.2544160.

[33] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral AST generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4): 12:1–12:50, July 2015. ISSN 0164-0925. doi:10.1145/2743016.

[34] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. Signal - a data flow-oriented language for signal processing. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 34(2):362–374, 1986. doi:10.1109/TASSP.1986.1164809.

[35] Frank Hannig. *Scheduling Techniques for High-Throughput Loop Accelerators*. Dissertation, University of Erlangen-Nuremberg, Germany, August 2009. Verlag Dr. Hut, Munich, Germany.

[36] Ilkka Hautala, Jani Boutellier, Teemu Nyländen, and Olli Silvén. Toward efficient execution of RVC-CAL dataflow programs on multicore platforms. *Journal of Signal Processing Systems*, 90(11):1507–1517, Nov 2018. ISSN 1939-8115. doi:10.1007/s11265-018-1339-x.

[37] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL `http://dl.acm.org/citation.cfm?id=1624775.1624804`.

[38] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K. . Wu. IBM Streams Processing Language: Analyzing Big Data in motion. *IBM Journal of Research and Development*, 57(3/4):7:1–7:11, May 2013. ISSN 0018-8646. doi:10.1147/JRD.2013.2243535.

[39] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, August 1978. ISSN 0001-0782. doi:10.1145/359576.359585.

[40] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*, pages 159–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-44833-4. doi:10.1007/978-3-540-44833-4_6.

[41] François Irigoin and Rémi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi:10.1145/73560.73588.

[42] Kenneth E. Iverson. A programming language. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), pages 345–351, New York, NY, USA, 1962. ACM. doi:10.1145/1460833.1460872.

[43] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004. ISSN 0360-0300. doi:10.1145/1013208.1013209.

[44] Pierre Jouvelot and Yann Orlarey. Dependent vector types for data structuring in multirate Faust. *Computer Languages, Systems & Structures*, 37(3):113–131, 2011. doi:10.1016/j.cl.2011.03.001.

[45] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress in Information Processing*, volume 74, pages 471–475, 1974.

[46] Michal Karczmarek, William Thies, and Saman Amarasinghe. Phased scheduling of stream programs. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03,

pages 103–112, New York, NY, USA, 2003. ACM. ISBN 1-58113-647-1. doi:10.1145/780732.780747.

[47] R. Karp and R. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6): 1390–1411, 1966. doi:10.1137/0114108.

[48] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, jul 1967. ISSN 0004-5411. doi:10.1145/321406.321418.

[49] Joachim Keinert and Jürgen Teich. Buffer analysis for complete application graphs. In *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*, pages 151–208. Springer New York, New York, NY, 2011. ISBN 978-1-4419-7182-1. doi:10.1007/978-1-4419-7182-1_7.

[50] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi:10.1145/1250734.1250761.

[51] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL `http://dl.acm.org/citation.cfm?id=977395.977673`.

[52] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI signal processing systems for signal, image and video technology*, 3(3):173–182, sep 1991. ISSN 0922-5773. doi:10.1007/BF00925828.

[53] Jakob Leben. 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 2016, pages 17–28, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4432-6. doi:10.1145/2975980.2975983.

[54] Jakob Leben. Arrp: A functional language with multi-dimensional signals and recurrence equations: Auxiliary material, May 2019. URL https://doi.org/10.5281/zenodo.3236443.

[55] Jakob Leben. Polyhedral compilation for multi-dimensional stream processing: Experimental framework and data, April 2019. URL https://doi.org/10.5281/zenodo.2650704.

[56] Jakob Leben and George Tzanetakis. Polyhedral compilation for multi-dimensional stream processing. *ACM Transactions on Architecture and Code Optimization*, 2019. doi:10.1145/3330999. IN PRESS.

[57] Jakob Leben and George Tzanetakis. Polyhedral compilation for multi-dimensional stream processing: Addendum, April 2019. URL https://doi.org/10.5281/zenodo.2652490.

[58] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995. ISSN 0018-9219. doi:10.1109/5.381846.

[59] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, jan 1987. ISSN 0018-9340. doi:10.1109/TC.1987.5009446.

[60] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, sep 1987. ISSN 0018-9219. doi:10.1109/PROC.1987.13876.

[61] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3-4):649–671, may 1998. ISSN 01678191. doi:10.1016/S0167-8191(98)00029-5.

[62] James McCartney. Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002. doi:10.1162/014892602320991383.

[63] Benoit Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-Stream compiler. In *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi:10.1007/978-0-387-09766-4_515.

[64] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi:10.1145/2694344.2694364.

[65] Praveen K. Murthy and Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing*, 50(8):2064–2079, aug 2002. ISSN 1053-587X. doi:10.1109/TSP.2002.800830.

[66] Vesa Norilo and Pohjoinen Rautatiekatu. Introducing Kronos - a novel approach to signal processing languages. In *Proceedings of the Linux Audio Conference*, pages 9–16, 2011.

[67] Yann Orlarey, Dominique Fober, and Stéphane Letz. Faust: an efficient functional approach to DSP programming. In *New Computational Paradigms for Computer Music*, Musique-sciences. Delatour France, 2009.

[68] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL `http://ptolemy.org/books/Systems`.

[69] Miller Puckette. Pure Data: another integrated computer music environment. In *Proceedings of the International Computer Music Conference*, pages 37–41, 1996.

[70] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28 (5):469–498, Oct 2000. ISSN 1573-7640. doi:10.1023/A:1007554627716.

[71] Sanjay V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3(2):88–105, jun 1989. ISSN 1432-0452. doi:10.1007/BF01558666.

[72] Sanjay V. Rajopadhye and Richard M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14(2):163–189, 1990. ISSN 0167-8191. doi:10.1016/0167-8191(90)90105-I.

[73] Yukinori Sato, Tomoya Yuki, and Toshio Endo. An autotuning framework for scalable execution of tiled code via iterative polyhedral compilation. *ACM Trans. Archit. Code Optim.*, 15(4):67:1–67:23, January 2019. ISSN 1544-3566. doi:10.1145/3293449.

[74] SVEN-BODO SCHOLZ. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13 (6):1005–1059, 2003. doi:10.1017/S0956796802004458.

[75] Artjoms Sinkarovs and Sven-Bodo Scholz. A lambda calculus for transfinite arrays: Unifying arrays and streams. *CoRR*, abs/1710.03832, 2017. URL `http://arxiv.org/abs/1710.03832`.

[76] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In R. Nigel Horspool, editor, *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45937-8. doi:10.1007/3-540-45937-5_14.

[77] William Thies, Jasper Lin, and Saman Amarasinghe. Phased computation graphs in the polyhedral model. Technical report, MIT Laboratory for Computer Science, 2002.

[78] Baltasar Trancón y Widemann and Markus Lepper. The shepard tone and higher-order multi-rate synchronous data-flow programming in Sig. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, pages 6–14, New York, NY, USA, 2015. ACM. doi:10.1145/2808083.2808086.

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

[80] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010*, pages 299–302. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-15582-6_49.

[81] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Proceedings of the Second International Workshop on Polyhedral Compilation Techniques*, IMPACT '12, 2012.

[82] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. Pn: A tool for improved derivation of process networks. *EURASIP J. Embedded Syst.*, 2007(1):19–19, January 2007. ISSN 1687-3955. doi:10.1155/2007/75947.

[83] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1—-54:23, jan 2013. ISSN 1544-3566. doi:10.1145/2400682.2400713.

[84] Doran K. Wilde. The ALPHA language. Research Report RR-2295, INRIA, 1994. URL `https://hal.inria.fr/inria-00074378`.

[85] Zheng Zhou, William Plishker, Shuvra S. Bhattacharyya, Karol Desnos, Maxime Pelcat, and Jean-Francois Nezan. Scheduling of parallelized synchronous dataflow actors for multicore signal processing. *Journal of Signal Processing Systems*, 83(3):309–328, Jun 2016. ISSN 1939-8115. doi:10.1007/s11265-014-0956-2.